

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY
DEPARTMENT OF ELECTRICAL
AND COMPUTER ENGINEERING

Sistemas de Telecomunicações

2022/2023

Trabalho 2:

Demonstração do ambiente Java
Aplicação com *sockets datagram*
Aplicação chat UDP

Aula 3

Licenciatura em Engenharia Eletrotécnica e de Computadores

<http://tele1.deec.fct.unl.pt/st>

Luís Bernardo
Paulo da Fonseca Pinto

Índice

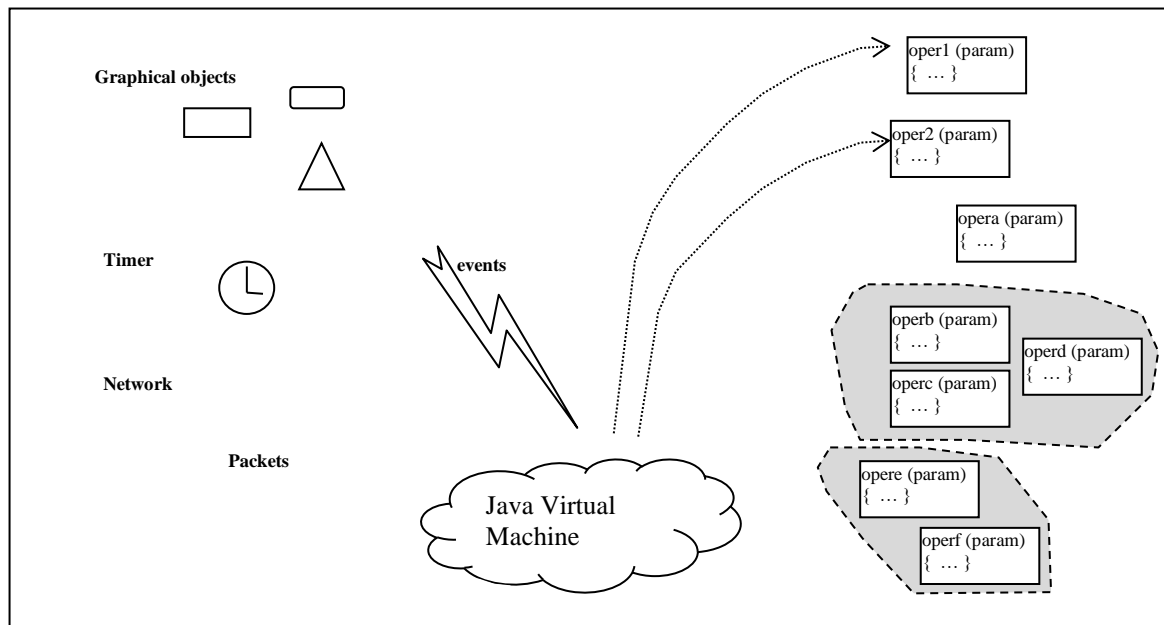
1.	Objetivo	1
2.	Estrutura do Programa (revisitada)	1
3.	Aplicação – Conversa em Rede com UDP	2
3.1.	Chat_UDP básico	2
3.1.1.	Objetos gráficos	2
3.1.2.	Criação de uma <i>thread</i> – classe <code>Daemon_udp</code>	4
3.1.3.	Manuseamento de um datagrama recebido	5
3.1.4.	Classe <code>Chat_udp</code> – inicialização	6
3.1.5.	Classe <code>Chat_udp</code> – tratamento de um pacote	8
3.1.6.	Classe <code>Chat_udp</code> – envio de pacotes	9
3.1.7.	Métodos de escrita nas caixas de texto	11
3.2.	Correr o código	12
3.3.	Chat_UDP avançado – Exercícios	12
3.3.1.	Envio de mensagens usando um temporizador	12
3.3.2.	Memorização da última mensagem dos utilizadores	13
3.3.3.	Envio de pacotes para todos os computadores na rede	14

1. OBJETIVO

Familiarização com a linguagem de programação Java e com o desenvolvimento de aplicações que comunicam usando *sockets* datagrama no ambiente de desenvolvimento NetBeans. O trabalho consiste na introdução parcial do código seguindo as instruções do enunciado, aprendendo a utilizar o ambiente e um conjunto de classes da biblioteca da linguagem Java. É fornecido um projeto com o início do trabalho, que é completado num conjunto de exercícios.

2. ESTRUTURA DO PROGRAMA (REVISITADA)

No trabalho anterior viu-se como se estrutura um programa Java. A figura dessa aula está reproduzida em baixo novamente, com o acrescento de uma nuvem para representar a máquina virtual do Java.



O funcionamento normal de programas como os usados em Sistemas de Telecomunicações é a máquina virtual do Java ficar bloqueada à espera que aconteça um evento e depois chamar o método que está ligado a esse evento. Tudo isto funciona muito bem se o método fizer o que tem a fazer rapidamente e devolver o controlo à máquina virtual.

Agora imagine que o método chama uma função que bloqueia por algum motivo, bloqueando a máquina virtual. Por exemplo, pense no equivalente ao “nosso” *scanf* do C, que fica bloqueado à espera de que o utilizador escreva qualquer coisa. Toda a estrutura da figura acima deixa de funcionar, pois um método ficou com o controlo e a máquina virtual fica impedida de receber mais eventos e chamar outros métodos.

Como se deve resolver este problema?

Um modo simples é proibir que qualquer método chame funções bloqueantes.

