FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Departamento de Engenharia Electrotécnica

Sistemas de Telecomunicações

2012/2013

Trabalho 1:
Aplicação sobre sockets

Mestrado integrado em Engenharia Eletrotécnica e de Computadores

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

Índice

1. OBJETIVO

Familiarização com o uso de *sockets* para comunicação entre máquinas e com o funcionamento da interface de programação *sockets* do Java.

O trabalho consiste no desenvolvimento de um sistema de transferência de ficheiros com um registo inicial de pedidos.

> **Sugestões**: Em certas partes do enunciado aparece algum texto formatado de um modo diferente que começa com a palavra "Sugestões". Não é obrigatório seguir o que lá está escrito, mas pode ser importante para os alunos ou grupos onde ainda não haja um à-vontade muito grande com programação, estruturas de dados e algoritmia.

2. ESPECIFICAÇÕES

Pretende-se desenvolver um sistema de transferência de ficheiros com inscrição prévia de clientes. O sistema de transferência comporta-se como um portal para os clientes dado que têm de conhecer apenas um endereço e porto de acesso. O serviço de chaves e o serviço de ficheiros têm portos de acesso não conhecidos publicamente.

A ideia geral é a seguinte: os clientes acedem ao servidor de registos (*trader*) para saber o endereço e porto do servidor de chaves (*key*) e de seguida ao acederem ao servidor de chaves obtêm uma chave (*key*) que terão de apresentar quando acederem a qualquer serviço registado no portal. De seguida, acedem novamente ao servidor de registos para obterem o porto do serviço de transferência de ficheiros. Na posse da chave juntamente com o porto do servidor de pedidos de ficheiro, acedem ao serviço de transferência de ficheiros para fazer a transferência de **um** ficheiro. Na resposta ao pedido, o cliente recebe informação relativa ao comprimento do ficheiro (0 significa ficheiro vazio e -1 significa inexistente) e o conteúdo do ficheiro. A chave caduca ao fim **de um minuto** se não for usada para um pedido ao servidor de ficheiros. É sempre necessário pedir uma chave para cada ficheiro transmitido, mesmo que a transferência de ficheiro falhe por o ficheiro não existir.

A estrutura do trabalho consiste num bloco (processo) servidor e num bloco (processo) cliente. O bloco servidor tem as seguintes componentes:

- Servidor de registos (*trader*) – Está sempre à espera de pedidos de clientes para um dos dois serviços. Responde a cada pedido válido com o porto do servidor, e a cada pedido inválido com a indicação de "inválido".
- Servidor de chaves – Recebe um pedido de inscrição e devolve uma chave válida para uma transferência, que tem de ser efectuada em menos de um minuto.
- Servidor de ficheiros – Recebe um pedido de transferência de ficheiros binários, em que nos parâmetros vêm o nome do ficheiro e a chave. Se a chave não for válida, ou se o ficheiro não existir devolve comprimento menor ou igual a 0. No caso contrário devolve o conteúdo do ficheiro.

O bloco cliente executa os procedimentos para se transferir um ficheiro.

2.1. FLUXO DE MENSAGENS

A sequência de mensagens está mostrada na figura 1, e consiste nas seguintes mensagens (assume-se que não existem erros do tipo de serviços inválidos, chaves inválidas, etc.):

(1) – O cliente envia uma mensagem de pedido do serviço de chaves ao servidor de registos;

(2) – O servidor de registos responde com o endereço do socket do servidor de chaves;
(3) – O cliente pede uma chave ao serviço de chaves;
(4) – O serviço de chaves gera uma chave e envia-a ao cliente. O tempo de um minuto começa a contar a partir desse momento e termina quando o cliente aceder ao servidor de ficheiros;
(5) – O cliente envia uma mensagem de pedido do serviço de ficheiros ao servidor de registos;
(6) – O servidor de registos responde com o endereço do socket do servidor de chaves;
(7) – O cliente procede à transferência de ficheiros:
(7_a) – O cliente estabelece uma ligação TCP com o servidor de ficheiros;
(7_b) – Inicialmente o cliente envia duas linhas de texto através da ligação TCP a chave e o nome do ficheiro;
(7_c-7_{n-1}) – O servidor de ficheiros responde com o tamanho do ficheiro que pode tomar valores zero, ou negativos para indicar erros. No caso de existir o ficheiro pedido, o servidor envia os bytes do ficheiro;
(7_n) – O servidor desliga a ligação.

