Departamento de Engenharia Electrotécnica

REDES INTEGRADAS DE TELECOMUNICAÇÕES II

2015 / 2016

Mestrado Integrado em Engenharia Electrotécnica
e de Computadores

4º ano
8º semestre

1º Trabalho prático:
Servidor HTTP com RESTful API

http://tele1.dee.fct.unl.pt/rit2 **Pedro Amaral / Luis Bernardo**

1. OBJECTIVOS

Familiarização com os protocolos Web e com a programação de aplicações em Java baseadas em *sockets* TCP. O trabalho consiste no desenvolvimento de um servidor Web *dual-stack* multi-tarefa que implementa uma API RESTfull. Este servidor deve realizar os comandos *GET*, *HEAD* e *POST*, e um subconjunto limitado das funcionalidades do protocolo HTTP, permitindo a sua realização num número reduzido de horas. Pretende-se que o servidor Web satisfaça um conjunto de requisitos:

- Funcione tanto para IPv6 como IPv4;
- Seja compatível com HTTP 1.1;
- Controle o número de segundos que mantém uma ligação aberta (em HTTP 1.1);
- Devolva repostas aos pedidos efectuados à API RESTful na forma de páginas HTML geradas dinamicamente;
- Interprete os cabeçalhos HTTP, passando os cabeçalhos relevantes para os objectos que implementam a API quando esta é evocada;
- Interprete os campos dos formulários, enviando-os para os objectos que implementam a API quando esta é evocada.

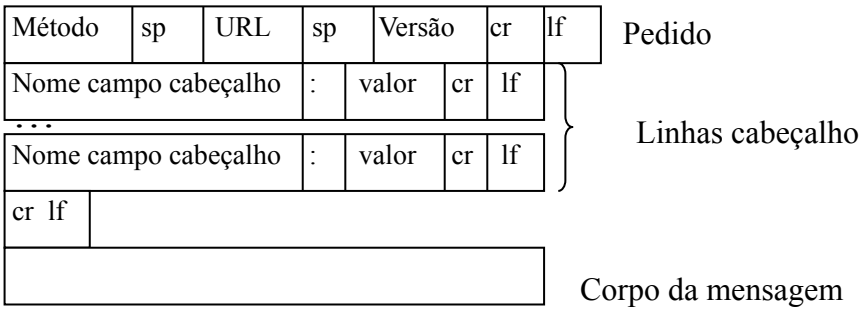
Este trabalho complementa a aprendizagem sobre Java realizada no primeiro trabalho de RIT1, acrescentando novos conhecimentos introduzidos na terceira secção deste documento. Antes, na segunda secção é apresentado um resumo das características do protocolo HTTP relevantes para este trabalho. Na quarta secção é apresentada a especificação completa do servidor. É também apresentada uma descrição de excertos de software que são fornecidos, para facilitar a realização do trabalho.

2. COMPLEMENTOS SOBRE HTTP

Esta secção introduz os aspectos mais relevantes do protocolo HTTP (*HyperText Transfer Protocol*), para a realização do trabalho. Recomenda-se que os alunos consultem mais documentação sobre o protocolo HTTP, no livro teórico recomendado e em [1].

2.1 O protocolo HTTP

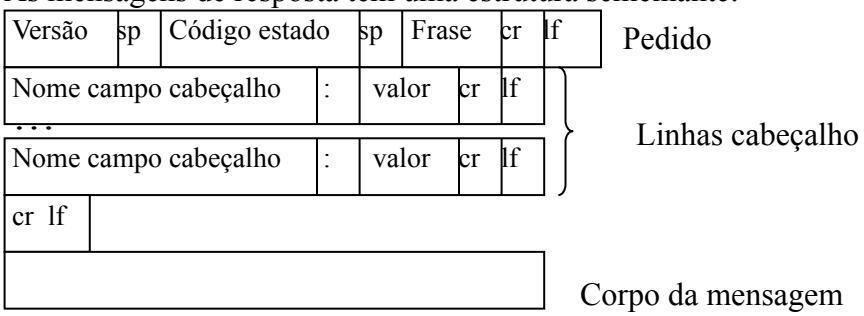
O protocolo HTTP define uma interacção do tipo pedido-resposta entre um cliente e um servidor, onde as mensagens trocadas contêm texto legível. Existem duas versões do protocolo: HTTP 1.0 [2] e HTTP 1.1 [3]. A mensagem de pedido, do cliente para o servidor tem a seguinte estrutura (sp = espaço ; cr lf = mudança de linha). O URL corresponde ao nome de um ficheiro local (num servidor):



