FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Departamento de Engenharia Eletrotécnica

REDES INTEGRADAS DE TELECOMUNICAÇÕES I

2014 / 2015

Mestrado Integrado em Engenharia Eletrotécnica
e Computadores

4º ano
7º semestre

Introdução à programação em rede para Java: Desenvolvimento de aplicações utilizando NetBeans e UDP

http://tele1.dee.fct.unl.pt

Luis Bernardo

ÍNDICE

1. Objetivo 3
2. A linguagem Java 3
  2.1. Tipos de dados básicos 3
    2.1.1. Números 3
    2.1.2. Strings 4
    2.1.3. Datas 4
    2.1.4. Constantes 5
  2.2. Tipos de dados estruturados 5
    2.2.1. Arrays 5
    2.2.2. Listas 6
    2.2.3. Arrays de tamanho variável 6
  2.3. Temporizadores 7
  2.4. Programação multi-tarefa 7
3. Programação de aplicações para uma rede IPv4 9
  3.1. A classe java.net.InetAddress 9
  3.2. Sockets Datagrama 9
    3.2.1. A classe DatagramPacket 10
    3.2.2. Composição e decomposição de mensagens 10
    3.2.3. A classe DatagramSocket 11
4. Programação de aplicações utilizando o ambiente NetBeans 12
  4.1 Criação do projeto 13
  4.2 Definição da interface gráfica 14
  4.3 Programação do envio e recepção de dados 15
  4.4 Exercícios 22
5. Bibliografia adicional 22

1. OBJETIVO

Familiarização com o ambiente NetBeans para Java e com o desenvolvimento de aplicações utilizando sockets UDP. Este documento descreve o método para a criação de aplicações com interface gráfica utilizando o ambiente de desenvolvimento NetBeans e o conjunto de classes e tipos de dados estruturados disponíveis em Java para a programação de aplicações em rede. Adicionalmente, é apresentado o código integral de uma aplicação que deve ser introduzido seguindo as instruções do enunciado, facilitando a aprendizagem do ambiente.

2. A LINGUAGEM JAVA

A linguagem Java é uma linguagem orientada por objetos que tem uma sintaxe semelhante à linguagem C++, com uma grande excepção: não existem ponteiros. As aplicações desenvolvidas em Java são vulgarmente compiladas para *bytecode*, que é depois interpretado em máquinas virtuais Java (ou compilado para código nativo num compilador JIT). A grande vantagem da plataforma Java é precisamente a possibilidade de correr em qualquer arquitetura (*Write Once, Run Anywhere*).

O trabalho proposto nesta disciplina vai ser desenvolvido para a versão 6 de Java, com um vasto conjunto de classes e interfaces na biblioteca. Na realidade, para realizar o trabalho proposto apenas vai ser necessário recorrer a um subconjunto reduzido dessas classes da biblioteca, apresentado neste documento. Os alunos podem, no entanto, usar qualquer outra classe disponível na biblioteca. Para ambiente de desenvolvimento propõe-se a utilização do NetBeans da Sun Microsystems, totalmente desenvolvido em Java, que pode correr em qualquer sistema operativo. Para além disso, está disponível para utilização graciosa em http://java.sun.com/. Existem outras alternativas igualmente interessantes, como o Eclipse (http://www.eclipse.org/), da IBM, incluído na distribuição Fedora ou Ubuntu de Linux.

2.1. Tipos de dados básicos

A linguagem Java define um conjunto de tipos primitivos para a definição de valores numéricos, carateres, datas, etc.

2.1.1. Números

A linguagem Java define um tipo inteiro básico para cada tipo de número, semelhante aos tipos equivalentes em C++, exceptuando o facto de o número de bits estar normalizado. Adicionalmente, a linguagem define uma classe que estende o tipo primitivo com um conjunto de métodos de conversão de formatos, etc.

Tipo primitivo	Classe estendida	Tipo de número
boolean		Inteiro com 1 bit
byte	Byte	Inteiro com 8 bits
short	Short	Inteiro com 16 bits
int	Integer	Inteiro com 32 bits
long	Long	Inteiro com 64 bits
float	Float	Fraccionário com 32 bits
double	Double	Fraccionário com 64 bits

Uma funcionalidade incluída nestas classes estendidas é a conversão de uma *string* para o número, que pode ser realizada utilizando o método "`parse???`" (parseByte, parseShort, etc., dependendo da classe). Caso a *string* não corresponda a um número válido é gerada a excepção *NumberFormatException*.

```
int n;
String str= "123";
try {
     n= Integer.parseInt(str);
}
catch (NumberFormatException e) {
     System.out.println("Número inválido"+e+"\n");
}
```

A operação inversa pode ser realizada utilizando os métodos "`toString`" na classe estendida correspondente. Contudo, é possível usar uma conversão automática para *strings*, por exemplo, realizando a concatenação do número com uma *string* vazia (e.g. `""+n`).

2.1.2. *Strings*

O Java usa duas classes distintas para representar *strings*:

- A classe `String` representa as *strings* num formato não editável;
- A classe `StringBuffer` representa as *strings* como uma cadeia de carateres editável.

A construção de *strings* pode ser feita a partir da adição dos vários componentes, sendo aceites vários tipos de parâmetros. Neste caso, a linguagem Java realiza a construção da *string* final utilizando uma variável temporária do tipo *StringBuffer*, que é convertida para `String` (utilizando o método `toString`) após a adição do último componente.

Na linguagem Java, os carateres são representados pelo tipo `char`, no formato Unicode, utilizando 16 bits. É possível obter o caráter na posição *i* da *string* utilizando o método `charAt`.

```
char c= str.charAt(0);       // Primeiro carater da string;
```

As classes *String* e *StringBuffer* oferecem o método *substring* para seleccionar uma parte da string:

```
String a= "Java is great";
String b= a.substring(5);          // b is the string "is great"
String c= a.substring(0, 4);       // c is the string "Java"
```

O método *trim()* remove os espaços e tabulações antes e depois da string.

A comparação de strings pode ser feita com os métodos *equals* ou *equalsIgnoreCase*.

2.1.3. Datas

A linguagem Java utiliza vulgarmente a classe `Java.util.Date` para representar datas, com uma precisão de até dezenas de milisegundos.

É possível obter a data atual criando um novo objeto:

```
Date dNow = new Date();
```

A escrita de datas para uma *string* e a leitura de datas a partir de *strings* é realizada utilizando um objeto da classe `java.text.SimpleDateFormat`. Antes de realizar qualquer destas operações é criado um objeto de formação onde se define o formato de representação da data.

```
SimpleDateFormat formatter = new SimpleDateFormat
                                          ("E hh:mm:ss 'em' dd.MM.yyyy");
```

Para escrever a data é usado o método *format*:

```
System.out.println("A data atual é " + formatter.format(dNow));
```

Para ler uma data neste formato a partir de uma *string* pode ser usado o seguinte código:

```
try {
     dNow= formatter.parse(str);
} catch (ParseException e) {
     System.out.println("Data inválida: " + e + "\n");
}
```

Uma variável do tipo `Date` também pode ser representada utilizando o tipo `long`. Utilizando o método `getTime`, obtém-se o número de milisegundos em relação a uma data de referência. O construtor da classe `Date` aceita argumentos do tipo `long`, conseguindo-se, desta forma, criar uma data a partir de um valor do tipo `long`.

```
long t= dNow.getTime();     // Data no formato long (n° de milisegundos)
Date nData= new Date(t);    // Data novamente no formato Date
```

Utilizando a representação no formato `long` é possível calcular diferenças entre datas.