Na comunicação com o servidor de registos e o servidor de chaves são usados *sockets datagrama*. Para a comunicação com o serviço de ficheiros são usados *sockets* orientados à conexão**.**

Os passos 7_c a 7_{n-1} correspondem ao envio do comprimento e dos dados do ficheiro. Não se sabe ao certo quantos pacotes serão pois o *socket* orientado à conexão é um feixe de octetos e não de pacotes, como nos *socket datagrama* (aliás para os alunos perceberem melhor esta diferença deve-se escrever no *socket* blocos de 1000 bytes de cada vez e o cliente deve ler blocos de 2000 bytes de cada vez).

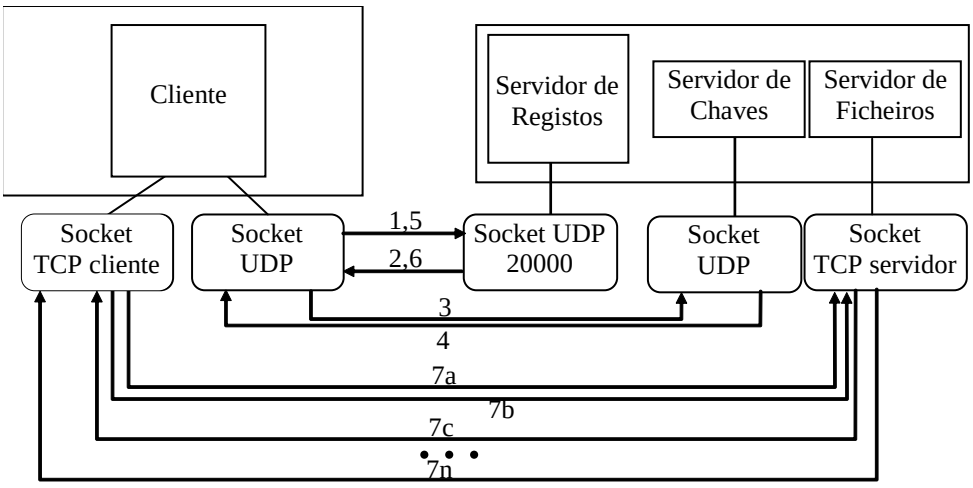
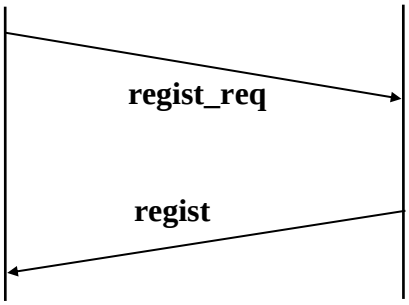


Fig 1 – Sequência de mensagens do trabalho

2.2. SERVIÇO DE REGISTOS

O acesso ao serviço de registos utiliza *sockets datagrama*.O acesso está representado nas mensagens 1,2 e 5,6 da Fig. 1, que obedecem ao seguinte protocolo de pedido-resposta:




```
Mensagem regist_req:// sequência contígua de
 int     porto;  // Porto UDP do cliente
 short codServ;  // Código de serviço pedido
   /* código serviço chaves    = 15 */
   /* código serviço ficheiros = 31 */


Mensagem regist: // sequência contígua de
 boolean validSer; // validade do serviço
   /*  false - serviço inexistente */
   /*  true  - serviço válido      */
 int     portoSer; // Porto de servidor :
```

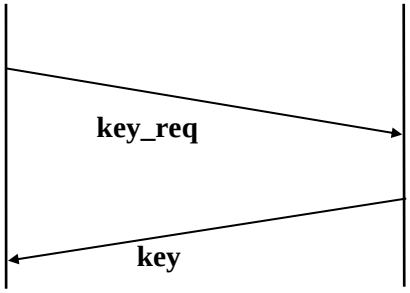
As mensagens *regist_req e regist* têm os dois parâmetros indicados. Caso o porto em *regist_req* não corresponda ao porto do socket UDP de onde veio a mensagem, deve ser devolvido "serviço inexistente".