Os métodos GET, HEAD e POST têm uma estrutura semelhante. Um pedido pode incluir diversas linhas de cabeçalhos opcionais, que definem a data de acesso ao servidor (*Date*), o nome do servidor acedido (*Host*) (obrigatório quando o servidor HTTP suporta várias máquinas virtuais), o tipo de browser e sistema operativo usado (*User-Agent*), os formatos de dados suportados (*Accept*), a língua pretendida (*Accept-Language*), se o ficheiro foi modificado desde o último acesso (*If-Modified-Since* e *If-None-Match*), para HTTP 1.1 a indicação se pretende manter a ligação aberta (*Connection*) e durante quanto tempo (*Keep-Alive*), o tipo dos dados no corpo da mensagem (*Content-Type*), o número de bytes da mensagem (*Content-Length*), dados de sessão (*Cookie*), etc. Depois, separando o cabeçalho de um corpo de mensagem opcional, existe uma linha em branco ("\r\n"). Os valores dos campos do formulário geralmente vêm nos dados. Um exemplo de pedido (GET) a um servidor será:

```
GET /page.html HTTP/1.1
Host: tele1.dee.fct.unl.pt
User-Agent: RIT2 Proxy Demo
Accept-language: pt-pt;pt
Connection: Keep-Alive
```

As mensagens de resposta têm uma estrutura semelhante:



O campo mais importante da resposta é o código de estado, que define o que se passou com o pedido. Os códigos de estado devolvidos podem ser:

Código	**Tipo**	**Exemplo de razões**
1xx	Informação	Recebeu pedido, continua processamento
2xx	Sucesso	Acção terminada com sucesso
3xx	Redirecção	Necessárias mais acções para completar
4xx	Erro do cliente	Pedido errado, não pode ser executado
5xx	Erro do servidor	Servidor falhou com pedido válido

Alguns exemplos de códigos de estado úteis para o trabalho são:

- **200 OK**: Sucesso - informação retornada no corpo da mensagem
- **301 Moved Permanently**: Moveu-se para URL contido no campo de cabeçalho '*Location*:'
- **304 Not Modified**: Ficheiro não foi modificado
- **400 Bad Request**: Pedido não entendido pelo servidor
- **403 Forbidden**: O ficheiro pedido não pode ser acedido;
- **404 Not found**: O ficheiro pedido não existe ou não pode ser acedido
- **501 Not implemented**: Pedido não suportado pelo servidor

As respostas incluem alguns cabeçalhos semelhantes aos pedidos (*Content-Type*, *Content-Length*, *Connection*, *Date*, etc.), acrescentando outros opcionais específicos, como a indicação da última modificação do ficheiro (*Last-Modified*), do valor de *hash* do ficheiro (*ETag*), a definição de um estado no cliente (*Set-Cookie*), e o controlo de cache (*Cache-Control* em HTTP 1.1 ou *Pragma* em HTTP 1.0). Observe-se que o cabeçalho *Content-Type* é obrigatório sempre que se devolverem dados, indicando o tipo de dados MIME [1], de forma ao browser saber como interpretar os dados. Ficheiros HTML são codificados no formato "*text/html*", enquanto ficheiros de imagens podem ser codificados em "*image/gif*" ou "*image/jpeg*", podendo para efeitos deste trabalho ser enviados os restantes dados como "*application/octet-stream*". No caso de ficheiros HTML, também é importante o cabeçalho *Content-Encoding*, com o formato de codificação. Recomenda-se que seja SEMPRE usado o formato "*ISO-8859-1*" em todos os ficheiros de texto no servidor. Um exemplo de uma resposta de um servidor seria:

```
HTTP/1.1 200 OK
Date: Wed, 27 Feb 2012 12:00:00 GMT
Server: Apache/2.2.8 (Fedora)
Last-Modified: Thu, 28 Set 2003 19:00:00 GMT
ETag: "405990-4ac-4478e4405c6c0"
Content-Length: 6821
Connection: close
Content-Type: text/html; charset=ISO-8859-1

… { dados html } …
```