Como às vezes pode ser impossível fazer isso, outro modo é **paralelizar** o programa. Isto é, é como se o nosso programa, em vez de correr apenas num *contexto* (ou uma atividade), corresse dois (ou mais) *contextos/atividades*. Isto é, está a correr código em paralelo, em simultâneo. Se assim for, uma das atividades pode ficar bloqueada numa função (no tal *scanf*), mas a atividade

principal nunca ficaria bloqueada e a máquina virtual do Java pode continuar a chamar métodos a partir de eventos. Se precisássemos de bloquear esta segunda atividade por algum motivo, a solução seria arranjar ainda mais uma atividade que ficaria bloqueada, mas nunca a atividade principal. A estas atividades, qualquer uma delas, vamos chamar *threads*.

No caso deste trabalho, um método tem de ficar à espera no *socket* datagrama pela chegada de um pacote vindo da rede. Enquanto o pacote não vier, ele fica bloqueado. Assim, vamos precisar de uma *thread* só para ele. A outra *thread* é a principal e vai tratando dos eventos dos botões, das caixas, etc., nunca ficando bloqueada.

3. APLICAÇÃO – CONVERSA EM REDE COM UDP






Esta secção ilustra o desenvolvimento de uma aplicação utilizando *sockets* datagrama. A aplicação suporta a troca de mensagens em rede, onde cada participante tem duas janelas: uma janela (*Remote*) onde recebe mensagens de outros elementos; e outra janela (*Local*) onde escreve as suas mensagens. O utilizador pode seleccionar o endereço IP e o número de porto da máquina para onde envia as mensagens, escrever mensagens, e desligar a aplicação.

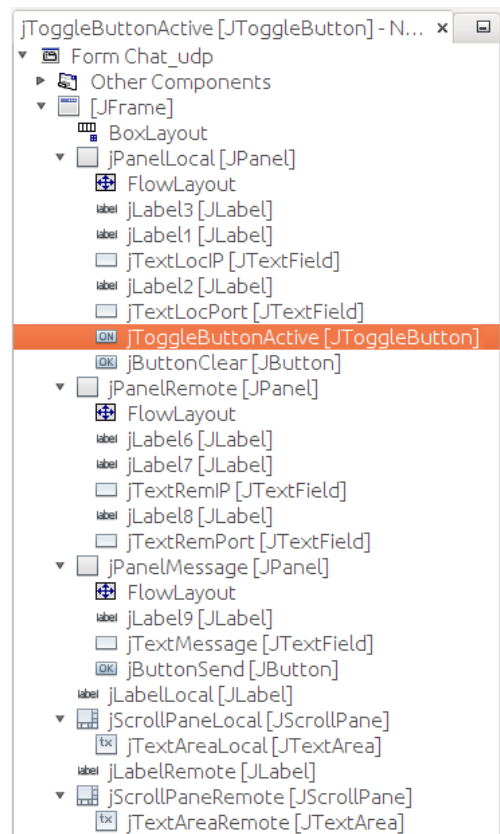
3.1.CHAT_UDP BÁSICO

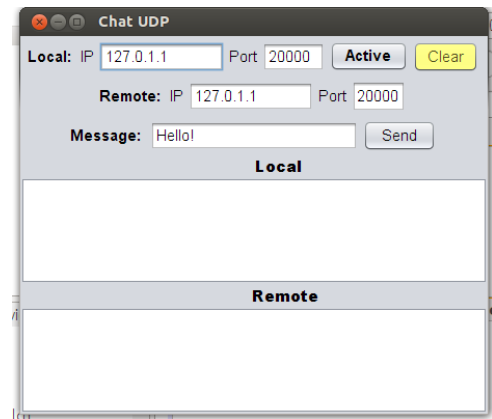
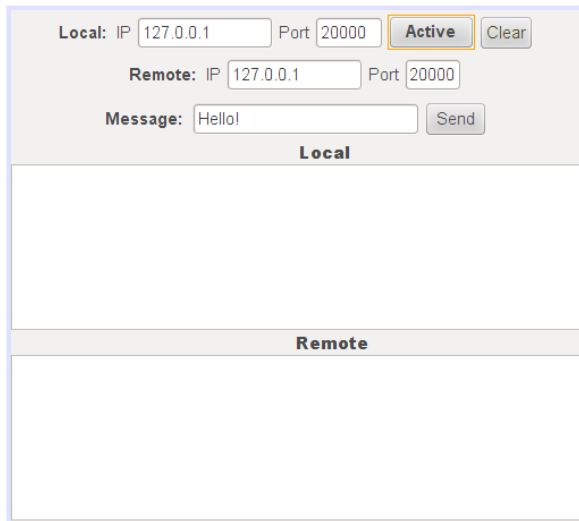
A primeira versão suporta apenas a troca simples de mensagens entre utilizadores, identificados de forma única na rede pelo conjunto {*endereço IP : número de porto*}. É fornecido aos alunos o projeto NetBeans com esta primeira parte completa.

3.1.1. Objetos gráficos

Tal como no exemplo da aula anterior, a primeira fase do desenvolvimento passa pela abertura do projeto *Chat_udp* distribuído com o enunciado, com o desenho da interface gráfica. Neste projeto definiu-se a janela representada na página seguinte, onde foram utilizados os seguintes componentes (com os nomes e a estrutura mostrada na figura ao lado):

- 2 objetos **Button**  {botões ‘Clear’ e ‘Send’}
- 1 objeto **ToggleButton** {botão ‘Active’}
- 5 objetos **Text Field**  {mensagem, IP e portos locais e remotos}
- 2 objetos **Text Area**  {texto local e remoto}
- 2 objetos **Scroll Pane**  {Barras de deslocamento para *TextArea*}
- 3 objetos **Panel**  {três linhas de grupos de botões}





De forma a ter o aspeto gráfico representado à direita, foram modificadas as seguintes propriedades dos objetos:

- objeto 'JFrame'
 - ✓ title= “*Chat UDP*”;
 - ✓ Layout=BoxLayout, com Axis= “*Y Axis*”.
- objeto 'JPanelLocal'
 - ✓ Layout=FlowLayout;
 - ✓ preferredSize= [450,38] e maximumSize= [450,40], limitando o crescimento vertical.
- objeto 'JTextLocIP'
 - ✓ preferredSize= [120,28], fixando a largura da caixa após “Local: IP”.
- objeto 'JTextLocPort'
 - ✓ preferredSize= [60,28], fixando a largura da caixa após “Local: Port”.
- objeto 'JButtonClear'
 - ✓ background= [220,220,100], modifica a cor para amarelo.
- objeto 'JPanelRemote'
 - ✓ Layout=FlowLayout;
 - ✓ preferredSize= [450,38] e maximumSize= [450,40], limitando o crescimento vertical.
- objeto 'JTextRemIP'
 - ✓ preferredSize= [120,28], fixando a largura da caixa após “Remote: IP”.
- objeto 'JTextRemPort'
 - ✓ preferredSize= [50,28], fixando a largura da caixa após “Remote: Port”.
- objeto 'JPanelMessage'
 - ✓ Layout=FlowLayout;
 - ✓ preferredSize= [450,38] e maximumSize= [450,40], limitando o crescimento vertical.
- objeto 'JTextMessage'
 - ✓ preferredSize= [200,28], fixando a largura da caixa após “Message:”.
- objeto 'JLabelLocal'
 - ✓ preferredSize= [50,22] e maximumSize= [50,22], limitando o crescimento vertical;
- objeto 'JScrollPaneLocal'
 - ✓ preferredSize= [222,87], ficando com o máximo ilimitado para poder crescer.
- objeto 'JTextAreaLocal'
 - ✓ preferredSize= [220,85], ficando com o máximo ilimitado para poder crescer.
- objeto 'JLabelRemote'
 - ✓ preferredSize= [64,22] e maximumSize= [64,22], limitando o crescimento vertical;
- objeto 'JScrollPaneRemote'

- ✓ `preferredSize= [222,87]`, ficando com o máximo ilimitado para poder crescer.
- objeto `'jTextAreaRemote'`
- ✓ `preferredSize= [220,85]`, ficando com o máximo ilimitado para poder crescer.

O `Layout` não é bem uma propriedade e é escolhido com o botão direito do rato quando se seleciona o objeto (por exemplo, `JFrame`) na janela do “*Navigator*”. A conjugação do `Layout` com os valores dados para as dimensões mostrados acima faz com que caso se aumente a janela, apenas aumentam as caixas com texto *local* e *remote*.

Para além das modificações anteriores, a fonte das etiquetas a negrito (*labels*) foi modificada para “*Arial 15 Bold*” e as restantes para “*Arial 15 Plain*” de maneira a adotar uma fonte que existe em Windows, MacOS e Linux, tornando o código mais portátil entre plataformas.

3.1.2. Criação de uma *thread* – classe `Daemon_udp`

Esta secção descreve a estrutura geral do programa relativamente a paralelismo

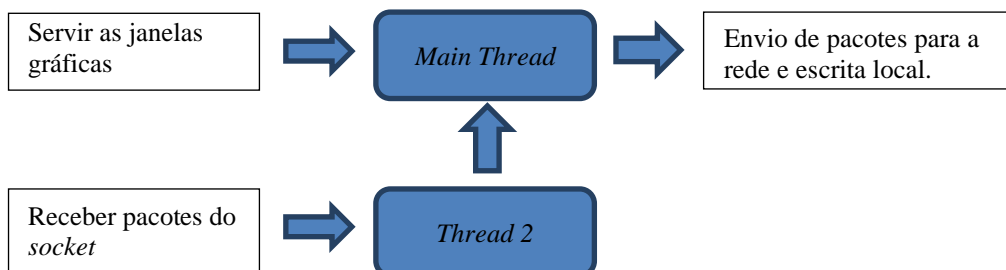
O programa vai ter duas *threads*: A classe `Chat_udp` e a `Daemon_udp`.

A `Chat_udp` tem a interface gráfica, e é responsável pela realização de toda a lógica do programa; a `Daemon_udp` vai executar a receção de pacotes.

O objeto de Java `DatagramSocket`, que representa e gere um *socket* datagrama, tem o “problema” de ter uma operação bloqueante para se ler o *socket*, isto é, para se ler um pacote da rede. O nome desse método é `nome_do_objeto.receive (dp)`. Assim, uma primeira hipótese seria ter-se uma atividade que estaria dedicada a ler o *socket* de entrada e depois escrevia a mensagem na caixa remota do ecrã; e outra atividade que servia as janelas gráficas e que enviasse pacotes para a rede, como está mostrado em baixo:



Na realidade não se vai usar o esquema da figura em cima. O problema é que tanto a *thread* principal como a *thread 2* manuseariam os mesmos objetos gráficos, e teríamos duas entidades paralelas a “mexer” nas mesmas coisas – potencial desastre!!! Então o melhor é fazer com que a *thread 2* seja muito simples, apenas leia o *socket* e chame um método da *thread* principal. Um esquema que traduz melhor a relação entre as *threads* é o seguinte:



Para criar a *thread 2* criou-se um objeto de uma classe nova. Para criar uma classe nova selecionou-se o *package* “*udpdemoproject*” e com o botão direito do rato escolheu-se “*New*” e “*Java Class*”. Escolheu-se um nome para ela – `Daemon_udp`. Para lhe dar o comportamento de *thread*, teve de se escrever manualmente que esta classe estende a classe `Thread`. Basicamente tem de se escrever o código mostrado em baixo. Este código deve ser utilizado pelos alunos como

uma “receita” sempre que usarem *sockets* datagrama nestes próximos tempos até se sentirem mais confiantes em Java.

