FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Departamento de Engenharia Electrotécnica

Sistemas de Telecomunicações

2012/2013

Trabalho 0:
Demonstração do ambiente Java
Aprendizagem do desenvolvimento de aplicações

Aula 3 – Aplicação com sockets orientados à ligação (Versão 2)

Mestrado integrado em Engenharia Eletrotécnica e de Computadores

Luís Bernardo
Paulo da Fonseca Pinto

Índice

1. OBJETIVO

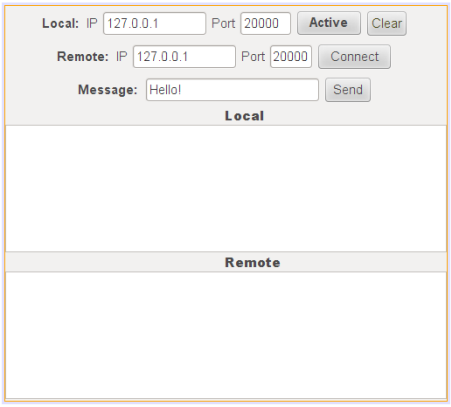
Familiarização com a linguagem de programação Java e com o desenvolvimento de aplicações que comunicam usando *sockets* orientados à ligação no ambiente de desenvolvimento NetBeans. O trabalho consiste na introdução parcial do código seguindo as instruções do enunciado, aprendendo a utilizar o ambiente 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. TERCEIRA APLICAÇÃO – CONVERSA EM REDE COM TCP

Esta secção ilustra o desenvolvimento de uma aplicação utilizando *sockets* orientados à ligação (TCP). A aplicação suporta a troca de mensagens e ficheiros 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 especifica o endereço IP e o número de porto da máquina à qual se pretende ligar, estabelecendo a ligação ou recebendo a ligação de outro utilizador, e podendo terminar a ligação.

2.1. *CHAT_TCP* BÁSICO

A primeira versão suporta apenas a troca simples de mensagens entre utilizadores e apenas suporta uma ligação a um utilizador de cada vez. É fornecido aos alunos o projeto NetBeans com esta primeira parte completa.

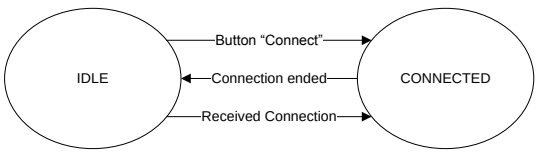


Neste projeto usa-se praticamente a mesma interface gráfica do exemplo da aula anterior (*Chat_udp*) e as funções gráficas associadas, excetuando um botão com estado adicional ("*Connect*"), associado à variável `jToggleButtonConnect`, que foi colocado no painel *Remote*, conforme está representado na figura à direita. Também se mudou o nome da janela principal para "*Chat_tcp*".

Embora a interface gráfica seja quase igual, a realização é substancialmente diferente porque a comunicação com *sockets* TCP usa duas classes. A classse `ServerSocket` é usada para receber ligações, e cada ligação recebida usa `Socket` para comunicar (e no caso do cliente para iniciar uma ligação). Ver o documento "Introdução ao desenvolvimento de aplicações…" que contém uma apresentação destas classes e exemplos do seu uso.

Para além disso, a comunicação com *sockets* TCP inclui duas fases: primeiro estabelece-se a ligação; só depois é que se pode comunicar.

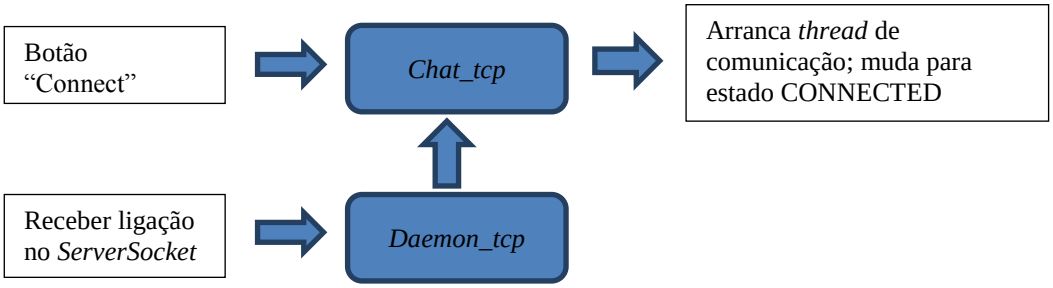
Desta forma, a aplicação vai ter dois estados:



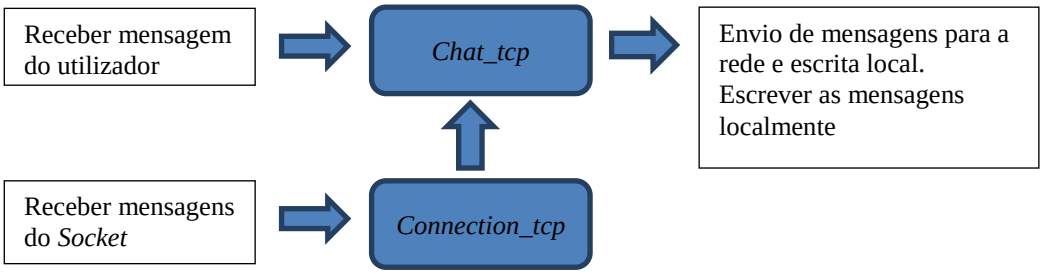
- No estado "IDLE" está à espera de estabelecer ligação, ou por iniciativa própria (botão "*Connect*") ou recebendo uma ligação de outro utilizador;
- No estado "CONNECTED" tem uma ligação ativa, recebe e envia dados em paralelo.

Em qualquer dos estados existem chamadas bloqueantes, pelo que para cada um dos estados existe sempre a necessidade de ter duas atividades em paralelo. A *thread* principal chama-se *Chat_tcp* e está associada à interface gráfica, tratando por exemplo do botão "*Connect*".

No estado IDLE usa-se outra *thread* (*Daemon_tcp*) para esperar por ligações de outros utilizadores. Cada vez que uma ligação é estabelecida a *thread Daemon_tcp* chama o objeto *Chat_tcp* pelo método `start_connection_thread` (Antes de continuar a ler, tente imaginar o que este método faz a partir do seu nome…).



Mal uma ligação seja aceite, o objeto *Chat_tcp* cria um objeto *Connection_tcp* que vai ficar a ler o *socket* de entrada. Estamos no estado CONNECTED e o objeto *Connection_tcp* é uma *thread*. Quando existe uma linha para ler (ver o documento "Introdução a desenvolvimento de aplicações…") é chamado o método `receive_message` do objeto *Chat_tcp*.



Neste momento já deve ser óbvia a estrutura do programa: Existe uma *thread* sempre à espera de novas ligações. Sempre que existe uma nova ligação é criada uma nova *thread* para ler o *socket* dessa ligação. Quando acabar a ligação essa *thread* é destruída.

Nesta versão inicial do programa optou-se por ter apenas uma *thread* extra ligada. Isto é, só pode haver uma ligação TCP e nunca se vai ter uma "*thread 2*", uma "*thread 3*", etc., ligadas ao mesmo tempo e a receber mensagens.

2.1.1. Receção de ligações – classe *Daemon_tcp*

O objeto *Daemon_tcp* fica bloqueado à espera de receber ligações no `ServerSocket ss`. Como se pretende que apenas aceite uma ligação de cada vez, a *thread* apenas corre uma vez a instrução `ss.accept()`, que retorna um objeto `s` da classe `Socket`, que depois é usado para trocar mensagens. Repare-se ainda na utilização das instruções do Java `try-catch-finally`, de forma a garantir que mesmo com erros, a variável `isRunning` fica sempre com o valor `false` no fim da *thread*.

Em relação ao *Chat_udp* também se introduziu uma novidade aqui: o método `stopRunning` invoca a instrução `this.interrupt()` de maneira a garantir que a operação `accept` é interrompida, evitando ter duas ligações ativas.

```
public class Daemon_tcp extends Thread {
  volatile boolean isRunning = false;
  Chat_tcp root;                    // Main window object
  ServerSocket ss;                  // server socket

  public Daemon_tcp(Chat_tcp root, ServerSocket _ss) {  // Constructor
    this.root = root;
    this.ss = _ss;
  }

  public boolean isRunning() {
    return isRunning;
  }

  public void run() {  // Thread "2" code
    isRunning = true;
    try {
      Socket s = ss.accept();    // Waits for a new connection; s is a new Socket
      root.start_connection_thread(s);  // Asks Chat_tcp to start the connection thread
    } catch (Exception e) { // Exception
      if (isRunning) {
        root.Log_rem("Exception in Daemon_tcp : " + e);
      }
    } finally { // Always runs
      isRunning = false;
    }
  }

  public void stopRunning() {   // Stops main thread, interrupting the accept operation
    if (isRunning) {
      isRunning = false;
      this.interrupt();
    }
  }
} // end of class Daemon_ tcp
```