2.1.4. Constantes

Em Java, as constantes são declaradas como variáveis de uma classe precedidas do modificador **final**. Por exemplo:

```
final int N= 1;
```

2.2. Tipos de dados estruturados

O desenvolvimento de uma aplicação em Java obriga ao desenvolvimento de pelo menos uma classe com uma função `main`, mas vulgarmente leva ao desenvolvimento de mais classes, que encapsulam dados e definem vários métodos para lidar com esses dados. Os dados têm neste contexto um significado genérico, que tanto engloba tipos primitivos como objetos de uma classe da biblioteca, ou de uma classe definida pelo programador. Estes dados podem, por sua vez, ser organizados em vários tipos de dados estruturados.

2.2.1. *Arrays*

O tipo de estrutura mais simples é a tabela (*array*). Em Java é possível definir um *array* multi-dimensional de qualquer tipo.

```
byte[] arrayBytes;    // Declaração de vetor vazio; não aloca memória
arrayBytes= new byte[40];   // inicialização da memória para o vetor
int[] matriz;   // Declaração de matriz bidimensional
matriz = new int[8][8];                              // inicialização 1
matriz = new int[8][];                               // inicialização 2
for (int i= 0; i<8; i++) matriz[i]= new int [8];     // ambas são equivalentes
```

Ao contrário do C++, em Java os *arrays* são tipos bem definidos, existindo campos para testar dimensão do *array* (e.g. `arrayBytes.length`). Sempre que se acede a um *array*, o ambiente Java testa se o índice está dentro dos limites, gerando uma excepção do tipo `ArrayIndexOutOfBoundsException` em caso de falha.

Os *arrays* têm uma dimensão fixa, só podendo ser modificados através da criação de um novo *array* e da cópia integral dos dados. Observe-se que a libertação da memória libertada é realizada automaticamente, após esta deixar de estar referenciada por uma variável.

```
byte[] aux;     // Declaração de novo vetor; não aloca memória
aux= new byte[80];    // inicialização da memória para o vetor
System.arraycopy(arrayBytes, 0, aux, 0, arrayBytes.length); // copia array
arrayBytes= aux; // substitui array, a memória antiga vai ser libertada pelo sistema
```

Observe-se que embora não existam ponteiros na linguagem Java, todas as referências para um objeto ou para um *array* são equivalentes a um ponteiro, podendo ser atribuídas à constante **`null`**, e só podendo ser usadas após alocar memória utilizando a função `new`. No caso de um array de uma classe Entry, depois de alocar o array é necessário alocar a memória para os elementos individuais do array, um a um:

```
Entry[] vec;    // Declaração de vetor vazio; não aloca memória
vec= new Entry [40];  // inicialização da memória para o vetor, com
                //  elementos todos a null
for (int i= 0; i<vec.length; i++)
     vec[i]= new Entry();   // Aloca memória para os elementos do array
```

2.2.2. Listas

Existem vários tipos disponibilizados na *framework collections* para lidar com listas. Entre eles existe um que permite lidar de uma forma eficaz com conjuntos de pares (nome propriedade, valor de propriedade) para valores de tipo arbitrário - a classe *HashMap*. Desde a versão 1.5 do Java que esta classe é usada como um template, definindo a classe da chave da lista, e a classe dos dados da lista Os elementos do vetor são de um tipo `Tipo` genérico, definido no segundo parâmetro do template <*Tipo_chave, Tipo_dados*>. Podem-se acrescentar ou remover elementos por nome e pode-se pesquisar por nome de propriedade ou exaustivamente.

```
Tipo val;                                                  // Objeto de tipo Tipo

HashMap<Character,Tipo> h= new HashMap<Character,Tipo> ();      // Cria lista vazia
h.clear();                                                 // Limpa lista
h.put('A', val);                                           // acrescenta val à lista;
                                                           //   se já existe substitui
Tipo val= h.get('A');                                      // obtém valor
for (Iterator<Tipo> it= h.values().iterator(); it.hasNext();)    // percorre lista
     val= it.next();                                       // Obtém próximo objeto da lista
   ou
for (Tipo val : h.values()) { ... val ... }                // Para Java > 6
```

2.2.3. *Arrays* de tamanho variável

Caso seja necessário lidar com matrizes cujo tamanho seja difícil de determinar, ou que varie continuamente, podem ser usados vários tipos de dados disponíveis na biblioteca Java. Note-se que estes métodos são mais pesados computacionalmente do que o tipo *array*.

Dois exemplos são o tipo `ArrayList<Tipo>` (novamente um template com um tipo genérico), ou o tipo `Vector<Tipo>`. Nestes casos, o acesso aos elementos é feito através de um conjunto de métodos da interface destas classes. Os elementos do vetor são do tipo definido nos parâmetros do template. Algumas das funções disponibilizadas por estes tipos são:

Assinatura do método	**Descrição**
add(Object o)	Adicionar objeto ao fim da lista

add(int i, Object o)	Adicionar objeto na posição *i* da lista
clear()	Remove todos os elementos da lista
get(int i)	Retorna referência para objeto na posição *i*
remove(int i)	Elimina objeto na posição *i*
toArray()	Retorna *array* com os objetos da lista

2.3. Temporizadores

A biblioteca de classes da linguagem Java inclui várias classes que podem funcionar como temporizadores, isto é, que após serem ativadas ficam inativas durante um intervalo de tempo programável, após o que correm uma função. Uma das que tem uma interface mais simples é a classe `javax.swing.Timer`. Para usar esta classe, é conveniente incluir as classes debaixo da diretoria `javax.swing` e `Java.awt.event` utilizando o seguinte código:

```
import javax.swing.*;
import java.awt.event.*;
```

A declaração da função de tratamento do temporizador pode ser realizada utilizando uma declaração compacta de um método dentro do argumento de uma função, durante a criação do objeto temporizador. Observe-se que o segundo argumento do construtor da classe `javax.swing.Timer` é a invocação do construtor da interface `ActionListener`, que recebe como argumento a declaração da função de tratamento do expirar do tempo:

```
timer= new javax.swing.Timer(delay/*msec*/, new ActionListener() {
     public void actionPerformed(ActionEvent evt) {
          // Código de tratamento do temporizador
     }
});
```

A variável `timer` deve ser declarada numa classe da aplicação.

```
javax.swing.Timer timer;
```

O temporizador inicia a contagem de tempo após ser invocado o método `start`.

```
timer.start();
```

O temporizador pode ser interrompido usando o método `stop`.

```
timer.stop();
```

A classe `javax.swing.Timer` oferece outros métodos que permitem reiniciar o temporizador (método `restart`), definir se corre uma vez ou funciona periodicamente (método `setRepeats`), modificar o tempo de espera (`setDelay`), etc. Pode descobrir com facilidade os métodos disponíveis escrevendo no NetBeans "`timer.`", onde `timer` é uma variável da classe.