O código contém:

- o construtor (onde os valores das variáveis da classe `root` e `ds` são inicializados). Estas variáveis permitem à *thread* 2 conhecer qual o *socket* que tem de ler, e saber a referência da *thread* principal para depois lhe chamar o método para lhe dar o pacote,
- o método `run` que será corrido quando a *thread2* for lançada (através da operação `start`). Este método tem o código que é corrido nesta *thread2*.
- o método `stopRunning`, que permite parar a *thread2* (através de uma variável Booleana `keepRunning`).

```
public class Daemon_udp extends Thread {    // inherits from Thread class
    volatile boolean keepRunning = true;
    Chat_udp root;                          // Main window object
    DatagramSocket ds;                      // datagram socket

    public Daemon_udp(Chat_udp _root, DatagramSocket _ds) { // Constructor
        this.root = _root;
        this.ds = _ds;
    }

    public void run() {                      // Function run by the thread
        byte[] buf = new byte[Chat_udp.MAX_PLLENGTH]; // buffer with maximum message size
        DatagramPacket dp = new DatagramPacket(buf, buf.length);
        try {
            while (keepRunning) {
                try {
                    ds.receive(dp); // Wait for packets
                    ByteArrayInputStream BAis = new ByteArrayInputStream(buf, 0, dp.getLength());
                    DataInputStream dis = new DataInputStream(BAis);
                    root.receive_packet(dp, dis); // process packet in Chat_udp object
                } catch (SocketException se) {
                    if (keepRunning) {
                        root.Log_rem("recv UDP SocketException : " + se + "\n");
                    }
                }
            }
        } catch (IOException e) {
            if (keepRunning) {
                root.Log_rem("IO exception receiving data from socket : " + e);
            }
        }
    }

    public void stopRunning() { // Stops loop by turning off keepRunning
        keepRunning = false;
    }
}
```

O *socket* vai ser criado na *thread* principal e é passado para esta classe no construtor (argumento `ds`) sendo guardado na variável com o mesmo nome (`this` identifica o objeto local), a referência para o objeto/*thread* principal é passada da mesma forma (`root`).

O método `run` fica em ciclo à espera de pacotes. Quando um pacote chega é invocado o método `receive_packet` da classe `Chat_upd` (objeto/*thread* principal), passando-lhe o objeto de leitura de campos do pacote (`dis`). O método `run` também lida com as exceções resultantes de erros na comunicação.

3.1.3. Manuseamento de um datagrama recebido

Um datagrama tem a estrutura mostrada na figura em baixo. Consiste numa primeira parte, o cabeçalho, que contém informação como os endereços da máquina de origem e da máquina de destino, os portos e mais alguma informação. Depois vem a parte de dados que contém a

informação que as aplicações pretendem trocar. Ora, as aplicações pretendem trocar inteiros, caracteres, reais, etc., e isso tem de se colocar sequencialmente uns atrás de outros nessa parte de dados. No caso desta secção, tem de retirar esses inteiros, etc. do datagrama recebido.

Como fazer isso?

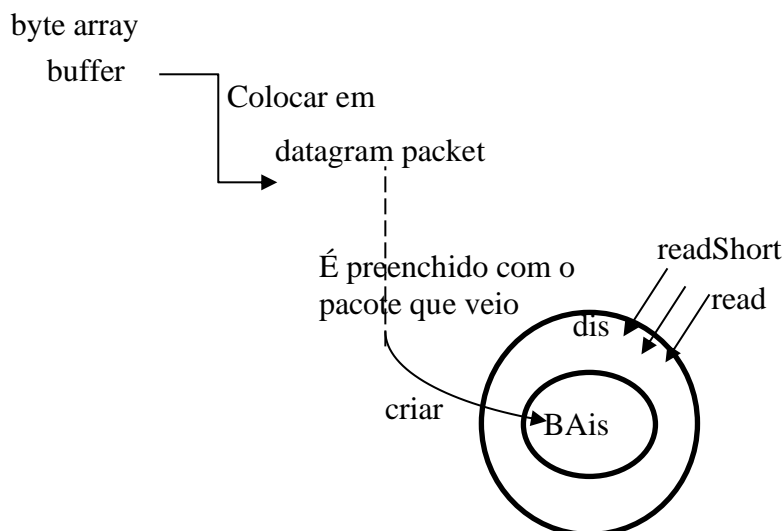
Cabeçalho	Dados
-----------	-------

A figura abaixo mostra os objetos que são usados no processo completo de tratamento de receção de datagramas (tente descobrir o código respeitante a esta descrição). Primeiro é criado um objeto *DatagramPacket* que é dado no método *receive* do objeto *DatagramSocket*. Vai ser copiado para o objeto *DatagramPacket* o pacote que veio da rede. Para se criar este pacote é necessário ter um array de bytes de um certo comprimento que vai ser a parte de dados. Os outros campos do objeto *DatagramPacket* – endereços, portos, etc. – são definidos na classe e são tratados pelo próprio objeto.

Quando chega um pacote, o método *receive*, que recebe como parâmetro este objeto *DatagramPacket*, preenche-o com o pacote que veio da rede (parte a tracejado na figura).

A informação sobre os endereços é acedida através de métodos do objeto *DatagramPacket*. O array de bytes, os dados do pacote, vai conter inteiros (com 2, 4 ou 8 bytes), caracteres, etc. Como é que poderemos obter esses elementos de um modo simples? Como foi feito, foi criar um objeto *ByteArrayInputStream* dando o array de bytes como suporte de memória (objeto *BAis*). Assim, é como se tivéssemos uns óculos *ByteArrayInputStream* para ver essa memória. Como ter uma interface de bytes ainda é complicado, envolveu-se o objeto anterior com um *DataInputStream* (objeto *dis*), ou dito de outro modo criou-se um objeto *DataInputStream* sobre o objeto *ByteArrayInputStream*. Agora temos uns óculos *DataInputStream* sobre essa memória. Este objeto já tem operações como *readShort* e outras do género.