2.1.2. Receção de mensagens numa ligação – classe *Connection_tcp*

O objeto *Connection_tcp* fica em ciclo à espera de receber mensagens do *socket* associado a uma ligação (classe `Socket`), invocando o método `receive_message` da classe *Chat_tcp* para cada mensagem recebida (i.e. cada vez que se recebe uma linha de texto). Este objeto centraliza toda a comunicação através do *socket* disponibilizando os métodos:

- `send_message` para o envio de mensagens, e
- `toString`, que retorna uma *string* com um identificador único constituído por "*endereço IP : número porto*". Devido a definirmos o método `toString`, um objeto desta classe pode ser convertido para `String` automaticamente pelo compilador de Java.

O protocolo usado para enviar as mensagens consiste em acrescentar uma mudança de linha (`"\n"`) ao final da mensagem, permitindo que seja lida pela instrução `in.readline()`, que lê até ao fim da linha, extraindo o fim de linha. Novamente, são usadas as instruções `try-catch-finally`, de forma a garantir que a ligação é fechada quando a *thread* termina. O código dentro de `finally` é sempre executado, independentemente de ter ocorrido um erro, ter-se chamado `return` para sair da função, ou ter-se chegado ao fim do código dentro do `try`.

```
public class Connection_tcp extends Thread {
  volatile boolean keepRunning = true;
  Chat_tcp root;          // Main window object
  Socket s;               // socket
  PrintStream pout;       // Device used to write strings to the socket
  BufferedReader in;      // Device used to read from socket

  Connection_tcp(Chat_tcp _root, Socket _s) {
    this.root = _root;
    this.s = _s;
  }

  public String toString() {  // Returns the remote's ID
    if ((s==null) || !s.isConnected()) {
      return "null";
    }
    return s.getInetAddress().getHostAddress()+":"+s.getPort();
  }

  public boolean send_message(String msg) {  // Sends a message through the connection
    try {
      pout.print(msg + "\n");
      return true;
    } catch (Exception e) {
      return false;
    }
  }

  public void run() {  // Threads code
    try {
      String message;
      in = new BufferedReader(new InputStreamReader(s.getInputStream(), "8859_1"));
      pout = new PrintStream(s.getOutputStream(), false, "8859_1");
      while (keepRunning) {   // Loop waiting for messages
        message = in.readLine();    // Blocks waiting for new messages (line of text)
        if (message == null) {   // End of connection
          return;
        }
        root.receive_message(this, message);  // Calls Chat_tcp object
      }
    } catch (Exception e) { // Catches all errors
      if (keepRunning) {
        System.out.println("Error " + e);
      }
    } finally {   // Always runs this code, even when return is called!
      try {
        s.close();   // Close the socket and all devices associated
      } catch (Exception e) { /* Ignore everything */ }
      root.connection_thread_ended(this); // Inform Chat_tcp object that thread ended
    }
  }

  public void stopRunning() {  // Stops the thread
    keepRunning = false;
    try {
      s.close(); // Closes the socket and all devices associated; triggers an exception
    } catch (Exception e) {
      System.err.println("Error closing socket: "+e);
    }
  }
} // end of class Connection_tcp
```

2.1.3. Classe *Chat_tcp* – inicialização

A classe *Chat_tcp* foi criada associada à interface gráfica, e é responsável pela realização de toda a lógica do programa. São usadas três variáveis principais: `ss`, `conn` e `serv`. A variável `ss` guarda o objeto do tipo `ServerSocket` usado para receber ligações; a variável `conn` vai conter objeto to tipo *Connection_tcp* (ou é `null` caso a *thread* esteja inativa); a variável `serv` vai conter objeto to tipo *Daemon_tcp* (ou é `null` caso a *thread* esteja inativa). Define ainda a

variável auxiliar `formatter`, para formatar a escrita de datas para "*hora:minutos*", que é inicializada logo na declaração, pois não depende de nenhum valor externo.

```
// Variables declaration
private ServerSocket ss;                            // server socket
private Connection_tcp conn                         // Connection object;
private Daemon_tcp serv;                            // thread for connection reception
private java.text.SimpleDateFormat formatter =      // Formatter for dates
                    new java.text.SimpleDateFormat("hh:mm:ss");
```