Como se disse, o *socket* do servidor de registos é o único conhecido. Costuma-se chamar a isso ***well-known socket*** (o endereço IP é o da máquina respetiva e o que realmente é conhecido é o porto). Todos os outros *sockets* devem ter portos livres pois os seus valores são fornecidos pelos vários componentes. O porto do *socket* conhecido do primeiro servidor executado é o **20020**. Caso arranque mais do que um servidor na mesma máquina, estes terão os números 20021, 20022, etc.

Sugestões: O servidor de registos tem de saber qual o porto do *socket datagrama* associado ao serviço de chaves e do *socket stream* do servidor de ficheiros. Os portos podem ser obtidos usando o método getLocalPort dos objetos `ServerSocket` e `DatagramSocket` associados.

2.3. SERVIÇO DE CHAVES

O acesso ao serviço de chaves usa *sockets datagrama*, sendo ilustrado pelas mensagens 3 e 4, que obedecem ao protocolo seguinte:




```
Mensagem key_req: //sequência contígua de
 int porto;  // Porto UDP do cliente


Mensagem key:  //sequência contígua de
 short  key_len; // nº bytes da chave
 byte[] key;     // chave
```

O cliente envia o pedido usando o porto que lhe foi dado pelo servidor de registos. A resposta contém a chave.

Sugestões 1 – criação de chaves: Para simplificar o trabalho, o servidor de chaves pode devolver inicialmente chaves que sejam incrementos de uma variável inteira (talvez começando num valor diferente de zero). Caso tenha tempo, deverá substituir essa forma inicial, por chaves aleatórias; por exemplo, construídas a partir de um número aleatório (ver excerto de código abaixo). Neste caso, tem de garantir que as chaves que existem num dado instante são únicas, e que não existem duas chaves iguais ativas num dado instante.

```
private Random keygen= new Random();
int number= keygen.nextInt(1000000);          // Random number between 0 and 999999
```

Sugestões 2 – Vários clientes ativos

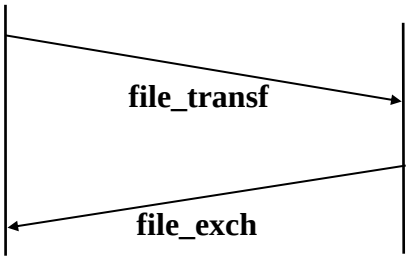
Numa primeira fase pode admitir que só existe um cliente ativo em cada instante (se isso lhe parecer que simplifica a programação). Caso disponha de tempo no final do trabalho, deve pensar em suportar vários clientes em paralelo. Neste caso, quando o servidor de chaves gera uma chave para um cliente deve colocá-la acessível ao servidor de ficheiros. Recomenda-se a utilização de uma lista para guardar as chaves alocadas e para gerir o tempo de validade das chaves. Mal haja uma transferência, ou caso passe o tempo, as chaves devem ser retiradas da lista. Para guardar a chave, os alunos devem criar uma nova classe, com todos os dados que considerem relevantes para gerir o tempo.

Sugestões 3 – Remoção de chaves

Quando passar um minuto, uma chave não usada deve ficar inválida e deve ser retirada da lista. Esta operação pode ser realizada por um temporizador, mas não pode ser realizada apenas com uma passagem sobre a lista, pois o iterador usado bloqueia qualquer remoção de elementos da lista. É necessário alocar uma lista ou array de tamanho variável auxiliar para guardar as chaves "fora de prazo", que poderão ser removidas apenas no fim.