A principal diferença entre HTTP 1.0 e HTTP 1.1 está no controlo da ligação TCP. No primeiro caso (1.0) a ligação é fechada após cada pedido. No segundo (1.1) podem ser usados os campos de cabeçalho (*Connection* e *Keep-Alive*) no pedido e na resposta para definir se se mantém a ligação TCP aberta e por quanto tempo. Se tanto o pedido como a resposta contiver um campo de cabeçalho **Connection** com um valor diferente de *close*, a ligação TCP não é desligada, podendo ser enviados vários pedidos através dessa ligação. Caso o pedido inclua o campo *Keep-Alive*, ele deverá ser enviado na resposta. Um exemplo de valores um cabeçalho de resposta a manter a ligação durante 60 segundos seria:

```
Connection: keep-alive
Keep-Alive: 60
```

[1] Ver página 635 de [4] com lista de tipos

2.2 *Cookies*

Para se poder ter estados associados a uma ligação HTTP (que por desenho não tem estado), foram definidos dois campos de cabeçalho adicionais numa norma externa [5] ao HTTP. Sempre que um servidor adiciona um campo *Set-Cookie* à mensagem, o browser memoriza o valor recebido e retorna-o nos pedidos seguintes ao mesmo servidor. A sintaxe deste campo é:

```
Set-Cookie: NAME=VALUE; expires=DATE; path=PATH; domain=DOMAIN_NAME; secure
```

- *NAME=VALUE* – define o nome e o valor do *cookie*. É o único campo obrigatório, podendo-se usa a forma simplificada "*Set-Cookie: VALUE*" para um nome vazio.
- *expires=DATE* – define a validade do *cookie*. Se não for definido, o *cookie* desaparece quando se fecha o browser. Pode eliminar-se um *cookie* no browser enviando uma data anterior à data actual.
- *domain=DOMAIN_NAME* – Define o nome do servidor. O mesmo servidor Web poderá suportar vários servidores virtuais, fazendo-se a distinção pelo valor do campo de cabeçalho *Host*. Se não for definido, o *cookie* é enviado baseado no endereço IP.
- *path=PATH* – documento raiz a partir da qual todos os documento pedidos devem levar o mesmo *cookie*. Se não for definido, o *cookie* apenas é enviado para a página pedida.
- *secure* – se seleccionado, o *cookie* apenas é enviado para ligações seguras.

Se um servidor enviar o campo:

```
Set-Cookie: CUSTOMER=WILE_E_COYOTE; path=/; expires=Wednesday, 09-
Nov-99 23:12:40 GMT
```

Recebe nos pedidos seguintes para todos os documentos abaixo de "/":

```
Cookie: CUSTOMER=WILE_E_COYOTE
```

O servidor pode definir vários *cookies* em campos de cabeçalho *Set-Cookie* separados, que são devolvidos concatenados (separados por ';') num único campo de cabeçalho *Cookie* nos pedidos futuros.

2.3 Formulários

Os formulários (*forms*) apareceram com o HTML 2.0 [6] e suportam a definição de campos de entrada de dados, delimitados entre as etiquetas <form> e </form>. Cada campo de um formulário é identificado por um nome (*name*) e tem associado um tipo (*type*) que pertence a um conjunto predefinido de tipos (e.g. *Checkboxes*, *Radio Buttons*, *Text boxes*, *Submit buttons*, etc.), e por um valor inicial (*value*). Por exemplo:

```
<form ACTION="http://tele1.dee.fct.unl.pt/create" method=POST>
Name <input type="text" name ="Finger" size=40></form>
```