O construtor da classe define o valor inicial das variáveis, preenche as caixas de texto dos endereços IP com o endereço local, e define o valor do porto local a 20000, por omissão.

```
public Chat_tcp() {
  initComponents();  // defined by NetBeans, creates the graphical window
  ss = null;         // Set null value - meaning "not initialized"
  serv = null;       // Set null value - meaning "not initialized"
  conn = 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");
}
```

Tal como no *Chat_udp*, o arranque da aplicação é controlado na função que trata o botão com estado "*Active*". Quando o botão é ativado, a função lê o número de porto local, cria o `ServerSocket ss`, e cria e arranca a *thread* que recebe ligações, pré-preenchendo o número de porto local e remoto. Em caso de erro, desliga o botão voltando ao estado inicial. Quando o botão é desativado, a função para as *threads* e fecha `ss`.

```
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 {
      ss = new ServerSocket(port);       // Create TCP Server socket
      jTextLocPort.setText("" + sock.getLocalPort());
      jTextRemPort.setText("" + sock.getLocalPort()+1);
      serv = new Daemon_tcp(this, ss);   // Create the connection receiver thread
      serv.start();                      // Start the thread
      Log_loc("Chat_tcp 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 connection receiver thread is running
      serv.stopRunning();    // Stop the thread
      serv = null;           // Thread will be garbadge collected after it stops
    }
    if (conn != null) {      // If connection thread is running
      conn.stopRunning();    // Stop the thread
      conn = null;           // Thread will be garbadge collected after it stops
```

```
      }
      if (ss != null) {        // If server socket is active
        try {
          ss.close();          // Close the socket
        } catch (IOException ex) {  /* Ignore */  }
        ss = null;             // Forces garbadge collecting
      }
      Log_loc("Chat_tcp stopped\n");
    }
  }
```

2.1.4. Classe *Chat_tcp* – controlo da ligação

Há duas maneiras de começar uma ligação: ou recebendo uma ligação através da *thread* da classe *Daemon_tcp*, ou iniciando uma nova ligação através do botão "*Connect*". O primeiro caso foi descrito na secção 2.1.1, onde se mostrou que após receber a ligação se invoca o método `start_connection_thread` da classe *Chat_tcp*. O código do método é apresentado de seguida, mostrando-se que é criado e ativado um objeto *thread* da classe *Connection_tcp*. Para além disso, ativa-se o botão "*Connect*", permitindo ao utilizador parar a ligação.

```
public void start_connection_thread(Socket s) {
  conn = new Connection_tcp(this, s);     // Create the connection thread object
  Log_rem("Connected to " + conn.toString() + "\n");
  jToggleButtonConnect.setSelected(true); // Set ON the "Connect" button
  conn.start();                           // Starts the connection thread
}
```

O segundo método de criar uma ligação está realizado na função de tratamento do botão com estado "*Connect*". Neste caso, se estiver ativo, são lidos os valores dos campos *Remote IP* e *Remote Port* e é criado o objeto `cs` da classe `Socket` ligado ao utilizador remoto. Caso tenha sucesso a criar a ligação, é parada a *thread* `serv` para não receber ligações enquanto tiver a ligação ativa e é iniciada a *thread* `conn` associada à ligação com a função `start_connection_thread`. Em caso de erro, o botão "*Connect*" é deixado desseleccionado.