2.4. SERVIÇO DE FICHEIROS

O acesso ao serviço de transferência de ficheiros deve utilizar *sockets* orientados à conexão e obedecer ao protocolo:




```
file_transf : //sequência contígua de
 String key + "\n";      // Chave
 String filename + "\n"; // Nome ficheiro
```

```
file_exch : //sequência contígua de
 String tamanho ficheiro+"\n";
   /* -2 - chave inválida          */
   /* -1 - ficheiro inexistente    */
   /*  0 - ficheiro vazio          */
   /*  comprimento do ficheiro     */
 contents /* sequência de carateres
             com conteúdo do ficheiro */
```

A comunicação das mensagens *file_transf* e *file_exch* vai ser realizada através da troca de *strings* delimitadas por mudanças de linha. A excepção é o conteúdo dos ficheiros que deve ser transmitido em formato binário, sem modificar os dados transmitidos. **Não se esqueça que** o servidor envia, em cada ciclo de envio, apenas 1000 bytes do ficheiro (caso envie todo o ficheiro numa escrita única vai ser penalizado na avaliação final do trabalho). No final, o servidor termina a ligação. Caso decida tentar suportar vários clientes em paralelo, lembre-se que necessita de ter uma thread para cada cliente, onde deve guardar todos os parâmetros relevantes para essa ligação.

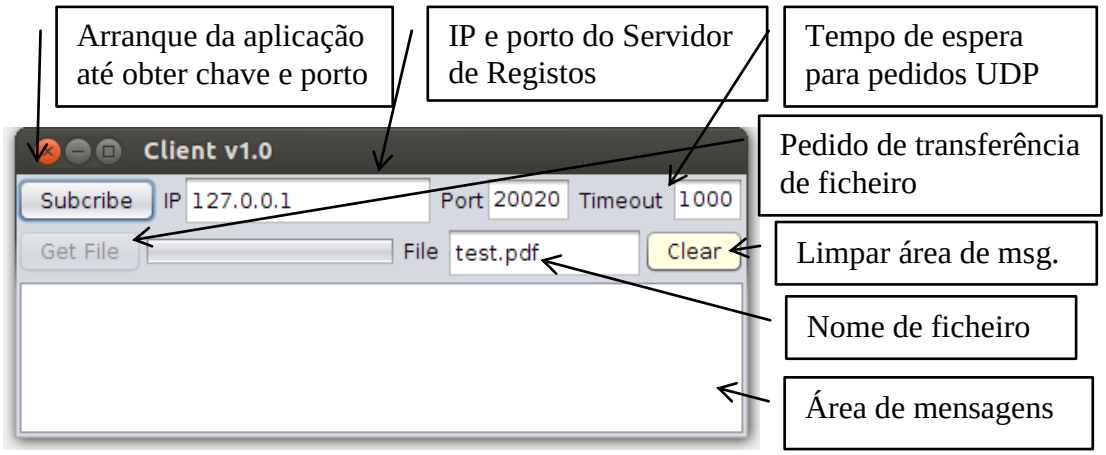
Sugestões: Não se esqueça que não não deve em nenhuma circunstância bloquear a thread principal (associada à interface gráfica). Qualquer operação bloqueante (e.g. leitura ou escrita no *socket*) deve ser realizada dentro de uma thread.

3. DESENVOLVIMENTO DO PROGRAMA

3.1. CLIENTE

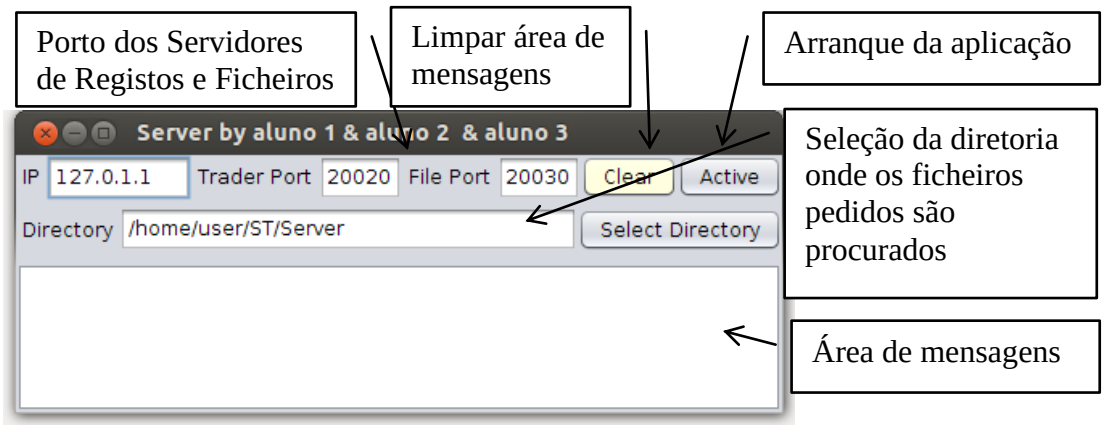
A aplicação cliente é fornecida totalmente realizada juntamente com o enunciado do trabalho. Foi desenvolvida uma aplicação em Java com a interface gráfica representada em baixo. Quando se prime o botão "*Subscribe*" o cliente realiza as trocas de mensagens representadas entre 1 a 6 na figura 1. Desta forma, o programa está pronto a comunicar com o servidor de ficheiros, ficando desbloqueado o botão "*Get File*". Quando é premido, realizam-se as trocas de mensagens representadas como 7 na figura 1. Durante a transferência a barra ilustra a evolução da receção de dados do ficheiro. Em qualquer altura é possível abortar uma transferência premindo o botão "*Subscribe*".

