



**FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA**

Departamento de Engenharia Electrotécnica

Sistemas de Telecomunicações

2014/2015

Trabalho 0:

Demonstração do ambiente Java

Aprendizagem do desenvolvimento de aplicações

Aula 2 – Aplicação com sockets datagrama

***Mestrado integrado em Engenharia Eletrotécnica e de
Computadores***

Luís Bernardo

Paulo da Fonseca Pinto

Índice

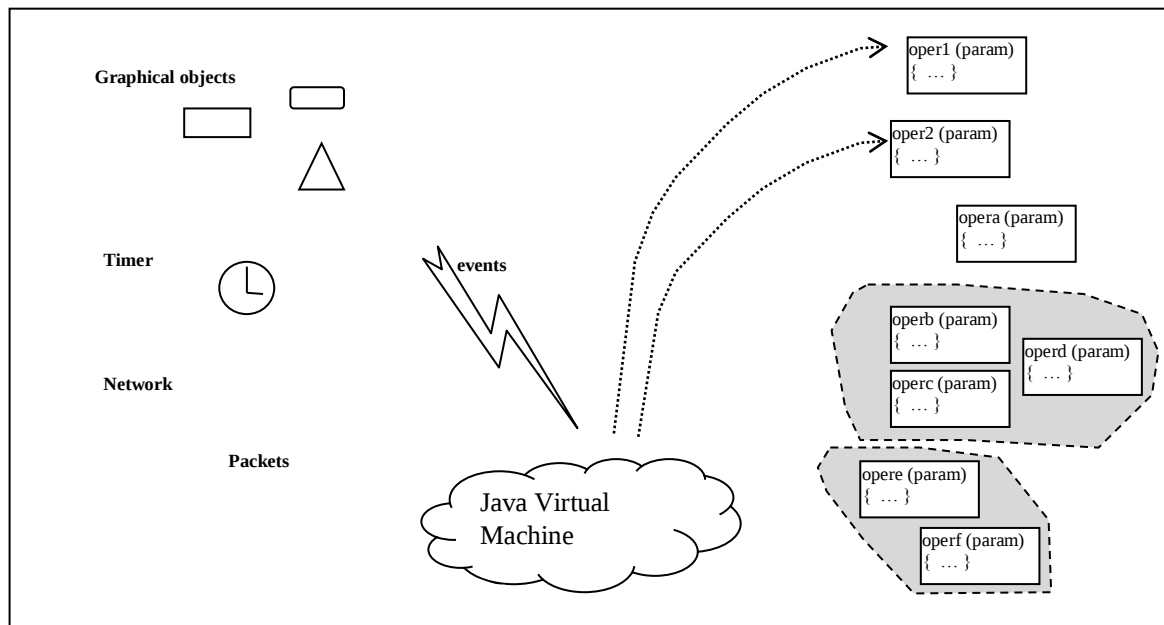
1.	Objetivo	1
2.	Estrutura do Programa (revisitada)	1
3.	Segunda Aplicação – Conversa em Rede com UDP	2
3.1.	Chat_UDP básico	2
3.1.1.	Objetos gráficos	2
3.1.2.	Criação de uma <i>thread</i> – classe <code>Daemon_udp</code>	4
3.1.3.	Classe <code>Chat_udp</code> – inicialização	6
3.1.4.	Classe <code>Chat_udp</code> – tratamento de um pacote	7
3.1.5.	Classe <code>Chat_udp</code> – envio de pacotes	8
3.2.	Chat_UDP avançado – Exercícios	9
3.2.1.	Envio de mensagens usando um temporizador	9
3.2.2.	Memorização da última mensagem dos utilizadores	10
3.2.3.	Envio de pacotes para todos os computadores na rede	11

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.



Ora, o funcionamento normal é a máquina virtual do Java ficar bloqueada à espera que aconteça um evento e depois chamar o método que lhe foi dito para chamar quando esse evento acontecesse. 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 o bloqueia. Por exemplo, o equivalente ao “nosso” *scanf* do C, ficando bloqueado à espera que o utilizador escrevesse 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 de fazer isso, vamos paralelizar o programa. É como se o nosso programa, em vez de correr apenas num *contexto* (ou uma atividade), corre em dois (ou mais) *contextos/atividades*. Está a correr código em paralelo. Agora, uma das atividades pode ficar bloqueada no tal *scanf*, que a outra atividade não está bloqueada e a máquina virtual do Java pode continuar a chamar métodos a partir de eventos. Se a segunda também se decidir

bloquear por algum motivo, arranja-se simplesmente outra atividade se for necessário. A estas atividades vamos chamar de *threads*.