2.4. Programação multi-tarefa

Outra diferença significativa da linguagem Java em relação à C++, é que suporta de raiz o paralelismo entre tarefas (*threads*). Isto quer dizer que o modelo normal de programação em Java não se baseia num núcleo síncrono único que invoca funções de tratamento de eventos do programador (e.g. nos trabalho de Sistemas de Telecomunicações em Delphi). Em Java, é

comum lançar vários objetos a correr em paralelo para receber eventos de cada uma das fontes. Esta funcionalidade é vulgarmente realizada criando classes que herdam da classe `Thread`, e que realizam a função `run`, corrida pelo sistema.

Na figura seguinte é exemplificada a estrutura da declaração de uma classe que realiza uma tarefa. A classe deve incluir um construtor, que inicia todas as variáveis locais da classe, e um método `run`. O método `run` é vulgarmente um ciclo controlado por uma variável de controlo (no caso `keepRunning`), onde se invoca uma operação bloqueante (e.g. leitura de dados de um *socket*). No fim do ciclo é vulgar passar o controlo para outra tarefa (`yield`), de forma a maximizar o paralelismo.

Embora a classe `Thread` disponibilize um método `stop`, este não deve ser usado para parar uma tarefa pois foi classificado na versão de Java 1.4 como DEPRECATED, isto é, desatualizada. Em sua substituição, deve ser usada uma função da classe (e.g. `stopRunning`), que modifique a variável de controlo do ciclo principal da tarefa.

```
public class Daemon extends Thread {

    volatile boolean keepRunning= true;
    // Parametros adicionais

    /** Creates a new instance of Daemon */
    public Daemon(/*argumentos*/) {
        // Inicialização de parametros a partir de argumentos
    }

    /** Runned function */
    public void run() {
        // Inicializações antes do ciclo
        while (keepRunning) {
            // Código do ciclo
            this.yield();   // Passar controlo para outra thread
        }
    }

    // Stops thread running safely
    public void stopRunning() {
        keepRunning= false;
    }
}
```

A tarefa pode ser criada e arrancada em qualquer função utilizando o seguinte excerto de código:

```
Daemon daemon= new Daemon(/*argumentos*/);
daemon.start();
```

O facto de existirem várias tarefas a correr em paralelo pode levantar diversos problemas de sincronismo no acesso a objetos partilhados por diversas tarefas. Por exemplo, as várias caixas de texto da interface gráfica, ou os elementos de um `array`. Para evitar problemas de acesso concorrente a objetos, a linguagem Java oferece diversos mecanismos para garantir que o acesso a uma função ou objeto só é realizado por uma tarefa de cada vez. O método mais simples é a classificação de métodos de classes com a palavra-chave "**synchronized**", que bloqueia todos os métodos do objeto. Outra alternativa é a definição de zonas críticas do código, definindo uma operação de *lock* implícita sobre uma variável Java também utilizando a palavra-chave **synchronized**. Por exemplo:

```
public Integer main_lock= new Integer(0);  // Synchronization lock
…
synchronized (main_lock) {
     … código crítico …
}
… código normal …
```

3. PROGRAMAÇÃO DE APLICAÇÕES PARA UMA REDE IPV4

A linguagem Java inclui um conjunto de classes no pacote `java.net.*` que suporta o desenvolvimento de aplicações em rede, tanto para IPv4 como para IPv6. Nesta seção são apresentadas apenas as classes mais importantes para a programação de aplicações baseadas em *sockets* datagrama para redes IPv4.

3.1. A classe `java.net.InetAddress`

A classe *InetAddress* permite lidar com endereços IPv4, suportando a conversão entre vários formatos. Suporta ainda a identificação do endereço IP da máquina local.

Um endereço IP pode ser representado por:

- uma *string* com um nome (e.g. "tele1.dee.fct.unl.pt");
- uma *string* com um endereço (e.g. "193.136.127.217");
- um *array* de 4 bytes com o endereço (e.g. byte[] end = {193, 136, 127, 217};

Qualquer um dos três formatos anteriores é suportado pela classe `InetAddress`, que internamente contém dois componentes privados: o nome (*string*) e o endereço (*array* de bytes).

Um endereço é vulgarmente inicializado a partir de uma *string*, através da função `getbyName`:

```
String str= …;
InetAddress netip;
try {
    netip= InetAddress.getByName(str);
} catch (UnknownHostException e) {
    // Tratar excepção
}
```

Alternativamente, pode-se inicializar uma variável do tipo `InetAddress` com o endereço da máquina local utilizando o método estático `getLocalHost` da classe `InetAddress`:

```
try {
    InetAddress addr = InetAddress.getLocalHost();
} catch (UnknownHostException e) {
    // tratar excepção
}
```

O método `getHostAddress` permite obter o endereço em formato *string*. O nome associado ao endereço pode ser obtido com o método `getHostName`.

3.2. *Sockets* Datagrama

Na linguagem Java, a interface para *sockets* datagrama é composta por duas classes: a classe `DatagramPacket` e a classe `DatagramSocket`.

3.2.1. A classe DatagramPacket

A classe `DatagramPacket` encapsula um pacote de dados. Esta classe inclui, como dados internos, o conteúdo da mensagem, um endereço IP e o número de porto, que podem conter o *socket* de destino ou de origem, conforme a mensagem seja enviada ou recebida.

Existem dois construtores para a classe `DatagramPacket`. O primeiro construtor limita-se a inicializar os dados da mensagem com o conteúdo de um *array* de bytes. O segundo construtor também inicia o endereço IP e porto de destino. Observe-se que o valor do endereço IP e número de porto pode ser modificado *a posteriori* usando os métodos `setAddress` e `setPort`.

```
byte[] data= …;
DatagramPacket dp;
dp= new DatagramPacket(data, data.length);          // Construtor 1
OU
try {
    dp= new DatagramPacket(data, data.length, addr, porto); // Constr. 2
} catch (UnknownHostException e) {
    // tratar exceção
}
```

A classe define adicionalmente métodos para obter os valores dos vários campos internos:

- `InetAddress getAddress()` : devolve o endereço IP
- `int getPort()` : devolve o número de porto
- `byte[]getData()` : devolve um *buffer* com a mensagem
- `int getLength()` : devolve número de bytes contidos na mensagem

3.2.2. Composição e decomposição de mensagens

O conteúdo das mensagens pode ser criado a partir do preenchimento byte a byte, diretamente no *array* de bytes. No entanto, este método é pouco flexível e de difícil aplicação. Outra maneira será usar objetos dos tipos `DataInputStream` e `DataOutputStream` respetivamente para decompor a mensagem em componentes e para compor a mensagem a partir dos componentes.