Pode correr o cliente a partir do terminal com o comando: `java -jar Client.jar`



3.2. SERVIDOR

O trabalho consiste na realização do bloco servidor. Para facilitar o desenvolvimento do programa e tornar possível o desenvolvimento do programa durante as quatro aulas previstas, é fornecido juntamente com o enunciado um programa *Server* incompleto, com a interface gráfica representada abaixo, que já realiza parte das funcionalidades pedidas. Cada grupo pode fazer todas as modificações que quiser ao programa base, ou mesmo, desenhar uma interface gráfica de raiz. No entanto, recomenda-se que invistam o tempo na correta realização dos protocolos propostos neste enunciado.



O programa fornecido é composto por quatro classes:

- *Daemon_udp.java* (completa) – Thread que recebe pacotes em sockets datagrama;
- *Daemon_tcp.java* (completa) – Thread que recebe ligações para o serviço de ficheiros;
- *SConn_tcp.java* (**a completar**) – Thread que trata uma ligação do serviço de ficheiros;
- *Server.java* (**a completar**) – Classe principal com interface gráfica, que faz a gestão de sincronismo dos vários objectos usados.

O programa fornecido reutiliza com modificações menores as classes *Daemon_udp* e *Daemon_tcp* que foram usadas nos trabalho 0 de aprendizagem, e propõe duas classes incompletas que suportam a interface com o utilizador (*Server*) e a comunicação numa ligação TCP.

A classe *Daemon_tcp* já foi usada na terceira aula do trabalho 0.

A classe *Daemon_udp* da segunda aula do trabalho 0 foi modificada neste projeto para poder ser usada com vários sockets UDP em paralelo. Quando se cria uma instância desta classe, define-se um tipo (um número) único. Quando se recebe um pacote, o método chamado (`receive_packet`) recebe como primeiro parâmetro o valor do tipo, podendo coexistir vários objetos desta classe com valores de tipos distintos.

```
public class Daemon_udp extends Thread {
    Server root;                        // Main window object
    DatagramSocket ds;                  // datagram socket
    int type;                           // socket type

    public Daemon_udp(Server root, DatagramSocket ds, int type);

    public void run();  // UDP socket thread code
     // Calls root.receive_packet(type, dp, dis);
     //   everytime a new packet is received

    public void stopRunning();  // Interrupt a running thread
}
```

A classe *SConn_tcp* define uma thread mas está praticamente vazia, sendo deixado aos alunos a tarefa de a programar. No final, antes de terminar, a thread deverá chamar o método `root.connection_thread_ended` informando o fim da ligação.