O botão "*Connect*" é usado para parar uma ligação quando é desligado (fica desseleccionado).

```
private void jToggleButtonConnectActionPerformed(java.awt.event.ActionEvent evt) {
  if (jToggleButtonConnect.isSelected()) { // The button is ON - Start the connection
    if (con != null) {  // A Connection is active; ignore request
      return;
    }
    InetAddress netip;
    try { // Test IP address in Remote IP text box
      netip = InetAddress.getByName(jTextRemIP.getText());
    } catch (UnknownHostException e) {
      Log_loc("Invalid remote host address: " + e + "\n");
      jToggleButtonConnect.setSelected(false);
      return;
    }
    int port;
    try { // Test port
      port = Integer.parseInt(jTextRemPort.getText());
    } catch (NumberFormatException e) {
      Log_loc("Invalid remote port number: " + e + "\n");
      jToggleButtonConnect.setSelected(false);
      return;
    }
    try {
      Socket cs = new Socket(netip, port); // Create and connect a socket to the remote
      if (cs != null) {   // Is connected
        start_connection_thread(cs);       // Start the connection thread
        serv.stopRunning();                // Stop connection receive thread
        serv= null;
      }
```

```
      } catch (Exception ex) {
        Log_loc("Connection to " + jTextRemIP.getText() + ":" + jTextRemPort.getText() +
                  " failed\n");
        jToggleButtonConnect.setSelected(false);
      }
    } else {  // The button is OFF - stop the connection
      if (con != null) {
        con.stopRunning();  // Stop the connection
        con= null;
      }
    }
  }
```

Sempre que a *thread* `Connection_tcp` termina, é invocado o método `connection_thread_ended` da classe *Chat_tcp*. Este método torna a reativar a *thread Daemon_tcp* e a desligar o botão "*Connect*", voltando a estar preparado para criar novas ligações.

```
  public void connection_thread_ended(Connection_tcp th) {
    Log_rem("Connection to "+ th.toString() + " ended\n");
    conn = null;
    serv = new Daemon_tcp(this, ss);   // Create the connection receiver thread
    serv.start();                      // Start the thread
    jToggleButtonConnect.setSelected(false); // Set OFF the "Connect" button
  }
```

2.1.5. Classe *Chat_tcp* – envio de mensagens

As mensagens são enviadas através do botão "*Send*", que invoca o método `send_message`. A realização é simples neste caso, pois recorre ao método `send_message` da classe *Connection_tcp*, após obter o texto que está na caixa de texto *Message*.

```
  public synchronized void send_message() {
    if (conn == null) {
      Log_loc("Connection isn't active!\n");
      return;
    }
    String message = jTextMessage.getText();   // Get the text from the Message box
    if (message.length() == 0) {
      Log_loc("Empty message: not sent\n");
      return;
    }
    if (conn.send_message(message)) {          // Send the message using the conn object
      // Write message to jTextAreaLocal
      Log_loc(formatter.format(new Date()) + " - sent '" + message + "'\n");
    }
  }
```

2.1.6. Classe *Chat_tcp* – receção de mensagens

A receção de mensagens é realizada na *thread Connection_tcp*, mas o conteúdo da mensagem é passado ao objeto da classe *Chat_tcp* através do método `receive_message`, que se limita a escrever o conteúdo na janela respetiva.

```
  public synchronized void receive_message(Connection_tcp con, String msg) {
    try {
      Date date = new Date();  // Get reception date
      // Write message contents
      Log_rem(formatter.format(date) + " - received '" + msg + "'\n");
    } catch (Exception e) {
      Log_rem("Error in receive_message: " + e + "\n");
    }
  }
```

2.2. *CHAT_TCP* AVANÇADO – EXERCÍCIOS

Na segunda fase do trabalho pretende-se que os alunos realizem dois exercícios:

- O primeiro consiste em acrescentar um botão para enviar um ficheiro;
- O segundo corresponde a modificar o programa para aceitar várias ligações em paralelo.