A figura seguinte ilustra a operação de iniciação do objeto de serialização para escrita de uma mensagem. O objeto `DataOutputStream` é criado a partir de um objeto `ByteArrayOutputStream`, disponibilizando um conjunto de funções para escrever o conteúdo de objetos de vários tipos. No fim, é usado um método do objeto `BAos` que retorna a totalidade do *buffer* criado.

```
ByteArrayOutputStream BAos= new ByteArrayOutputStream();
DataOutputStream dos= new DataOutputStream(BAos);
try {
    // Escrita sequencial dos componentes. Exemplos de funções:
    //     dos.writeBytes(str); - escreve string
    //     dos.writeShort(n); - escreve short
} catch (IOException e) {
    // tratar excepção
}
byte [] buffer = BAos.toByteArray();    // Cria array de bytes com os dados
```

Na recepção de dados pode ser utilizado um excerto de código dual. Após receber um *array* de bytes, é usado um objeto `DataInputStream` para ler a mensagem campo a campo.

```
ByteArrayInputStream BAis= new ByteArrayInputStream(buf, 0, len);
DataInputStream dis= new DataInputStream(BAis);
try {
    // Escrita sequencial dos componentes (exemplos de funções)
    //    byte [] aux= new byte [len];              // ler len bytes
    //    int n= dis.read(aux,0,len);          //   para uma
    //    String str= new String(aux, 0, len);      //    string
    //      em alternativa, poder-se-ia ler diretamente de buf.
    //    int len_msg= dis.readShort();             // Ler short
} catch (IOException e) {
    // tratar excepção
}
```

3.2.3. A classe DatagramSocket

A classe `DatagramSocket` permite criar *sockets* datagrama associados a um número de porto, e enviar e receber pacotes. A associação a um porto é realizada no construtor da classe. Existem três variantes de construtor, representadas na figura seguinte. O construtor 1 inicia um socket num porto indefinido; o construtor 2 tenta iniciá-lo num porto definido; o construtor 3 define o porto e o endereço IP da interface, para o caso de existirem várias interfaces de rede.

```
public DatagramSocket() throws SocketException            // Construtor 1
public DatagramSocket(int port) throws SocketException    // Construtor 2
public DatagramSocket(int port, InetAddress intf) throws SocketException
                                                          // Construtor 3
```

Qualquer dos construtores pode gerar uma excepção `SocketException`, indicando que não foi possível criar o *socket* porque, por exemplo, o número de porto pretendido não está livre. É possível obter o número de porto associado ao *socket* usando o método `getLocalPort()`. O porto associado ao socket é libertado usando o método `close()`.

Após iniciar um *socket*, é possível enviar pacotes usando o método `send`. Caso falhe o envio do pacote é gerada a excepção `IOException`. Observe-se que o endereço IP e o número de porto são preenchidos no objeto `DatagramPacket`.

```
DatagramSocket ds= new DatagramSocket();
DatagramPacket dp = …              // prepara pacote e define IP e porto
try {
    ds.send(dp);
} catch (IOException e) {
    System.err.println("Excepção :"+e);
}
```

Os pacotes são recebidos usando o método `receive`. Caso não seja possível receber pacotes, é gerada a excepção `IOException`. O endereço IP e o número de porto de origem do pacote são guardados no objeto `DatagramPacket`.

```
DatagramSocket ds;
try {
    ds = new DatagramSocket(porto);           // Define porto
} catch (SocketException e) {
    System.err.println("Falhou criação de socket: "+e);
}
byte [] buf= new byte[PACKET_SIZE];     // Tamanho máximo teórico é 65535
DatagramPacket dp= new DatagramPacket(buf, buf.length);
try {
    ds.receive(dp);
    int len= dp.getLength();      // dimensão do pacote recebido
    // buf contém len bytes de dados recebidos - processar pacote
} catch (IOException e) {
    // Tratar erro
}
```

Por omissão, a operação de leitura é bloqueante, esperando indefinidamente por pacotes enquanto o *socket* for válido. No entanto, usando o método `setSoTimeout` é possível definir um tempo máximo de espera por pacotes. Esta operação pode gerar a excepção `IOException` em caso de erro. Caso expire o tempo máximo de espera é gerada a excepção `SocketTimeoutException`.

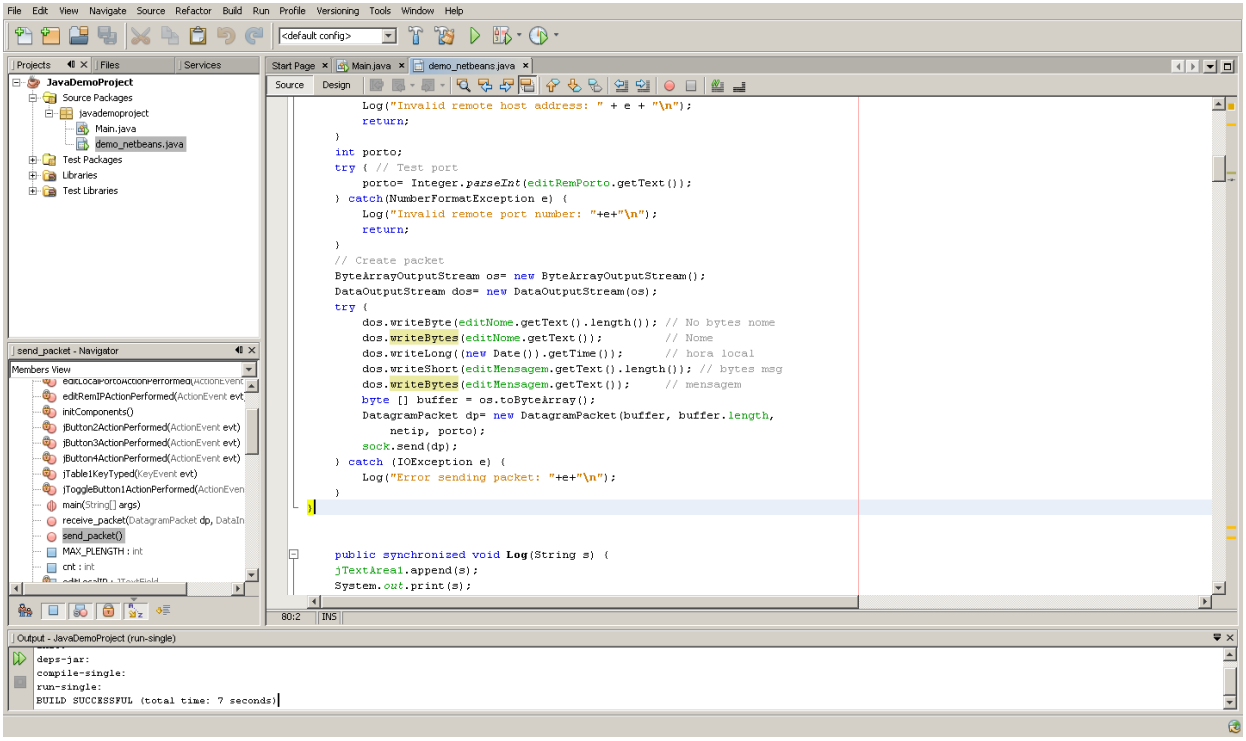
Para além da classe `DatagramSocket`, o JDK define uma classe semelhante (`MulticastSocket`) que apenas difere em dois aspetos: permite associar um socket a endereço de grupo (*multicast*), e permite partilhar um número de porto por vários sockets.

```
ds.setSoTimeout(3000);      // 3 segundos
try {
    ds.receive(dp);
    // processa o pacote
} catch (SocketTimeoutException e) {
    System.err.println("No packet within 3 seconds");
}
```