É este último objeto que é dado ao objeto da classe *Chat_udp*.



Tente identificar estas relações no código. Perceber isto é útil para quando tiver de programar autonomamente sockets UDP.

3.1.4. Classe *Chat_udp* – inicialização

A classe *Chat_udp* tem a interface gráfica, é responsável pela realização de toda a lógica do programa e pela definição das configurações. Uma delas é *MAX_LENGTH* que define um valor constante (porque se usou *final*) a nível da classe (porque se usou *static*) com o tamanho máximo dos pacotes trocados. Contém também algum código em comentário que será útil para a segunda parte dos exercícios.


```
public static final int MAX_PLNGTH = 8096; // Constant - Maximum packet length
```

As variáveis principais `sock_udp` e `listen_udp` guardam o objeto do tipo `DatagramSocket` (o socket por onde se recebem e enviam pacotes e que vai ser dado à *thread2*) e uma referência para a classe `Daemon_udp` (a *thread 2*), e que vai apontar para o objeto quando ele for criado. A variável auxiliar `formatter` serve para formatar a escrita de datas na forma “hora:minutos” e é inicializada logo na declaração pois não depende de nenhum valor externo.

```
// Variables declaration
private DatagramSocket sock_udp; // datagram socket
private Daemon_udp listen_udp; // thread for message reception
private java.text.SimpleDateFormat formatter = // Formatter for dates
    new java.text.SimpleDateFormat("hh:mm:ss");
```

O construtor da classe `Chat_udp` (método `Chat_udp`) cria os vários objetos gráficos e preenche algumas caixas de texto com o valor do endereço IP da máquina. Para isso, primeiro vai saber o endereço IP na forma `InetAddress` e depois usando o método `getHostAddress` escreve a *string* devolvida na caixa de texto. Escreve também o valor de 20000 na caixa de texto do porto.

```
public Chat_udp() {
    initComponents(); // defined by NetBeans, creates the graphical window
    sock_udp = null; // Set null value - meaning "not initialized"
    listen_udp = null; // Set null value - meaning "not initialized"
    try {
        // Get local IP and set port to 0
        InetAddress addr = InetAddress.getLocalHost(); // Get the local IP address
        jTextLocIP.setText(addr.getHostAddress()); // Set the IP text fields to
        jTextRemIP.setText(addr.getHostAddress()); // the local address
    } catch (UnknownHostException e) {
        System.err.println("Unable to determine local IP address: " + e);
        System.exit(-1); // Closes the application
    }
    jTextLocPort.setText("20000");
}
```

A aplicação já está a correr, mas ainda nada aconteceu para além da escrita de texto nas caixas de texto feita pelo construtor. Quando se carregar no botão “Active”, o método que o trata lê o número de porto local escrito na caixa, cria um *socket* nesse porto (atenção que se já houver um *socket* criado nesse porto a criação aborta), cria o objeto `Daemon_udp`, dando-lhe a sua referência e a do *socket*, e lança-o com a operação `start`.

Quando se desativa o botão “Active”, a *thread2* é parada pela chamada ao método `stopRunning`, o objeto é destruído, e o *socket* é fechado.

```
private void jToggleButtonActiveActionPerformed(java.awt.event.ActionEvent evt)
if (jToggleButtonActive.isSelected()) { // The button is ON
    int port;
    try { // Read the port number in Local Port text field
        port = Integer.parseInt(jTextLocPort.getText());
    } catch (NumberFormatException e) {
        Log_loc("Invalid local port number: " + e + "\n");
        jToggleButtonActive.setSelected(false); // Set the button off
        return;
    }
    try {
        sock_udp = new DatagramSocket(port); // Create UDP socket
        jTextLocPort.setText("" + sock_udp.getLocalPort());
        jTextRemPort.setText("" + sock_udp.getLocalPort());
        listen_udp = new Daemon_udp(this, sock_udp); // Create the receiver thread
        listen_udp.start(); // Start the receiver thread
        Log_loc("Chat_udp active\n");
    }
```

```

    } catch (SocketException e) {
        Log_loc("Socket creation failure: " + e + "\n");
        jToggleButtonActive.setSelected(false); // Set the button off
    }
} else { // The button is OFF
    if (listen_udp != null) { // If thread is running
        listen_udp.stopRunning(); // Stop the thread
        listen_udp = null; // Thread will be garbage collected after it stops
    }
    if (sock_udp != null) { // If socket is active
        sock_udp.close(); // Close the socket
        sock_udp = null; // Forces garbage collecting
    }
    Log_loc("Chat_udp stopped\n");
}
}
}

```

3.1.5. Classe Chat_udp – tratamento de um pacote

Os pacotes são lidos pelo objeto da classe `Daemon_udp`, que depois invoca o método `receive_packet` do objeto da classe `Chat_udp` para o tratar. O formato de pacote, a parte dos dados, que foi definido para esta aplicação é:

Comprimento (short)	Mensagem (byte[])
---------------------	-------------------

Repare que o método `receive_packet` tem também a propriedade de `synchronized`. Isto serve para que ele não seja corrido em paralelo enquanto estiver a ser corrido. Isto é, tem de terminar a sua execução antes de começar uma nova (pode acontecer se vier um segundo pacote muito rapidamente). Basicamente, ele escreve os dados na caixa de texto “*Remote*” chamando o método `Log_rem` (ver mais abaixo). Repare que o primeiro campo dos dados (comprimento da mensagem) é lido para um inteiro e depois a mensagem é lida para um buffer de bytes. O buffer de bytes é transformado numa `String` para depois ser escrito:

```

public synchronized void receive_packet(DatagramPacket dp, DataInputStream dis) {
    try {
        Date date = new Date(); // Get reception hour
        String from = dp.getAddress().getHostAddress() // IP address of the sending host
            + ":" + dp.getPort(); // + port of sending host = User ID

        // Read the packet fields using 'dis'
        int len_msg = dis.readShort(); // Read message length
        if (len_msg > MAX_PLENGTH) {
            Log_rem(formatter.format(date) + " - received message too long (" + len_msg +
                ") from " + from + "\n");
            return; // Leaves the function
        }
        byte[] sbuf2 = new byte[len_msg]; // Create an array to store the message
        int n = dis.read(sbuf2, 0, len_msg); // returns number of byte read
        if (n != len_msg) {
            Log_rem(formatter.format(date) + " - received message too short from " +
                from + "\n");
            return;
        }
        String msg = new String(sbuf2, 0, n); // Creates a String from the buffer
        if (dis.available() > 0) { // More bytes after the message in the buffer
            Log_rem("Packet too long\n");
            return;
        }
        // Write message contents
        Log_rem(formatter.format(date) + " - received from " + from + " - '" + msg + "'\n");
    } catch (IOException e) {
        Log_rem("Packet too short: " + e + "\n");
    }
}
}

```

3.1.6. Classe Chat_udp – envio de pacotes

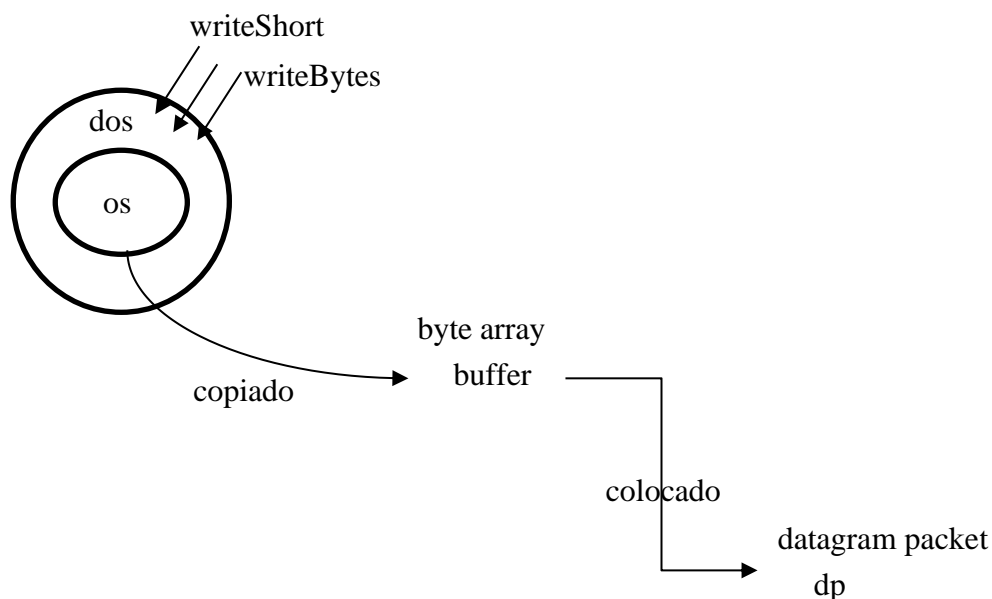
O envio de pacotes é realizado pelo método de serviço ao botão “Send”:

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

Este método invoca simplesmente o método privado e sincronizado `send_packet`, que lê o endereço IP e o porto das caixas de texto respectivas e lê o texto da área de texto.

Agora tem-se o problema inverso: como colocar inteiros, caracteres, etc., sequencialmente uns atrás dos outros para fazer a parte de dados do pacote?

O envio do datagrama segue uma lógica parecida com a que foi montada para a leitura e está mostrado na figura abaixo. Veja a figura e tente perceber que objetos são e porque é que são usados.



Resposta: **LEIA DEPOIS DE PENSAR.**

Basicamente criou-se um objeto da classe `ByteArrayOutputStream (os)` e envolveu-se com um objeto da classe `DataOutputStream (dos)` para se ter operações de se escrever inteiros, reais, bytes, etc. Depois escreve-se o que se tem de escrever nesse objeto.

Seguidamente copia-se o que foi sendo construído para um array de bytes e coloca-se esse array de bytes num objeto datagrama que se acabou de criar. Cuidado que outra informação importante do objeto datagrama, como o endereço de destino, os portos, etc., ainda não estão lá colocada... Ela vai ser colocada usando métodos do objeto datagrama, *DatagramPacket*.