No caso deste trabalho, um método tem de ficar à espera no *socket* datagrama por um pacote. Enquanto um pacote não vier, ele fica bloqueado. Assim, vamos precisar de uma *thread* só para ele. A outra *thread* vai tratando dos eventos dos botões, caixas, etc., e não fica nunca bloqueada.

3. SEGUNDA 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 uma janela onde recebe mensagens de outros elementos (*Remote*) e uma janela onde escreve as suas mensagens (*Local*). O utilizador selecciona o endereço IP e o número de porto da máquina para onde envia as mensagens e pode desligar a aplicação.

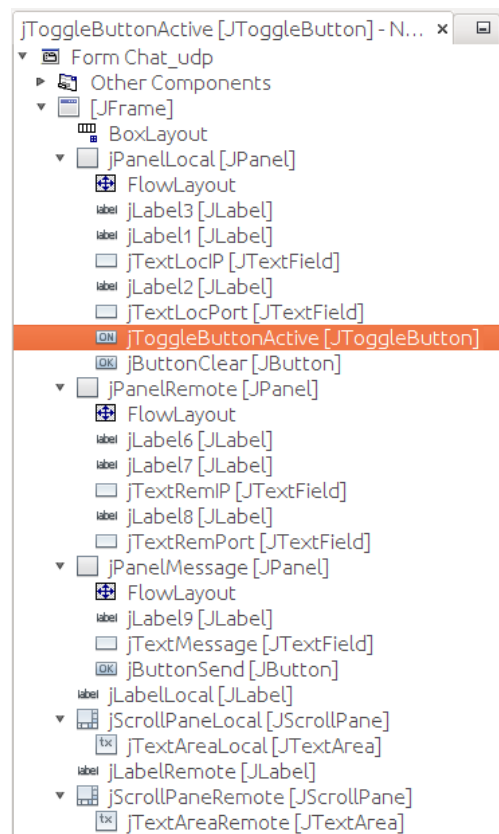
3.1. CHAT_UDP BÁSICO

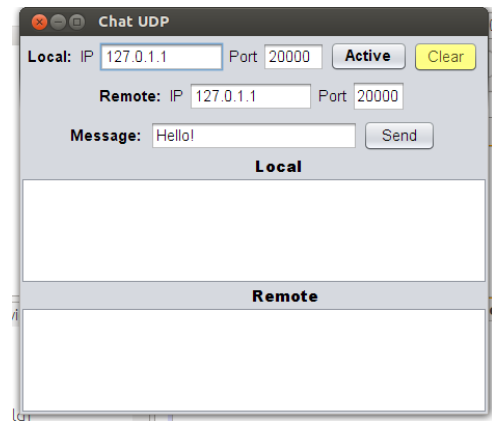
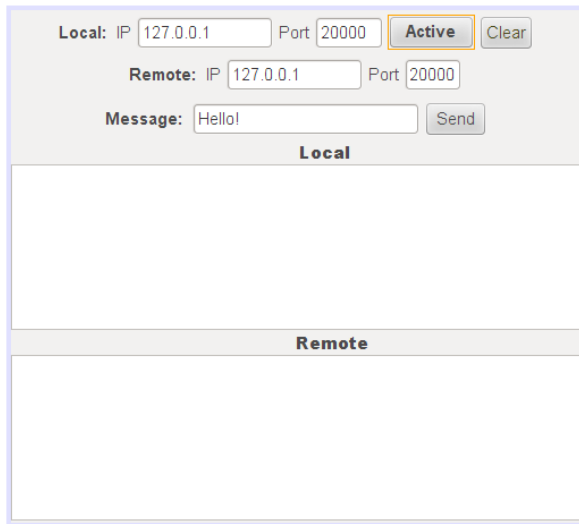
A primeira versão suporta apenas a troca simples de mensagens entre utilizadores, identificados de forma única na rede pelo conjunto {*endereço IP : número de porto*}. É fornecido aos alunos o projeto NetBeans com esta primeira parte completa.

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 fazem 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.

De seguida criaram-se métodos vazios para os botões, com duplo *click*.