4. PROGRAMAÇÃO DE APLICAÇÕES UTILIZANDO O AMBIENTE NETBEANS

O sistema Java inclui vários pacotes para a criação da interface gráfica de aplicações. As duas mais conhecidas são a `java.awt.*` e `javax.swing.*`. É possível programar aplicações diretamente num editor de texto, mas é conveniente a utilização de um ambiente integrado de desenvolvimento de aplicações que integre um editor de interfaces gráficas, editor de texto, compilador e *debugger*. Escolheu-se para esta disciplina o ambiente NetBeans devido às vastas funcionalidades disponibilizadas, à compatibilidade com Linux e Windows, e devido ao preço (disponível na web a partir da página da Sun Microsystems juntamente com o JDK em http://www.java.com).

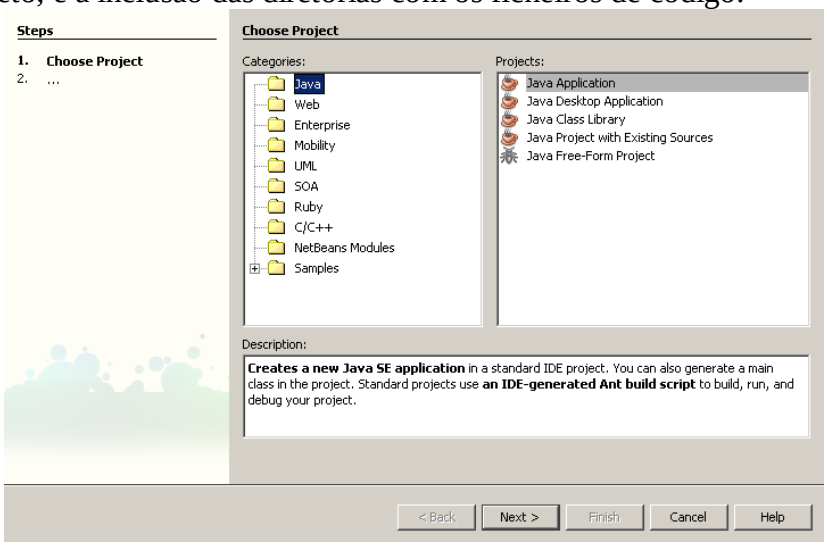
O ambiente NetBeans contém várias janelas que suportam um conjunto variado de opções. Existem quatro modos principais de funcionamento: "*Editing*" (edição de texto), "*GUI Editing*" (edição da interface gráfica), "*Running*" (a correr aplicações); e "*Debugging*" (a correr passo a passo, ilustrada na figura seguinte). Paralelamente, existem várias opções realizáveis através de opções do menu, ou de botões.



Nesta seção vai ser apresentado o desenvolvimento de uma aplicação de demonstração das funcionalidades do Java, passo a passo, que se recomenda que os alunos introduzam no ambiente NetBeans, como forma de aprendizagem. Na parte final, são apresentados alguns exercícios que se pretende que sejam realizados pelos alunos.

4.1 Criação do projeto

O primeiro passo para a criação de um projeto em NetBeans é a seleção de uma diretoria base para o projeto, e a inclusão das diretorias com os ficheiros de código.



Passos a realizar pelos alunos:

Passo 1: Criar uma diretoria *demo_netbeans*, com uma subdiretoria *src* para o código;
Passo 2: Criar um projeto novo (*General* / *Java Application*) baseado nas diretorias.

4.2 Definição da interface gráfica

A construção da aplicação vai ser realizada a partir da interface gráfica, cuja classe vai conter a função `main`. O primeiro passo na definição da interface gráfica é a criação da janela principal, do tipo `JFrame Form`, na pasta `Java GUI Forms`, com o nome `demo_netbeans`. Posteriormente, passando para o modo de edição de *forms*, é desenhada a interface gráfica da aplicação representada abaixo.

A janela é composta por quatro subjanelas:

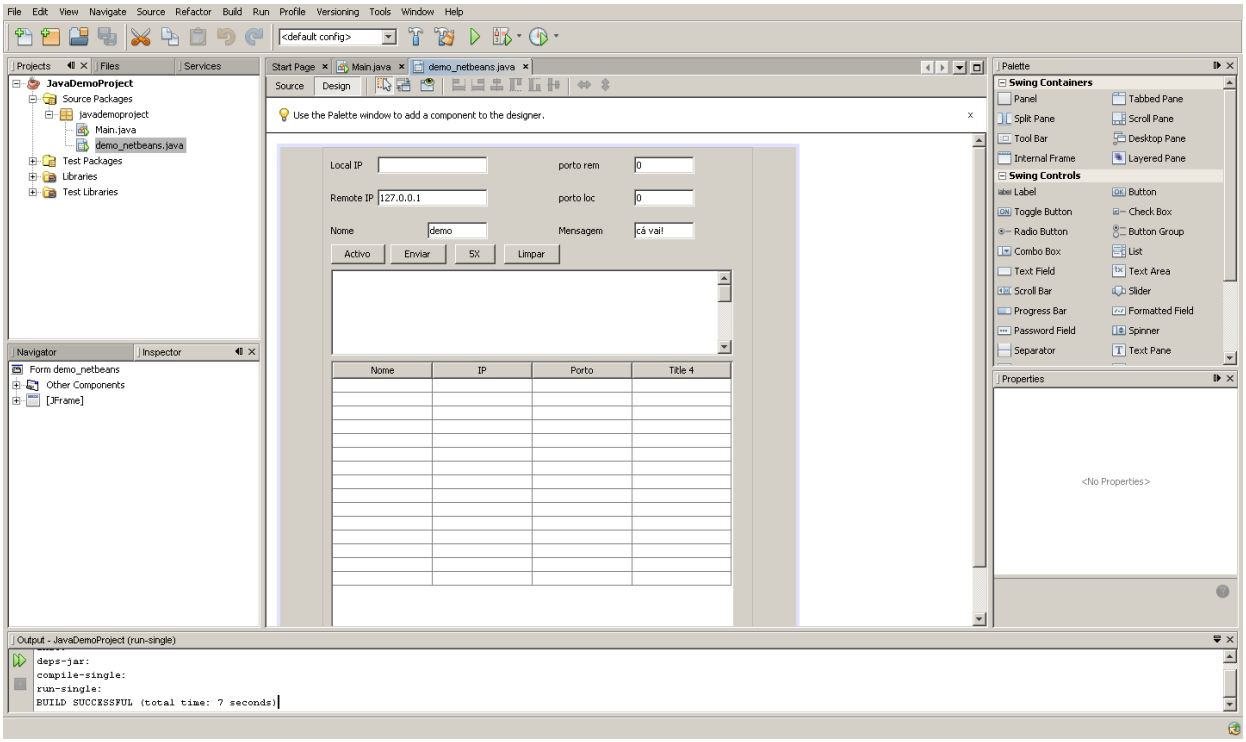
- No canto superior direito existe uma lista de objetos gráficos que podem ser acrescentados à janela;
- Na segunda janela do lado direito existe uma representação em árvore de todos os componentes da janela;
- Na terceira janela do lado direito são representadas as propriedades do objeto seleccionado na representação em árvore.
- No lado esquerdo é representada o aspeto gráfico da janela que está a ser desenhada;