Quando se invoca o método POST através de um "*submit button*", ou indirectamente através de um GET com um campo "*?*" num URL, os valores são enviados para o servidor num formato compacto, que concatena todos os campos do formulário numa *string*, delimitados pelo carácter '&'. Cada campo é codificado como "*nome=valor*", onde todos os espaços em branco são substituídos pelo carácter '+', e os caracteres reservados são substituídos pelo valor numérico correspondente em hexadecimal no formato "*%aa*" (no caso do Internet Explorer constata-se

que muitas vezes não segue a norma e substitui os espaços por "*%A0*"). Por exemplo, o seguinte conjunto de campos (representados na forma *nome=valor*)

```
star = Eastwood
cert = 15
movie = Pale Rider
```

seria codificado na cadeia de caracteres "*star=Eastwood&cert=15&movie=Pale+Rider*". A invocação equivalente realizada com um GET seria:

http://tele1.dee.fct.unl.pt/cgi-bin/finger?star=Eastwood&cert=15&movie=Pale+Rider

Vulgarmente, os dados enviados através de um POST são processados num *script* Perl (abordagem vulgarmente designada por CGI (*Common Gateway Interface*)), num *script* PHP, em objectos Java (com *servlets* ou JSP), objectos ASP ou ainda como vamos fazer no segundo trabalho usando Java Script no lado do Servidor. Para evitar a tentação de usar o muito código existente na Internet, neste trabalho define-se mais um formato de interface do servidor Web com um objecto externo para tratar o método POST ao invés de usar classes *servlets*.

Pode-se encontrar mais informação sobre a utilização de *forms* nas páginas 667-670 do livro recomendado na disciplina [4], ou em várias páginas tutoriais existentes na Internet (e.g. [7]).

2.4 Controlo de *caching*

Quando um browser recebe um ficheiro, costuma guardá-lo numa memória local, designada de *cache*. Para que o ficheiro possa ser guardado, ele não deverá ter campos de autenticação ou *cookies*, e não ser uma página dinâmica. O HTTP define vários métodos para controlar se o ficheiro continua actual. Os mais comuns usam os campos de cabeçalho "*If-Modified-Since*" e "*If-None-Match*". No primeiro caso, no segundo pedido, acrescenta-se o cabeçalho com a data da última modificação; no segundo caso o cabeçalho com a assinatura digital do ficheiro (recebida no campo "*ETag*"). Caso o ficheiro seja atual responde-se com o código 304, e com os campos *Date*, *Server*, *ETag*, e os campos de controlo de ligação, caso existam. Caso contrário, deve-se ignorar o campo de cabeçalho, e devolver o novo ficheiro.

```
PEDIDO 1:
GET /somedir/page.html HTTP/1.0
…
RESPOSTA 1:
HTTP/1.0 200 OK
Last-Modified: Thu, 23 Oct 2002 12:00:00 GMT
ETag:"a4137-7ff-42068cd3"
…
PEDIDO 2:
GET /somedir/page.html HTTP/1.0
If-Modified-Since: Thu, 23 Oct 2002 12:00:00 GMT
If-None-Match: "a4137-7ff-42068cd3"
…
RESPOSTA 2:
HTTP/1.0 304 Not Modified
Date: Thu, 23 Oct 2002 12:35:00 GMT
```

Outros métodos consistem na utilização de campos de controlo de *caching*. Em HTTP 1.0 pode ser retornado o campo de cabeçalho "*Pragma: no-cache*" tanto nos pedidos (não usar como resposta valores em cache) como nas respostas (não guardar em cache). Em HTTP 1.1

deve ser usado o campo de cabeçalho "*Cache-control*" que pode ter um subconjunto de vários valores possíveis. O valor "*no-cache*" tem o mesmo significado que anteriormente. O valor "*no-store*" no pedido ou resposta significa que a resposta não pode nunca ser guardada em cache. O valor "*max-age=n*" define o tempo máximo de vida da resposta, ou o tempo máximo que o ficheiro pode estar em cache para ser válido.