Para facilitar a escrita nas caixas de texto “*Local*” e “*Remote*” definiram-se dois métodos na classe associada à janela (`Chat_udp`): `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("");
}
```

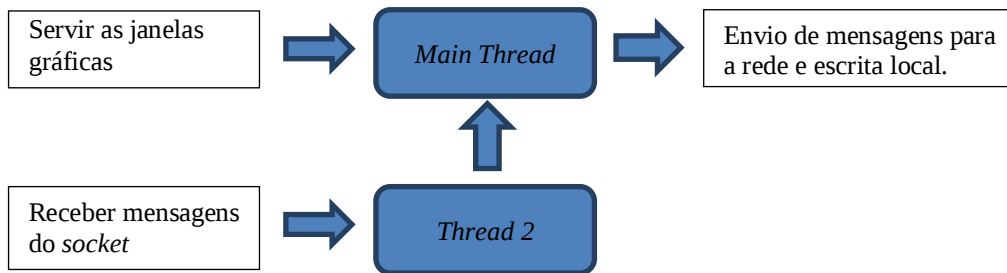
3.1.2. Criação de uma *thread* – classe `Daemon_udp`

O objeto *socket* datagrama tem o problema de ter uma operação bloqueante para se ler o *socket*. O nome desse método é `nome_do_objeto.receive (dp)`. Assim temos de ter uma atividade só para tratar do *socket* de entrada e outra para as janelas gráficas:



Na realidade não se vai usar o esquema da figura em cima. Este esquema implicava que tanto a *thread* principal como a *thread 2* manuseassem os objetos gráficos. Então o melhor é

fazer com que a *thread 2* seja muito simples e 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 isso seleccionou-se o *package* “*udpdemoproject*” e com o botão direito do rato escolheu-se “*New*” e “*Java Class*”. Escolheu-se um nome para ela (por exemplo, *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 o *socket* que tem de ler, e de saber a referência da *thread* principal para depois lhe chamar um método para lhe dar o pacote,
- o método *run* que será corrido quando a *thread* for lançada (através da operação *start*). Este método tem o código que é corrido nesta *thread*.
- o método *stopRunning*, que permite parar a *thread* (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_udp* (objeto/thread principal), passando-lhe o objeto de leitura de campos da mensagem (*dis*). O método *run* também lida com as exceções resultantes de erros na comunicação.

3.1.3. Classe *Chat_udp* – inicialização

A classe *Chat_udp* tem a interface gráfica, e é responsável pela realização de toda a lógica do programa e pela definição das configurações. Uma delas é *MAX_PLENGTH* que define uma constante (porque se usou *final*) a nível da classe (porque se usou *static*) com o tamanho máximo da mensagem trocada.

```

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

```

As variáveis principais *sock* e *serv* guardam o objeto do tipo *DatagramSocket* (que vai ser dado à *thread* 2) e o objeto classe *Daemon_udp*, (que estende *Thread* e é a nossa *thread* 2) e que vai ser criada por esta. A variável auxiliar *formatter*, que serve para formatar a escrita de datas na forma “hora:minutos”, é inicializada logo na declaração pois não depende de nenhum valor externo.

```

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

```

O construtor da classe (método *Chat_udp*) cria os vários objetos gráficos e preenche algumas caixas de texto com o valor do endereço IP da máquina. Primeiro vai saber o endereço IP na forma *InetAddress* e depois usando o método *getHostAddress* escreve essa *string* 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 = null;      // Set null value - meaning "not initialized"
    serv = 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 senão a 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, a `thread 2` 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 = new DatagramSocket(port); // Create UDP socket
            jTextLocPort.setText("" + sock.getLocalPort());
            jTextRemPort.setText("" + sock.getLocalPort());
            serv = new Daemon_udp(this, sock); // Create the receiver thread
            serv.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 (serv != null) { // If thread is running
            serv.stopRunning(); // Stop the thread
            serv = null; // Thread will be garbage collected after it stops
        }
        if (sock != null) { // If socket is active
            sock.close(); // Close the socket
            sock = null; // Forces garbage collecting
        }
        Log_loc("Chat_udp stopped\n");
    }
}
}
```

3.1.4. 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 que foi definido para esta aplicação é:

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

O método `receive_packet` tem também a propriedade de `synchronized` para não ser corrido em paralelo enquanto estiver a ser corrido. Basicamente ele escreve os dados na caixa de texto “*Remote*”:

```
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");
    }
}
}