A interface gráfica é construída adicionando componentes, e em paralelo, editando os parâmetros de cada componente gráfico.

Passos a realizar pelos alunos:

Passo 3: Criar a classe principal seleccionando "*New File*" no menu "*File*", e o template "*Swing GUI Forms*"/ "*JFrame Form*" – criar janela com nome "*demo_netbeans*";

Passo 4: Dividir num primeiro nível hierárquico a janela em três partes: seleccionar "*BoxLayout*" (com "*Set Layout*" na tecla da direita do rato seleccionando a janela raiz), e acrescentar um `JPanel` e dois `JScrollPane` (pode usar a função "*Add From Palette*" na tecla da direita do rato seleccionando a janela raiz), colocando uma `JTextArea` e uma `JTable` respetivamente nos dois JScrollPanes.



Passo 5: Editar propriedades: *BoxLayout* (*Axis=Y Axis*); *jPanel1* (*maximumSize= minimumSize= preferredSize=* [300,120]); *jScrollPane1* e *jScrollPane2* (*maximumSize= minimumSize= preferredSize=* [300,100]); *jTextArea1* (*preferredSize*=[280,300]); *jTable1* (*Other Properties*:*preferredSize*=[300,300]); *JFrame* (*title*="demo_netbeans").

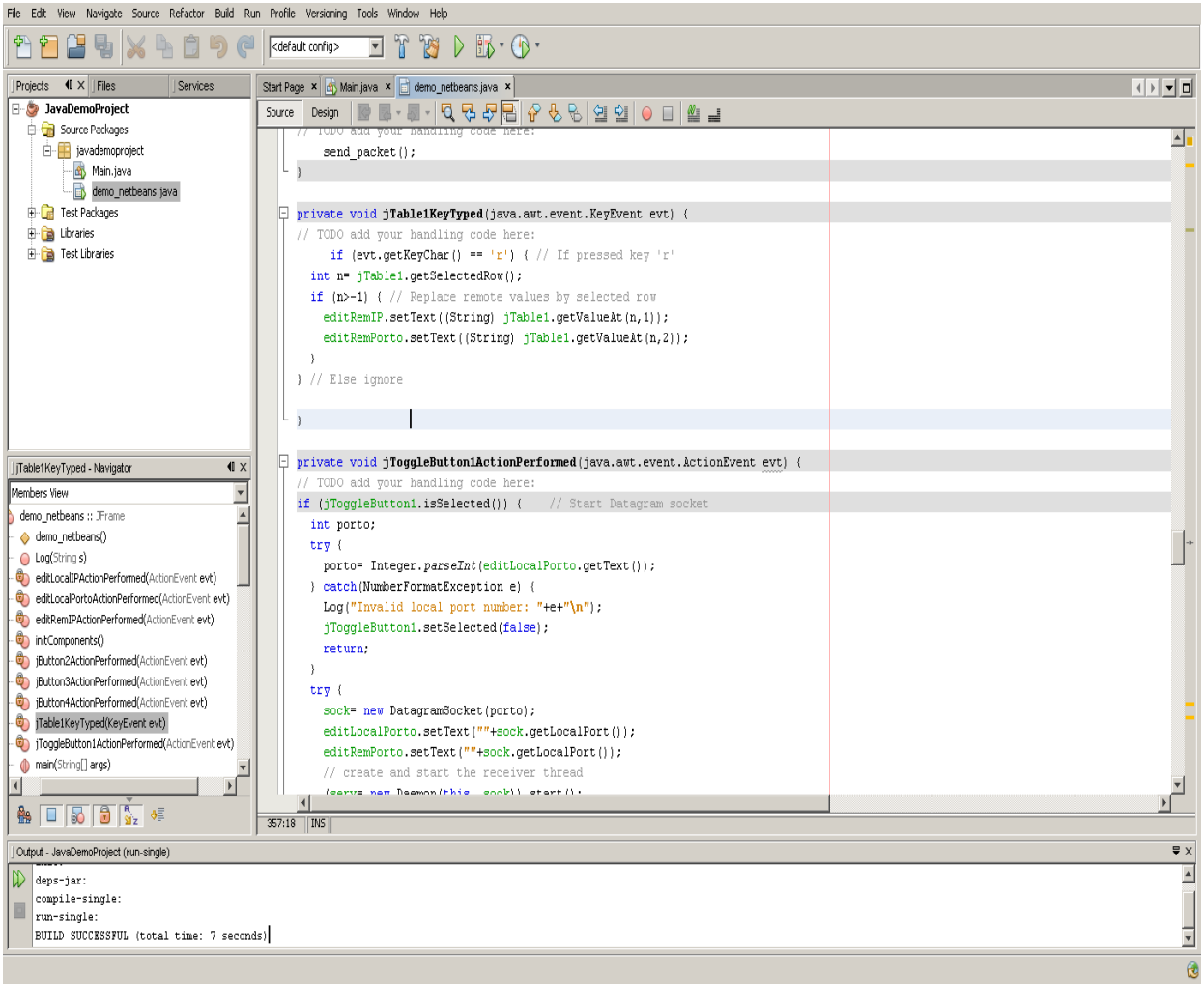
Passo 6: Colocar dentro do *jPanel1* objetos do tipo *JLabel*, *JTextField*, *JButton* e *JToggleButton* ("Activo"), de acordo com o que está representado na figura. Modificar nome das *jTextField* para *editLocalIP*, *editLocalPorto*, *editRemIP*, *editRemPorto*, *editNome*, *editMensagem*.

Passo 7: Editar conteúdo da tabela (*Properties:model* – seleccionar **...**) definindo 4 colunas e 15 linhas, com os títulos representados na figura.

4.3 Programação do envio e recepção de dados

A parte final do desenvolvimento de uma aplicação corresponde à programação das rotinas de tratamento de eventos e de todas as classes de apoio necessárias para realizar a aplicação. Todas as funções associadas a eventos gráficos devem ser criadas premindo o botão do rato sobre o elemento, seleccionando um evento na pasta "*Events*". Outras classes e variáveis devem ser adicionadas com a janela de edição de texto escrevendo diretamente o código, ou usando as funções disponibilizadas pelo NetBeans a partir da pressão do botão direito do rato sobre a janela "*Filesystems*".