2.2.1. Envio do conteúdo de um ficheiro

Para realizar esta tarefa é necessário acrescentar um novo botão "*Send File*" e um objeto de escolha de ficheiro (`File Chooser` em *Swing Windows*). A rotina do novo botão deverá depois abrir uma janela de escolha de ficheiro (ver exemplo da Calculadora) e invocar um método de envio de ficheiro (por exemplo `send_file`) a criar na classe *Connection_tcp*. O método `send_file` recebe o ficheiro (e.g. `File f`) e deve, por esta ordem, abrir o ficheiro em modo leitura, criar um objeto `FileInputStream`, criar um *buffer* (`byte []`), ler o conteúdo do ficheiro para o *buffer* e enviar o conteúdo para o *socket*, não se esquecendo de fechar o objeto `FileInputStream`:

```
public boolean send_file(File f) {   // Should be public because is called by Chat_tcp
  FileInputStream fis = null;
  try {
    fis = new FileInputStream (f); // Open file input stream
    byte[] buffer= new byte[fis.available()];// allocate a buffer with the
                                             //     length of the file
    int n= fis.read(buffer); // read the entire file to buffer;
                             //n counts the number of bytes actually read
    if (n != buffer.length) {
      root.Log_loc("Did not read the entire file\n");
      return false;
    }
    pout.write(buffer, 0, n); // write to socket
    return true;
  } catch (IOException ex) {
    root.Log_loc("Error sending file "+f+"\n");
    return false;
  } finally {
    try {
      fis.close();   // Always close the file
    } catch (IOException ex) { /* Ignore error */ }
  }
}
```

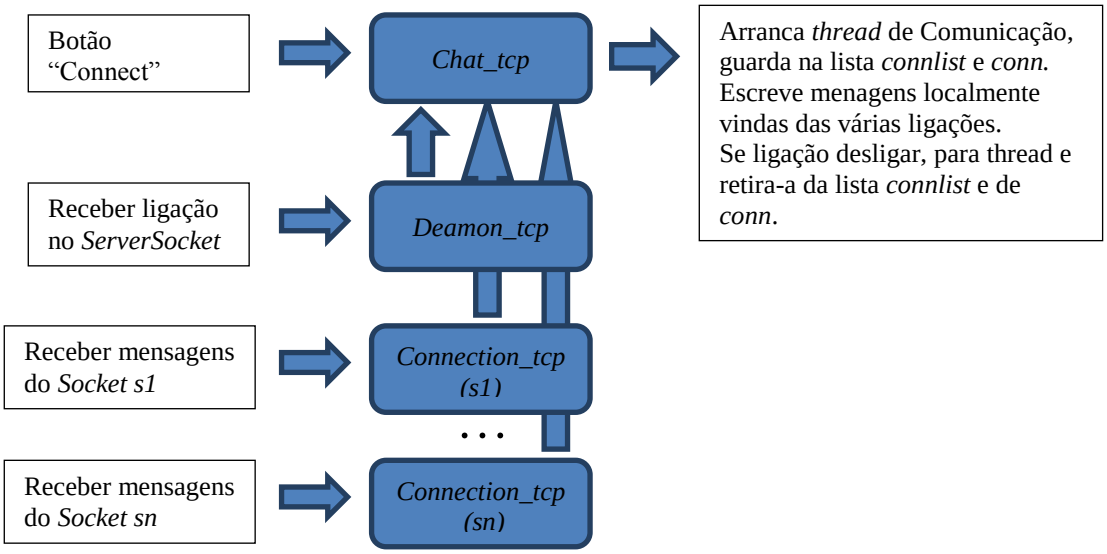
Exercício 5.2.1: Acrescente o método `send_file` à classe *Connection_tcp*. Implemente a funcionalidade de envio de ficheiro, descrita anteriormente, na classe *Chat_tcp*. Consulte a documentação de apoio fornecida com o enunciado do trabalho e o exemplo da Calculadora.

2.2.2. Várias ligações em paralelo

Este exercício é complexo, e vai requerer a utilização avançada de lista, de *threads* e de classes. Recomenda-se que antes de o realizar relembre como são usadas as *HashMaps* no exercício *Chat_udp* e que observe com muita atenção como são geridas as duas *threads* no trabalho, tal como está agora.

Quando se passa a permitir usar várias ligações em paralelo, surge a necessidade de ficar "à espera" de mensagens em muitas ligações. Em Java, necessitamos de uma *thread* para cada ligação, portanto, vamos usar "muitas" *threads* em paralelo. Vamos continuar a usar as *threads* anteriores (*Daemon_tcp* e *Connection_tcp*), mas o objeto (a *thread*) para receber ligações (*Daemon_tcp*) está sempre ativo e podem existir vários objetos (*threads*) de ligação (*Connection_tcp*) ativos.

Para guardar as ligações usa-se uma estrutura de dados do tipo `HashMap<String, Connection_tcp>`, isto é, uma lista de objetos de ligação, indexada pela *string* "*IP:porto*" (que pode ser obtida num objeto da classe *Connection_tcp*, com o método `toString()`).Vai-se autorizar apenas uma ligação iniciada pelo utilizador (no botão "*Connect*"), que vai ficar guardada na variável `conn`, já existente no programa.