3. COMPLEMENTOS SOBRE JAVA

Nesta secção admite-se que os alunos já estão familiarizados com o desenvolvimento de aplicações utilizando NetBeans e sockets UDP usados durante trabalhos das disciplinas precedentes, no site da disciplina podem encontrar um documento de uma dessas cadeiras anteriores que pode ser útil para os que não tiverem ainda essa familiarização. Este trabalho vai ser desenvolvido no ambiente (NetBeans), que é distribuído juntamente com o ambiente Java.

3.1. IPv6

O protocolo IPv6 é suportado de uma forma transparente na linguagem Java, a partir da versão Java 1.4. A classe *InetAddress* tanto permite lidar com endereços IPv4 como IPv6. Esta classe tem como subclasses *Inet4Address* e *Inet6Address*, que suportam respectivamente as funções específicas relativas aos endereços IPv4 e IPv6. Esta arquitectura permite suportar endereços IPv6 de uma forma transparente nas restantes classes do pacote *java.net*, uma vez que as restantes funções usam parâmetros da classe *InetAddress* nos argumentos e nas variáveis dos objectos das classes.

É possível obter o endereço local da máquina IPv4 ou IPv6 usando a função da classe `getLocalHost`:

```
try {
    InetAddress addr = InetAddress.getLocalHost();
} catch (UnknownHostException e) {
    // tratar excepção
}
```

O método `getHostAddress` permite obter o endereço em formato *string*. O nome associado ao endereço pode ser obtido com o método `getHostName`.

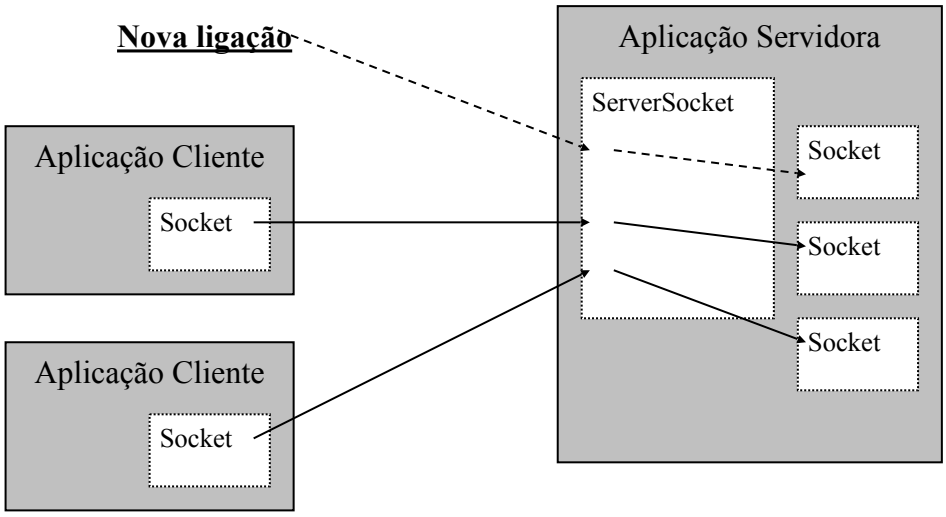
De forma semelhante, os métodos `InetAddress.getByName` ou `InetAddress.getAllByName` permitem obter os endereços (IPv4 ou IPv6) associados a um nome de domínio.

3.2 Sockets TCP

Em Java, a interface para sockets TCP é realizada através de duas classes: *ServerSocket* e *Socket*.

3.2.1 A classe ServerSocket

A classe *ServerSocket* define um objecto servidor que pode receber e manter várias ligações abertas. Quando se cria um objecto, define-se o porto onde ele vai escutar. O método *accept()* bloqueia o objecto até que seja recebida uma ligação ao porto, nessa altura é criado um objecto da classe *Socket* que permite comunicar com o cliente.