A figura abaixo representa a janela de edição de texto do NetBeans, com uma visão hierárquica do código. Marcado a azul aparece código gerado pelo IDE que não pode ser modificado. As palavras reservadas aparecem em letras azuis, os comentários com letras cinzentas, e as strings aparecem com letras vermelhas.



Relativamente à aplicação de exemplo, conversa em rede, falta programar todas as classes e rotinas de tratamento de eventos. Pretende-se uma aplicação que envia mensagens

(manualmente ou com um temporizador) e que em paralelo recebe mensagens vindas da rede. Assim, deve-se:

- Incluir instruções "`import`" para todos os pacotes usados nas classes;
- Declarar todas as variáveis adicionais na classe principal;
- Declarar todas as classes auxiliares (e.g. tarefa para receber dados);
- Modificar o construtor da classe principal de maneira a arrancar todos os objetos.

Passos a realizar pelos alunos:

Passo 8: Adicionar, no início do ficheiro, a lista de pacotes usados na aplicação:

```
import java.net.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import java.awt.event.*;
```

Passo 9: Adicionar no fim da declaração da classe `demo_netbeans` a lista de variáveis adicionais usadas:

```
private DatagramSocket sock;      // Socket datagrama
private Daemon serv;              // Thread para recepção de dados
private int last_line;            // última linha preenchida na tabela
private javax.swing.Timer timer;  // Temporizador
public volatile int cnt;          // Contador
public final int MAX_PLENGTH= 8096;     // Tamanho máximo de pacote
```

A variável `last_line` memoriza qual é a última linha preenchida da tabela.

Passo 10: Adicionar o método `Log` à classe `demo_netbeans` para escrever a `string` recebida como parâmetro na `jTextArea1` e na linha de comando. O método é declarado `synchronized` de forma a poder ser acedido de várias tarefas em paralelo sem haver problemas:

```
public synchronized void Log(String s) {
    jTextArea1.append(s);   // Escreve na janela
    System.out.print(s);    // Ecoa os carateres para o terminal
}
```

Passo 11: Adicionar um método para tratar o premir do botão "**Limpar**" através de um duplo *click* na janela de edição de *forms*. Posteriormente, editar o texto gerado de forma a limpar a `jTextArea1` e `jTable1`. Caso abra o ficheiro noutro editor de texto verifica o Netbeans adiciona ao texto diretivas para o editor (`//GEN-FIRST:event_jButton3ActionPerformed` e `//GEN-LAST:event_jButton3ActionPerformed`), para definir as zonas editáveis.

```
private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
    jTextArea1.setText(""); // Clean jTextArea1
    for (int i= 0; i<last_line; i++)
        for (int j= 0; j< jTable1.getColumnCount(); j++)
            jTable1.setValueAt("",i,j);    // Cleans jTable1
    last_line= 0;  // Reset last_line
}
```

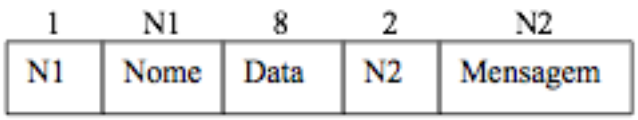
Passo 12: Modificar o construtor da classe `demo_netbeans` de forma a iniciar as variáveis da classe e a iniciar as caixas de edição `editLocalIP`, `editLocalPorto` e `editRemIP`:

```
public demo_netbeans () {
    initComponents();
    sock= null;
    serv= null;
    last_line= 0;
    timer= null;
    try {
        // Get local IP and set port to 0
        InetAddress addr = InetAddress.getLocalHost();
        editLocalIP.setText(addr.getHostAddress());
        editRemIP.setText(addr.getHostAddress());
    } catch (UnknownHostException e) {
        Log("Unable to determine local IP address: " + e + "\n");
        System.exit(-1);
    }
    editLocalPorto.setText("0");
}
```

Passo 13: Adicionar o método `send_packet` à classe `demo_netbeans` para enviar uma mensagem para o endereço IP e porto contidos nas caixas de texto `editRemIP` e `editRemPorto`:

```
public synchronized void send_packet() {
    if (sock == null) {
        Log("Socket isn't active!\n");
        return;
    }
    InetAddress netip;
    try { // Test IP address
        netip= InetAddress.getByName(editRemIP.getText());
    } catch (UnknownHostException e) {
        netip= null;
        Log("Invalid remote host address: " + e + "\n");
        return;
    }
    int porto;
    try { // Test port
        porto= Integer.parseInt(editRemPorto.getText());
    } catch(NumberFormatException e) {
        Log("Invalid remote port number: "+e+"\n");
        return;
    }
    // Create packet
    ByteArrayOutputStream os= new ByteArrayOutputStream();
    DataOutputStream dos= new DataOutputStream(os);
    try {
        dos.writeByte(editNome.getText().length());       // N° bytes nome
        dos.writeBytes(editNome.getText());         // Nome
        dos.writeLong((new Date()).getTime());      // hora local
        dos.writeShort(editMensagem.getText().length()); // bytes msg
        dos.writeBytes(editMensagem.getText());     // mensagem
        byte [] buffer = os.toByteArray();
        DatagramPacket dp= new DatagramPacket(buffer, buffer.length,
             netip, porto);
        sock.send(dp);
    } catch (IOException e) {
        Log("Error sending packet: "+e+"\n");
    }
}
```

Observe-se que a estrutura da mensagem enviada é constituída por cinco campos:



Passo 14: Adicionar um método para tratar o premir do botão "**Enviar**" através de um duplo *click* na janela de edição de *forms*. Posteriormente, editar o texto gerado:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt){
    send_packet();
}
```

Passo 15: Adicionar método para tratar evento `KeyTyped` da tabela (objeto `jTable1`), através de duplo *click* através da janela de edição de *forms*. Sempre que se prime o "r", copia-se o endereço IP e porto da linha seleccionada para as caixas de texto.