Confira o código:

```

public synchronized void send_packet() {
    if (sock_udp == null) {
        Log_loc("Socket isn't active!\n");
        return;
    }
    InetAddress netip;
    try { // Get IP address
        netip = InetAddress.getByNames(jTextRemIP.getText());
    } catch (UnknownHostException e) { // O endereço IP não é válido
        Log_loc("Invalid remote host address: " + e + "\n");
        return;
    }
    int port;
    try { // Get port
        port = Integer.parseInt(jTextRemPort.getText());
    } catch (NumberFormatException e) {
        Log_loc("Invalid remote port number: " + e + "\n");
        return;
    }
    String message= jTextMessage.getText();
    if (message.length() == 0) {
        Log_loc("Empty message: not sent\n");
        return;
    }
    // Create and send packet
    ByteArrayOutputStream os = new ByteArrayOutputStream(); // Prepares a message
    DataOutputStream dos = new DataOutputStream(os); // writting object
    try {
        dos.writeShort(message.length()); // Write the message's length to buffer
        dos.writeBytes(message); // Write the message contents to buffer
        byte[] buffer = os.toByteArray(); // Convert to byte array
        DatagramPacket dp = new DatagramPacket(buffer, buffer.length); // Create packet
        send_one_packet(dp, netip, port, message); // Send packet to netip:port and log event
    } catch (Exception e) { // Catches all exceptions
        Log_loc("Error sending packet: " + e + "\n");
    }
}
}

```

Finalmente, o método `send_one_packet` recebe como parâmetros: o datagrama (classe *DatagramPacket*), um endereço (classe *InetAddress*), o porto de destino (inteiro) e mais a mensagem que vai ser enviada em formato *String* apenas para poder ser escrita na caixa local usando o método `Log_loc`.

O método define o que falta no datagrama (o endereço e o porto) e envia-o para a rede através do *socket*. Note que o código tem dois `catch` para duas exceções diferentes.

```

private void send_one_packet(DatagramPacket dp, InetAddress netip /* destination IP */,
                             int port /* destination port */, String message) {
    try {
        dp.setAddress(netip); // Set destination ip
        dp.setPort(port); // Set destination port
        sock_udp.send(dp); // Send packet
        // Write message to JTextAreaLocal
        String to = netip.getHostAddress() + ":" + port; // 'name' of remote host
        String log = formatter.format(new Date()) + " - sent to " + to
            + " - '" + message + "'\n";
        Log_loc(log); // Write to Local text area
    } catch (IOException e) { // Communications exception
        Log_loc("Error sending packet: " + e + "\n");
    } catch (Exception e) { // Other exception (e.g. null pointer, etc.)
        Log_loc("Error sending packet: " + e + "\n");
    }
}
}

```

3.1.7. Métodos de escrita nas caixas de texto

Para facilitar a escrita nas caixas de texto “*Local*” e “*Remote*” definiram-se dois métodos: `Log_loc` e `Log_rem`. Os dois são `synchronized` para garantir que as operações de escrita não são interrompidas.

```
public synchronized void Log_loc(String s) {
    try {
        JTextAreaLocal.append(s);           // Write to the Local text area
        System.out.print("Local: " + s);    // Write to the terminal
    } catch (Exception e) {
        System.err.println("Error in Log_loc: " + e + "\n");
    }
}

public synchronized void Log_rem(String s) {
    try {
        JTextAreaRemote.append(s);          // Write to the Remote text area
        System.out.print("Remote: " + s);   // Write to the terminal
    } catch (Exception e) {
        System.err.println("Error in Log_rem: " + e + "\n");
    }
}
```

Estas caixas de texto são limpas no método que trata o evento associado ao botão “*Clear*”:

```
private void jButtonClearActionPerformed(java.awt.event.ActionEvent evt) {
    JTextAreaLocal.setText("");
    JTextAreaRemote.setText("");
}
```

3.2. CORRER O CÓDIGO

Se quiser ter o emissor e o recetor no mesmo computador vai ter de correr o código duas vezes no seu computador. Para correr o código desligado do ambiente de desenvolvimento NetBeans seleccione o projeto e com o botão direito do rato execute a opção “Run”. Um modo alternativo é expandir o botão “Run” (o triângulo verde) e escolher a segunda opção (que tem o nome do pacote seguido de “(run)”). Pode usar o endereço do computador ou, por exemplo, 127.0.0.1.

3.3. CHAT_UDP AVANÇADO – EXERCÍCIOS

Pretende-se que os alunos realizem um conjunto de três exercícios:

- A. Realizar o envio de cinco mensagens (uma por segundo) durante 5 segundos, controlado por um temporizador, ao premir uma tecla “Send 5”.
- B. Memorizar a última mensagem recebida de cada um dos outros utilizadores.
- C. Enviar uma mensagem para um porto em todas as máquinas de uma rede.

3.3.1. Envio de mensagens usando um temporizador

Para realizar o envio de mensagens usando um temporizador é necessário criar um objeto temporizador (`timer`), que vai disparar o envio das mensagens. Recorde que o objeto temporizador já foi pensado como tendo um método de *callback*. Assim, não precisamos de mais uma *thread* para ele. Para controlar o número de vezes que ainda falta enviar, é usada uma variável `counter` que funciona como contador – conta o número de vezes que o temporizador ainda deve disparar.

Exercício A1: acrescente as seguintes variáveis à declaração da classe `Chat_udp`:

```
private javax.swing.Timer timer;      // Timer object
public volatile int counter;          // Counter of messages left to send
```

Para usar um temporizador, é necessário criar a função `set_timer_function` que inicia o objeto `timer`, e que define a função que vai ser corrida cada vez que passar o tempo. Note que o temporizador fica criado mas não está ativo – só quando se invoca o método `start()`.

Exercício A2: acrescente o seguinte método à classe `Chat_udp`:

```
private void set_timer_function(int period /*ms*/) {
    java.awt.event.ActionListener action;      // Callback object
    action = (java.awt.event.ActionEvent evt) -> {
        // Define the timer callback function
        if (counter > 0) {                      // While there are more packets to send ...
            send_packet();                      // Send the packet
            counter--;                          // Decrement the counter
        } else { // Counter reached 0
            timer.stop();                      // Stop the timer
        }
    };
    timer = new javax.swing.Timer(period, action); // Create the timer's object
}
```

Para que o objeto `timer` fique inicializado, é necessário chamar este método. Isso pode ser feito quando se seleccionar o botão “Active” (no método `jToggleButtonActiveActionPerformed` da classe `Chat_udp`). Quando se desseleccionar o botão “Active” também é necessário desligar o temporizador. Os dois exercícios seguintes abordam isto.