Os construtores da classe *ServerSocket* permitem definir o porto, o número de novas ligações pendentes que são aceites pelo objecto e o endereço IP (a interface) a que se faz a associação. Existe ainda um quarto construtor sem parâmetros, que não inicializa o porto, permitindo usar a função *setReuseAddress()*, antes de definir o porto com a função *bind()*.

```
// Public Constructors
public ServerSocket(int port) throws IOException;
public ServerSocket(int port, int backlog) throws IOException;
public ServerSocket(int port, int backlog, InetAddress bindAddr) throws
     IOException;
public ServerSocket() throws IOException;
```

As duas funções principais permitem receber ligações e terminar o servidor.

```
public Socket accept() throws IOException;
public void close() throws IOException;
```

Outras funções permitem ter acesso a parâmetros e configurações do socket:

```
public InetAddress getInetAddress();
public int getLocalPort();
public synchronized int getSoTimeout() throws IOException;
public synchronized void setSoTimeout(int timeout) throws
      SocketException;
```

Para permitir comunicar com os clientes e aceitar novas ligações em paralelo é comum usar múltiplas tarefas (objectos do tipo *thread*) para lidar com cada ligação.

3.2.2 A classe Socket

A classe *Socket* define um objecto de intercomunicação em modo feixe. Pode ser criado através de um construtor ou a partir da operação *accept*. O construtor permite programar

clientes: especifica-se o endereço IP e porto a que se pretende ligar, e o construtor estabelece a ligação.

```
public Socket(String host, int port) throws UnknownHostException,
     IOException;
public Socket(InetAddress address, int port) throws IOException;
public Socket(String host, int port, InetAddress localAddr,
     int localPort) throws IOException;
public Socket(InetAddress address, int port, InetAddress localAddr,
     int localPort) throws IOException;
```

As operações de escrita e leitura do socket são realizadas através de objectos do pacote *java.io* (*InputStream* e *OutputStream*), descritos na próxima secção, retornados por duas funções da classe. Existem ainda funções para fechar o socket e para obter informações sobre a identidade da ligação.

```
public InputStream getInputStream() throws IOException;
public OutputStream getOutputStream() throws IOException;
public synchronized void close() throws IOException;
public InetAddress getInetAddress();
public InetAddress getLocalAddress();
public int getLocalPort();
public int getPort();
public isConnected();
public isClosed();
```

Várias operações de configuração dos parâmetros do protocolo TCP podem ser realizadas através de métodos desta classe. As funções *getReceiveBufferSize*, *setReceiveBufferSize*, *getSendBufferSize* e *setSendBufferSize* permitem modificar a dimensão dos buffers usados no protocolo TCP. As funções *getTCPNoDelay* e *setTCPNoDelay* controlam a utilização do algoritmo de Nagle (*false* = desligado). As funções *getSoLinger* e *setSoLinger* controlam o que acontece quando se fecha a ligação: se está ligada o valor define o número de segundos que se espera até tentar enviar o resto dos dados que estão no buffer TCP e ainda não foram enviados. Caso esteja activo, pode originar perda de dados não detectável pela aplicação.

```
public int getReceiveBufferSize() throws SocketException;
public synchronized void setReceiveBufferSize(int size) throws
     SocketException;
public int getSendBufferSize() throws SocketException;
public synchronized void setSendBufferSize(int size) throws
      SocketException;
public boolean getTcpNoDelay() throws SocketException;
public void setTcpNoDelay(boolean on) throws SocketException;
public int getSoLinger() throws SocketException;
public void setSoLinger(boolean on, int val) throws SocketException;
public synchronized int getSoTimeout() throws SocketException;
public synchronized void setSoTimeout (int timeout) throws
     SocketException;
```

As funções *getSoTimeout* e *setSoTimeout* permitem configurar o tempo máximo que uma operação de leitura pode ficar bloqueada, antes de ser cancelada. Caso o tempo expire é gerada uma excepção *SocketTimeoutException*. Estas funções também existem para as classes ServerSocket e DatagramSocket.

3.2.3 Comunicação em sockets TCP

Os objectos da classe *Socket* oferecem métodos para obter um objecto da classe *InputStream* (*getInputStream*) para ler do socket, e para obter um objecto da classe *OutputStream* (*getOutputStream*). No entanto, estas classes apenas suportam a escrita de *arrays* de bytes. Assim, é comum usar outras classes do pacote *java.io* para envolver estas classes base, obtendo-se uma maior flexibilidade:

Para ler *strings* a partir de um socket é possível trabalhar com:

- A classe *InputStreamReader* processa a cadeia de bytes interpretando-a como uma sequência caracteres (convertendo o formato de carácter).
- A classe *BufferedReader* armazena os caracteres recebidos a partir de um feixe do tipo *InputStreamReader*, suportando o método *readLine()* para esperar pela recepção de uma linha completa.