```
private void jTable1KeyTyped(java.awt.event.KeyEvent evt) {
  if (evt.getKeyChar() == 'r') { // If pressed key 'r'
    int n= jTable1.getSelectedRow();
    if (n>-1) { // Replace remote values by selected row
      editRemIP.setText((String) jTable1.getValueAt(n,1));
      editRemPorto.setText((String) jTable1.getValueAt(n,2));
    }
  } // Else ignore
}
```

Passo 16: Definir a classe `Daemon` que realiza uma tarefa que espera continuamente por novos pacotes de dados. Quando recebe um novo pacote invoca o método `receive_packet`, declarado no passo 17:

```
public class Daemon extends Thread {

  volatile boolean keepRunning= true;
  demo_netbeans root;                    // Objeto raiz
  DatagramSocket ds;                     // socket

  public Daemon(demo_netbeans root, DatagramSocket ds) {
    this.root= root;
    this.ds= ds;
  }

  public void run() {
    byte [] buf= new byte[MAX_PLENGTH];
    DatagramPacket dp= new DatagramPacket(buf, buf.length);
    try {
      while (keepRunning) {
        try {
          ds.receive(dp);
          ByteArrayInputStream BAis=
              new ByteArrayInputStream(buf, 0, dp.getLength());
          DataInputStream dis= new DataInputStream(BAis);
          Log("Received packet ("+dp.getLength()+") from " +
              dp.getAddress().getHostAddress() + ":" +dp.getPort()+"\n");
          root.receive_packet(dp, dis);       // process packet
          serv.yield();
        } catch (SocketException se) {
          if (keepRunning)
            Log("recv UDP SocketException : "+se+"\n");
        }
      }
    } catch(IOException e) {
      if (keepRunning)
        Log("IO exception receiving data from socket : "+e);
    }
  }

  public void stopRunning() {
    keepRunning= false;
  }
}
```

Passo 17: Adicionar o método `receive_packet` à classe `demo_netbeans` para processar os campos do pacote recebido:

```
public synchronized void receive_packet(DatagramPacket dp,
     DataInputStream dis) {
  try {
    int len_nome= dis.readByte();
    byte [] sbuf1= new byte [len_nome];
    int n= dis.read(sbuf1,0,len_nome);
    if (n != len_nome) {
      Log("Sender name too short\n");
      return;
    }
    String nome= new String(sbuf1,0,n);
    Date snd_date= new Date(dis.readLong());
    int len_msg= dis.readShort();
    byte [] sbuf2= new byte [len_msg];
    n= dis.read(sbuf2,0,len_msg);
    if (n != len_msg) {
      Log("Message too short\n");
      return;
    }
    String msg= new String(sbuf2,0,n);
    if (dis.available()>0) {
      Log("Packet too long\n");
      return;
    }
    // Write message contents
    java.text.SimpleDateFormat formatter=
          new java.text.SimpleDateFormat("hh:mm:ss");
    Log("From "+nome+" at "+formatter.format(snd_date)+
          " sent '"+msg+"'\n");
    // Add log to jTable1
    if (last_line < jTable1.getRowCount()) {
      // Write packet data in last_line
      jTable1.setValueAt(nome,last_line,0);
      jTable1.setValueAt(dp.getAddress().getHostAddress(),last_line,1);
      jTable1.setValueAt(""+dp.getPort(),last_line,2);
      jTable1.setValueAt(formatter.format(snd_date),last_line,3);
      last_line++;    // Increment last_line
    }
  } catch(IOException e) {
    Log("Packet too short: "+e+"\n");
  }
}
```

Passo 18: Adicionar um método para tratar o premir do botão com estado (`jToggleButton1`) "**Activo**" através de um duplo *click* na janela de edição de *forms*. Quando se passa ao estado ligado pretende-se ligar a aplicação, criando o socket e lançando a tarefa de recepção de pacotes. Caso contrário, deve terminar todas as tarefas que estejam ativas nesse instante. Se durante o arranque houver alguma falha, o programa deve ficar no estado desligado.

```
private void jToggleButton1ActionPerformed(java.awt.event.ActionEvent
    evt) {
  if (jToggleButton1.isSelected()) {    // Start Datagram socket
    int porto;
    try {
      porto= Integer.parseInt(editLocalPorto.getText());
    } catch(NumberFormatException e) {
      Log("Invalid local port number: "+e+"\n");
      jToggleButton1.setSelected(false);
      return;
    }
    try {
      sock= new DatagramSocket(porto);
      editLocalPorto.setText(""+sock.getLocalPort());
      editRemPorto.setText(""+sock.getLocalPort());
      // create and start the receiver thread
      (serv= new Daemon(this, sock)).start();
    } catch (SocketException e) {
      Log("Socket creation failure: " + e + "\n");
      jToggleButton1.setSelected(false);
    }
  } else {      // Stop Datagram socket
    if (timer != null) {
      timer.stop();
      timer= null;
    }
    if (serv != null) {
      serv.stopRunning();
      serv= null; // Thread will be garbadge collected after it stops
    }
    if (sock != null) {
      sock.close();
      sock= null; // Forces garbadge collecting
    }
  }
}
```

Passo 19: Adicionar um método para tratar o premir do botão "**5x**" através de um duplo *click* na janela de edição de *forms*. Este método lança um temporizador que envia 5 pacotes nos cinco segundo seguintes (1 por segundo):

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
  cnt+= 5;
  // Set timer to fire 5 times
  (timer= new javax.swing.Timer(1000 /* 1 sec */, new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      // Timer action
      if (cnt>0) {
        send_packet();
        cnt--;
      } else
        timer.stop();
      }
    })
  ).start();  // Defines, Creates and Starts the timer
}
```

Passo 20: Compilar e correr a aplicação

4.4 Exercícios

1) Modifique a aplicação de maneira a deixar de estar ativa automaticamente, caso não receba nenhuma mensagem durante mais do que dois minutos.

2) Modifique a aplicação de maneira a limpar automaticamente as janelas `jTextArea1` e `jTable1`, sempre que for acrescentada uma linha depois da última linha.

3) Modifique a aplicação de maneira a memorizar a última mensagem recebida de cada nome de origem num *HashMap* e a escrever as últimas mensagens recebidas sempre que se prime a tecla "L".

5. BIBLIOGRAFIA ADICIONAL

Este documento resume uma pequena parte das especificações da linguagem Java e do ambiente de desenvolvimento NetBeans, necessárias para a realização do trabalho prático. No contexto da disciplina de Sistemas de Telecomunicações foi produzido um conjunto de documentos introdutórios à linguagem Java no Trabalho 0, que podem ser usados pelos alunos que estão a ter o primeiro contato com a linguagem Java.

Caso necessite de mais informação avançada sobre a linguagem do que a fornecida por este documento recomenda-se a consulta de:

"Thinking in Java First Edition", de Bruce Eckel, 2000, Prentice Hall. Disponível na web (http://www.mindview.net/Books).

"Thinking in Java Second Edition", de Bruce Eckel, 2001, Prentice Hall. Disponível na web (http://www.mindview.net/Books).

"Thinking in Java Third Edition", de Bruce Eckel, 2002. Disponível na web (http://www.mindview.net/Books).

"Thinking in Java Forth Edition", de Bruce Eckel, Prentice Hall, 2006, ISBN: 0131872486.

Documentação sobre o JDK e NetBeans disponível na web (http://java.sun.com/) e no laboratório (http).

"Java Cookbook, Second Edition", de Ian F. Darwin, O'Reilly & Associates, Inc., 2004, ISBN: 0596007019.

"Java Network Programming, Third Edition", de Elliotte R. Harold, O'Reilly & Associates, Inc., 2004, ISBN: 0596007213.

"Java in a Nutshell, 5th Edition", de David Flanagen, O'Reilly & Associates, Inc., 2005, ISBN: 0596007736.