A classe *Server* é a classe principal. Inclui a definição da interface gráfica, incluindo o arranque dos sockets do serviço de registos (*trader*) e do *socket* TCP. Também inclui o código para escolher a diretoria para ler ficheiros, e um conjunto de *callbacks* vazias, que devem ser realizadas pelos alunos ao longo do trabalho.

```
public class Server extends javax.swing.JFrame {
  public final static int MAX_MLENGTH;   // Maximum UDP message size
  public final static long KEY_VALIDITY; // Key validity (ms)

  /* Types of Service in Registration service */
  public final static byte KEY_SERVICE;  // Key service
  public final static byte FILE_SERVICE; // File Service

  public final static int TRADER_SOCKET = 101;  // Type of trader socket

  public void Log(String text); // Log function

  // Handle "Active" toggle button
  private void jToggleActiveActionPerformed(java.awt.event.ActionEvent evt);

  private void jButtonClearActionPerformed(java.awt.event.ActionEvent evt):

  public void close_all(); // Close all sockets and threads

  /* Logs an error message associated to the reception of a packet */
```

```
    public void Log_err(DatagramPacket dp, String type, String err);

    // Handle trader request – called by 'receive_packet'
    public void handle_trader_packet(DatagramPacket dp, DataInputStream dis);

    // Handle an UDP packet of type 'type'
    public void receive_packet(int type, DatagramPacket dp, DataInputStream dis);

    public File get_fileref(String filename); // Return the file descriptor associated
                                              //      to a filename

    SConn_tcp start_connection_thread(Socket s); // Handle TCP connections received;

    void connection_thread_ended(SConn_tcp th);  // Handle end of TCP Thread

  // Variables declared
    private DatagramSocket tradersock;  // UDP socket of the trader service
    private Daemon_udp daemon_trader;  // Thread of the trader service
    private ServerSocket ss;  // TCP socket of the file service
    private Daemon_tcp daemon_tcp;  // Thread of the file service
    public SimpleDateFormat formatter;  // Formatter for dates
  }
```

3.3. METAS

Uma sequência para o desenvolvimento do programa *Server* poderá ser:

1. Programar o arranque do socket e da thread para o serviço de chaves no método `jToggleActiveActionPerformed` da classe `Server`. Deve também programar o método `close_all` para os terminar;
2. Programar o método que trata a resposta ao pedido ao serviço de registos, lendo e validando os campos do pedido, e preparando a resposta;
3. Programar o método que trata o pedido ao serviço de chaves, lendo e validando os campos do pedido, e preparando a resposta. Nesta primeira implementação pode-se devolver uma chave com um valor interior incrementado linearmente;
4. Programar o método `start_connection_thread` da classe `Server` de maneira a criar um objeto da classe `SConn_tcp`, lançando de seguida a thread associada. Se não se sentir confortável a lidar com múltiplas threads, numa primeira fase pode preparar o programa para ter apenas uma thread destas ativas, deixando para mais tarde o suporte de múltiplas threads;
5. Programar a classe `SConn_tcp` de maneira a realizar o protocolo indicado para o serviço de ficheiros (não esquecer de enviar o ficheiro em blocos, para evitar penalizações). Programar o método `connection_thread_ended` da classe `Server`;
6. Programar o suporte de várias ligações de clientes em paralelo, incluindo a possibilidade de parar todas as threads, no método `close_all`;
7. Realizar o mecanismo de validação da chave recebida no serviço de ficheiros, para chaves com valores aleatórios. Sugere-se que seja criada uma classe auxiliar, para guardar a informação sobre a chave e sobre a sua validade. Depois, esta classe poderá ser usada para criar uma lista de objetos, com as chaves alocadas.
8. Realizar um mecanismo automático de limpeza da lista de chaves, suportado num temporizador que corre periodicamente.

TODOS os alunos devem tentar concluir **pelo menos a fase 5**. Na primeira semana do trabalho é feita uma introdução geral do trabalho, devendo-se terminar a fase 2. No fim da segunda semana devem ter iniciado a fase 5. No fim da terceira semana deve ter começado a realizar a fase 7. No fim da quarta e última semana devem tentar realizar o máximo de fases

possível, tendo sempre em conta que é preferível fazer menos e bem (a funcionar e sem erros), do que tudo e nada funcionar.

As fases 2-3 e 4-5 podem ser desenvolvidas em paralelo em grupos com dois ou três elementos, encurtando o tempo de desenvolvimento do trabalho.

POSTURA DOS ALUNOS

Cada grupo deve ter em consideração o seguinte:

- Não perca tempo com a estética de entrada e saída de dados
- Programe de acordo com os princípios gerais de uma boa codificação (utilização de indentação, apresentação de comentários, uso de variáveis com nomes conformes às suas funções...) e
- Proceda de modo a que o trabalho a fazer fique equitativamente distribuído pelos dois membros do grupo.