```

3.1.5. 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. Depois prepara o pacote a enviar do modo como está explicado no documento de “Introdução ao Java”. O objetivo da existência deste método, é ele não necessitar de parâmetros. Por sua vez, ele invoca outro já com os parâmetros necessários.

```

public synchronized void send_packet() {
    if (sock == null) {
        Log_loc("Socket isn't active!\n");
        return;
    }
    InetAddress netip;
    try { // Get IP address
        netip = InetAddress.getByName(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 and log the event
    } catch (Exception e) { // Catches all exceptions
        Log_loc("Error sending packet: " + e + "\n");
    }
}

```

```
}
```

Finalmente, o método `send_one_packet` recebe um pacote UDP (classe `DatagramPacket`) e um endereço (classe `InetAddress`) como parâmetros, e com essa informação define o que falta no pacote (o endereço e o porto) e envia-o para a rede através do *socket*. Note que o código tem dois `catch` para 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.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");
    }
}
```

3.2. CHAT_UDP AVANÇADO – EXERCÍCIOS

Na segunda fase deste trabalho pretende-se que os alunos realizem um conjunto de três exercícios tendo por base a primeira fase do trabalho.

- O primeiro é realizar o envio de cinco mensagens (uma por segundo) durante 5 segundos, controlado por um temporizador, ao premir uma tecla “Send 5”.
- O segundo corresponde a memorizar a última mensagem recebida de cada um dos outros utilizadores.
- O terceiro é enviar uma mensagem para um porto em todas as máquinas de uma rede.

3.2.1. Envio de mensagens usando um temporizador

Para realizar este exercício é necessário criar uma variável (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 `cnt` que funciona como contador – conta o número de vezes que o temporizador ainda deve disparar.

Exercício 4.2.1A: acrescente as seguintes variáveis à declaração da classe `Chat_udp`:

```
private javax.swing.Timer timer; // Timer object
public volatile int cnt;        // Counter - 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 4.2.1B: acrescente o seguinte método à classe `Chat_udp`:

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

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

Exercício 4.2.1C: acrescente a invocação ao método `set_timer_function` passando como argumento 1 segundo (1000 ms). Deve ser colocado depois já terem sido criados com sucesso o `socket` e a `thread`.

Exercício 4.2.1D: acrescente o código que desliga o objeto `timer` quando o botão “Active” for desselecionado.

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

Exercício 4.2.1E: 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 `cnt` de maneira a enviar 5 pacotes.

Lembre-se que `timer` pode ser igual a `null` (porque a aplicação pode não estar ativa).

3.2.2. 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 selecionado, todas as mensagens recebidas devem ser guardadas; quando se desligar o botão deve-se escrever todas as últimas mensagens de todos os utilizadores, um a um.

Exercício 4.2.2A: 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 4.2.2B: 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 no método. De seguida, deve-se guardar na lista o pacote recebido associado ao identificador.

Exercício 4.2.2C: acrescente ao método `receive_packet` (apresentado na secção 3.1.4) a operação de registo na lista se o botão estiver selecionado:

```
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. Vai ser usado um iterador 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 4.2.2D: acrescente à classe `Chat_udp` o método `write_record`, que escreve no terminal o conteúdo da lista e limpa a lista.

```
public void write_record() {
    Iterator<String> it = record.keySet().iterator(); // iterator over keys
    while (it.hasNext()) { // While there are more keys to get
        String remote = it.next(); // remote has next key
        System.out.println("\nCommunication with " + remote);
        System.out.println(record.get(remote)); // Write the last message from remote
    }
    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 4.2.2D: programe o método de tratamento do botão “Record” que criou no exercício 4.2.2A de maneira a invocar o método `write_record` apenas quando se desliga o botão; repare que a rotina vai ser corrida quando se liga e quando se desliga o botão.

3.2.3. Envio de pacotes para todos os computadores na rede

Este último exercício 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.

Caso esteja noutra rede, poderá saber o endereço de rede a partir do endereço IP da máquina e da máscara de rede.

Para controlar o envio, poderá criar um botão com estado (*Toggle Button*) “All”, que desde que esteja ligado leva ao envio para todas as máquinas da rede, no porto selecionado. Portanto, todas as modificações têm apenas de ser feitas no método `send_packet`, que passa a chamar o método `send_one_packet` para todas as máquinas da rede.

Exercício 4.2.3:

- 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`.