Exercício A3: acrescente a invocação ao método `set_timer_function` passando como argumento 1 segundo (1000 ms). Deve ser colocado depois de já terem sido criados com sucesso o *socket* e a *thread*.

Exercício A4: acrescente o código que desliga o objeto `timer` quando o botão “Active” for desseleccionado.

Para terminar, falta apenas acrescentar um botão “Send 5”, que vai arrancar o temporizador.

Exercício A5: acrescente um botão “Send 5” ao lado do “Send” e crie o método de tratamento ao botão de maneira a:

- arrancar o timer (com `timer.start()`;) caso ele ainda não tivesse arrancado (pode saber isso com `timer.isRunning()`);
 - inicializar convenientemente o valor de `counter` de maneira a enviar 5 pacotes.
- Lembre-se que `timer` pode ser igual a `null` (porque a aplicação pode não estar ativa).

3.3.2. Memorização da última mensagem dos utilizadores

Pretende-se que a aplicação memorize a última mensagem que recebeu de todos os utilizadores com quem está a comunicar. Para controlar esta funcionalidade, propõe-se a utilização de um botão com estado designado por “Record” (com o nome `jToggleButtonRecord`). Quando o botão estiver seleccionado, todas as mensagens recebidas são guardadas; quando se desligar o botão escreve-se todas as últimas mensagens de todos os utilizadores, um a um.

Exercício B1: Acrescente um *Toggle Button* à interface gráfica, com o nome `jToggleButtonRecord` e com o texto “Record”. Crie o método de tratamento do botão (com duplo clique no botão).

Para realizar esta funcionalidade vai ser usada uma lista indexada por identificador de utilizador remoto – constituído pela concatenação do *endereço IP*+”.”+*número de porto*. (`HashMap<String,String>`).

Exercício B2: acrescente a seguinte declaração da lista `record` à classe `Chat_udp`:

```
private HashMap<String, String> record = new HashMap<String, String>();
```

Cada vez que se recebe um pacote, é necessário criar uma *string* com o identificador de utilizador. De seguida, deve-se guardar na lista a *string* do pacote recebido associado ao identificador.

Exercício B3: acrescente ao método `receive_packet` (apresentado na secção 3.1.5) a operação de registo na lista se o botão estiver seleccionado:

```
if (jToggleButtonRecord.isSelected()) { // If button is selected
    String str = formatter.format(date) + " - received from " + from + " - " + msg +
        "\n";
    record.put(from, str); // Put string into the list associated to key from
}
```

Pretende-se escrever todos os registos e origens guardados na lista, que mantêm a última mensagem recebida de cada utilizador. Vai ser usado um ciclo `for` para percorrer a lista de chaves. A partir da chave vai-se usar o método `get` para obter o último pacote associado a cada

utilizador. Por exemplo, pode-se saber o último pacote enviado por “127.0.0.1:20000” fazendo a invocação `String str= record.get("127.0.0.1:20000").`

Exercício B4: acrescente à classe `Chat_udp` o método `write_record`, que escreve no terminal e na janela de mensagens remotas o conteúdo da lista e limpa a lista.

```
public void write_record() {
    Log_rem("-----\n");
    for (String remote : record.keySet()) { // for cycle over keys
        Log_rem("Communication with " + remote+"\n");
        Log_rem(record.get(remote)); // Write the last message from remote
    }
    Log_rem("-----\n");
    record.clear(); // Clear the list
}
```

Para terminar, falta apenas programar a função que trata o evento de premir o botão “Record” de forma a chamar o método `write_record` quando se desliga o botão “Record”.

Exercício B5: programe o método de tratamento do botão “Record” que criou no exercício B1 de maneira a invocar o método `write_record` apenas quando se desliga o botão; repare que o método de tratamento vai ser corrido quando se liga e quando se desliga o botão.

3.3.3. Envio de pacotes para todos os computadores na rede

O último exercício, envio de pacotes para todos os computadores na rede, só deve ser realizado caso ainda falem mais de 20 minutos para o fim da aula.

Pretende-se enviar um pacote para todos os computadores que estiverem ligados à rede, em vez de enviar apenas para um endereço IP. O laboratório 3.4 tem 11 máquinas na rede 172.16.54.0, ocupando os endereços IP 172.16.54.101 – 172.16.54.111. Existem depois os computadores portáteis que, caso estejam ligados à rede WiFi ST existente no laboratório obtêm um endereço dinâmico na sub-rede 172.16.54.

Caso esteja noutra rede, poderá saber o endereço de rede a partir do endereço IP da máquina e da máscara de rede. Admitindo uma rede até 254 máquinas, pode obter o prefixo de rede (“172.16.54.” no exemplo anterior) procurando pelo último ‘.’ da *string* com o endereço.

Para controlar o envio, crie um botão com estado (*Toggle Button*) “All”, que sempre que esteja ligado envie o pacote para um intervalo de identificadores de endereços que defina, no porto selecionado. Note que ao se usar datagramas, podemos enviar pacotes para máquinas que não existem, o que não tem importância. Repare que todas as modificações têm apenas de ser feitas no método `send_packet`, que passa a chamar o método `send_one_packet` para cada máquina desse intervalo.

Exercício C:

- 1) Criar um *Toggle Button* “All”;
- 2) Modificar o método `send_packet` para que caso o botão “All” esteja selecionado, gerar todos os endereços IP e chamar a função `send_one_packet`.
Sugestão: pode criar uma *string* com o endereço IP, e posteriormente, convertê-la para o tipo `InetAddress`.