Para escrever strings num socket é possível trabalhar com:

- A classe *OutputStreamWriter* processa a cadeia de caracteres e codifica-a para uma sequência de bytes.
- A classe *PrintWriter* suporta os métodos de escrita de *strings* e de variáveis de outros formatos (*print* e *println*) para um feixe do tipo *OutputStreamWriter*.
- A classe *PrintStream* suporta os métodos de escrita de *strings* e de *byte []* para um feixe do tipo *OutputStream*.

Caso se pretendesse enviar objectos num formato binário (não legível), dever-se-ia usar as classes *DataInputStream* e *DataOutputStream*.

Um exemplo de utilização destas classes seria o apresentado em seguida:

```
try {          // soc representa uma variável do tipo Socket inicializada
     // Cria feixe de leitura
   InputStream ins = soc.getInputStream( );
   BufferedReader in = new BufferedReader(
      new InputStreamReader(ins, "8859_1" ));  // Tipo de caracter
      // Cria feixe de escrita
   OutputStream out = soc.getOutputStream( );
   PrintStream pout = new PrintStream(out);
   // Em alternativa poder-se-ia usar PrintWriter:
   // PrintWriter pout = new PrintWriter(
   //        new OutputStreamWriter(out, "8859_1"), true);
      // Lê linha e ecoa-a  para a saída
   String in_string= in.readln();
   pout.writeln("Recebi: "+ in_string);
}
catch (IOException e ) { ... }
```

3.2.4 Exemplo de aplicação - TinyFileServ

No exemplo seguinte é apresentado um servidor de ficheiros parcialmente compatível com o protocolo HTTP (utilizável por browsers), que recebe o nome do ficheiro no formato (* *nome-completo* *) e devolve o conteúdo do ficheiro, fechando a ligação após o envio do último caracter do ficheiro. O servidor recebe como argumentos da linha de comando o número de porto e o directório raiz correspondente à directoria "/" (e.g. *"java TinyFileServ 20000 /home/rit2/www"*). Para cada nova ligação, o servidor lança uma tarefa que envia o ficheiro pedido e termina após o envio do ficheiro. O servidor é compatível com um browser: lê o segundo campo do pedido (e.g. GET / HTTP/1.1). Não é enviado o código de resposta, mas os browsers mais

vulgares (IE, Firefox, etc.) sabem interpretar o ficheiro recebido a partir dos primeiros caracteres e do nome do ficheiro. Este exemplo usa algumas classes descritas nas próximas secções deste capítulo.

```
//file: TinyFileServd.java
import java.net.*;
import java.io.*;
import java.util.*;

public class TinyFileServd {
    public static void main( String argv[] ) throws IOException {
        ServerSocket ss = new ServerSocket(
            Integer.parseInt(argv.length>0 ? argv[0] : "20000"));
        while ( true )
            new TinyFileConnection(ss.accept(),
                                   (argv.length>1 ? argv[1] : "")).start( );
    }
} // end of class TinyFileServd

class TinyFileConnection extends Thread {
    private Socket client;
    private String root;
    TinyFileConnection ( Socket client, String root ) throws SocketException {
        this.client = client;
        this.root= root;
        setPriority( NORM_PRIORITY + 1 );  // Higher the thread priority
    }

    public void run( ) {
        try {
            BufferedReader in = new BufferedReader(
                new InputStreamReader(client.getInputStream( ), "8859_1" ));
            OutputStream out = client.getOutputStream( );
            PrintStream pout = new PrintStream(out, false, "8859_1");
            String request = in.readLine( );      // Reads the first line
            System.out.println( "Request: "+request );
            StringTokenizer st= new StringTokenizer (request);
            if (st.countTokens() != 3) return;  // Invalid request
            String code= st.nextToken();    // USES HTTP syntax
            String file= st.nextToken();    // for requesting files
            String ver= st.nextToken();
            String filename= root+file+(file.equals("/")?"index.htm":"");
            System.out.println("Filename= "+filename);
            FileInputStream fis = new FileInputStream ( filename );
            byte [] data = new byte [fis.available()]; // Fails for large files!
            fis.read( data );   // Read from file
            out.write( data );  // Write to socket
            out.flush( );       // Flush socket buffer
            fis.close();
        }
        catch ( FileNotFoundException e ) {
            System.out.println( "File not found" );
        }
        catch ( IOException e ) {
            System.out.println( "I/O error " + e );
        }
        finally {   // Always closes the socket
            try {
                client.close( );
            } catch (Exception e) { /* Ignore everything */ }
        }
    }
} // end of class TinyFileConnection
```