Exercício 5.2.2A: Declare e inicialize a lista `connlist` na classe *Chat_tcp*:

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

A classe *Connection_tcp* já está preparada para ter vários objetos a correr em paralelo. Torna-se apenas necessário modificar o que acontece cada vez que se arranca ou se para um objeto desta classe. Quando se arranca um objeto é necessário acrescentá-lo à lista (não se pode modificar o objeto `conn`, pois agora "significa" ligação local).

Exercício 5.2.2B: Modifique o método `start_connection_thread` da classe *Chat_tcp* de maneira a acrescentar o novo objeto à lista `connlist`, retornando o objeto criado.

```
public Connection_tcp start_connection_thread(Socket s) {
  Connection_tcp c = new Connection_tcp(this, s);// Create the connection thread object
  Log_rem("Connected to " + c.toString() + "\n");
  c.start();                          // Start the connection thread
  connlist.put(c.toString(), c);  // Add object c to the list
  return c;
}
```

Quando um objeto *Connection_tcp* termina, é necessário removê-lo da lista, e caso seja o objeto `conn`, é necessário limpar a variável. Deixa-se de modificar a *thread Daemon_tcp*.

Exercício 5.2.2C: Modifique o método `connection_thread_ended` da classe *Chat_tcp* de maneira a acrescentar o novo objeto à lista `connlist`, retornando o objeto criado.

```
public void connection_thread_ended(Connection_tcp th) {
  Log_rem("Connection to " + th.toString() + " ended\n");
  if (th == conn) {      // if it is the thread initiated locally
    conn = null;
    jToggleButtonConnect.setSelected(false);  // Set the Connect button OFF
  }
  connlist.remove(th.toString());  // Removes the thread from the list using the key
}
```

Para suportar várias ligações em paralelo, a *thread Daemon_tcp* deixa de receber apenas uma ligação, mas passa a ficar indefinidamente em ciclo à espera de novas ligações, de forma semelhante ao que acontecia na classe *Daemon_udp*.

Exercício 5.2.2D: Modifique a classe *Daemon_tcp* de maneira a que fique em ciclo sempre à espera de novas ligações, e só saia em caso de erro ou quando o utilizador termina a *thread*.

O botão "*Connect*" também muda o seu funcionamento. Quando se liga, deve guardar a *thread Connection_tcp* criada na variável `conn`, não modificando a *thread Daemon_tcp*. A ideia é usar o valor de `conn` para saber se o objeto iniciou (`conn != null`) ou não (`conn == null`) uma ligação. Quando se desliga, deve parar apenas a *thread* `conn`, e removê-la da lista.

Exercício 5.2.2E: Modifique o método `jToggleButtonConnectActionPerformed` da classe *Chat_tcp* de maneira a ligar/desligar uma *thread* de ligação iniciada pelo utilizador, de acordo com o descrito no parágrafo anterior.

O envio de mensagens (e de ficheiros) deve ser enviado por todas as ligações.

Exercício 5.2.2F: Modifique o método `send_message` da classe *Chat_tcp* de maneira a enviar as mensagens por todas as ligações. O método `send_message` deve ser invocado sobre todos os objetos da lista `connlist`. Para obter todos os objetos, pode-se usar um iterador sobre os valores, obtido com:

```
Iterator<Connection_tcp> it= connlist.values().iterator();
```

A mensagem é enviada desde que a lista não esteja vazia (i.e. `!connlist.isEmpty()`).

Exercício 5.2.2G: Modifique o método associado ao envio de ficheiro da classe *Daemon_tcp* de maneira a enviar o ficheiro para todas as ligações.

Para concluir a aplicação, falta apenas modificar o tratamento do botão "*Active*" de maneira a terminar TODAS as ligações ativas quando se desliga a aplicação.

Exercício 5.2.2H: Modifique o método `jToggleButtonActiveActionPerformed` da classe *Daemon_tcp* de maneira a parar todas as ligações ativas e limpar a lista no fim (`connlist.clear()`). Tal como nos exercícios anteriores, deve "mandar parar" cada *thread* individualmente.