Redes de Computadores

Aulas Práticas

Protocolo HTTP
e
Servidores concorrentes

Aula #10

Sumário

- Protocolo HTTP
- Exercícios Wireshark, TCP / HTTP
- Servidores concorrentes (multithreaded)

HTTP

- Protocolo cliente/servidor
- Servidor espera conexões dos clientes num socket TCP no porto 80 (por omissão)
- Na versão 1.0, cada conexão é apenas usada para obter um recurso (i.e. após enviar a primeira resposta, o servidor fecha a conexão com o cliente)
- Na versão 1.1, uma conexão pode ser usada para obter uma sequência de recursos

HTTP - Pedido

- Um pedido HTTP é composto por:
  - uma linha de texto com a instrução de pedido
    - Ex: `GET  /index.html  HTTP/1.0`
  - uma sequência de linhas de texto de cabeçalho
    - Ex.:

```
User-Agent: Mozilla/4.7(X11;Linux 2.4 i586; Nav)
Content-Length: 374
```

  - uma linha vazia (CR+LF) ("\r\n")
  - corpo do pedido (pode não existir)
    - O cabeçalho Content-Length indica o número de bytes do corpo
    - Ex.: numa instrução de POST, os dados enviados ao servidor

HTTP - Resposta

- Uma resposta HTTP é composta por:
  - uma linha de texto com a resposta (versão e código)
    - Ex: `HTTP/1.1  200  OK`
  - uma sequência de linhas de texto de cabeçalho
    - Ex:

```
Server: Apache/2.2.9
Content-Length: 2119
```

  - uma linha vazia (CR+LF) ("\r\n")
  - Conteúdo da resposta
    - Ex.: o conteúdo do ficheiro ou recurso pedido

HTTP - Exemplo

- Para experimentar o funcionamento do HTTP:
  – Utilize o programa "telnet", que cria uma conexão TCP e permite escrever/ler dados da conexão. Ex.:

```
$ telnet asc.di.fct.unl.pt 80
> GET  /rc/index.html  HTTP/1.0     pedido
>
HTTP/1.1 200 OK      resposta
. . .
```


Análise de protocolos HTTP e TCP

- Use o Wireshark para analizar as interacções entre um browser e um servidor HTTP
- Siga a folha de exercícios #3

Java Threads

- Os processos podem ter vários fios (ou fluxos) de execução independentes (threads) no mesmo programa.
- Além do thread principal que arranca no método main(), num programa Java pode criar tantos threads auxiliares quantos os necessários, desde que não esgote os recursos da máquina...
- Os threads são representados por objectos da classe java.lang.Thread (ou suas subclasses)

Vida dos Java Threads

- Cada thread tem inicio no método **run()** do objecto
- Os threads terminam quando chegam ao fim do método run() onde arrancaram
- Os threads não se podem matar directamente
- Podemos esperar pelo término dum thread usando o método **join()**

Criação e arranque de Threads Java

- Há várias possibilidades para criar threads. Ex. 1:

```
class myClass extends Thread {
public void run() {...}
}
```

  – Criação: `myClass a = new myClass();`
  – Arranque: `a.start();`
    - o novo thread arranca no método run() de myClass

Criação e arranque de Threads Java

- Ex. 2 :

```
class myClass implements Runnable {
public void run() {...}
}
```

  – Criação: `Thread t = new Thread(new myClass()) ;`
  – Arranque: `t.start();`
    - o novo thread arranca no método run() de myClass ...
- E, ainda, usando classes anónimas...
  – Programação Java avançada...
  – (os curiosos podem estudar o código de exemplo na página da cadeira...)

Comunicação e Threads

- Os threads permitem fazer, em **simultâneo**, várias operações.
  – Por exemplo: leituras, as quais são bloqueantes por natureza.
    - só com o thread principal não é possível estar à espera de uma mensagem num socket (receive) e, ao mesmo tempo, ler da consola (System.in) ou de um ficheiro. Ou bloqueia numa operação ou na outra...
- Criando um thread auxiliar resolve-se esse problema...

Classes para Threads Java

- Pacotes
  – java.lang
- Classes específicas
  – java.lang.Thread → fio de execução independente
  – java.lang.Runnable → **interface** auxiliar para criar Threads dentro de outras classes...

Exemplo de servidor concorrente

```
public class ConcurrentServer {

   public static void main(String args[]) throws Exception {
      ServerSocket serverSocket = new ServerSocket( PORT ) ;

      for(;;) { // ciclo infinito de atendimento
       Socket clientSocket = serverSocket.accept() ;
       Thread t = new ClientHandler(clientSocket);
        t.start();
      }
   }
}

class ClientHandler extends Thread {
   Socket localSock;
   ClientHandler( Socket s ) {
      localSock = s;
   }
   public void run() { // thread auxiliar
      // trata um cliente via this.localSock
      localSock.close();
   }
}
```