3.3 Strings

Uma vez que o protocolo HTTP é baseado em *strings*, para realizar o servidor web vai ser necessário interpretar comandos e gerar respostas em formato ASCII. Nesta secção são apresentadas algumas das classes que podem ser usadas para desempenhar estas tarefas.

3.3.1. Separação em componentes

Um dos aspectos essenciais é a separação de uma string em componentes fundamentais. As classes *String* e *StringBuilder* oferecem o método *substring* para seleccionar uma parte da *string*:

```
String a= "Java is great";
String b= a.substring(5);          // b is the string "is great"
String c= a.substring(0, 4);       // c is the string "Java"
```

O método *trim()* remove os espaços e tabulações antes e depois da string.
A comparação de *strings* pode ser feita com os métodos *equals* ou *equalsIgnoreCase*.

A linguagem Java oferece a classe *StringTokenizer* para decompor uma string em substrings separadas por espaços ou outros separadores. O método *nextToken()* permite percorrer todos os componentes, enquanto os métodos *hasMoreTokens()* e *countTokens()* permitem saber quando se termina. Por exemplo, o código seguinte permite descodificar a primeira linha de um cabeçalho HTTP nos vários componentes. O construtor pode ter um segundo parâmetro opcional com uma string com a lista de separadores (por omissão tem apenas o espaço por separador).

```
StringTokenizer st = new StringTokenizer( request );
            // Pode-se acrescentar um parâmetro extra com o separador e.g. ":"
if ( (st.countTokens( ) != 3)  { … erro … }
String rqCode= st.nextToken( );
String rqName= st.nextToken( );
String rqVersion = st.nextToken( );
if (rqCode.equals("GET") || rqCode.equals("HEAD")|| rqCode.equals("POST")) {
  if ( rqName.startsWith("/") )
      rqName = rqName.substring( 1 );             // Removes first character
  if ( rqName.endsWith("/") || rqName.equals("") )
      rqName = rqName + "index.htm";              // Adds "index.htm" if is a directory
  …
}
```

3.3.2. Datas compatíveis com o protocolo HTTP

O protocolo HTTP define um formato específico para a escrita de datas:

```
    Thu, 23 Oct 2002 12:00:00 GMT
```

É possível escrever e ler variáveis com este formato utilizando uma objecto da classe *DateFormat*.

```
DateFormat httpformat=
    new SimpleDateFormat ("EE, d MMM yyyy HH:mm:ss zz", Locale.UK);
httpformat.setTimeZone(TimeZone.getTimeZone("GMT"));
// Escrita de datas
out.println("A data actual é " + httpformat.format(dNow));
// Leitura de datas
try {
    Date dNow= httpformat.parse(str);
} catch (ParseException e) {
    System.out.println("Data inválida: " + e + "\n");
}
```

É possível comparar datas utilizando o método *compareTo* de objectos da classe *Date*. A adição e subtracção de intervalos de tempo a datas também são possíveis convertendo a data em *long* utilizando o método *getTime*. Como o valor conta o número de milisegundos desde 1970, basta somar ou subtrair o valor correspondente ao intervalo. Por exemplo, para avançar um dia seria:

```
Date d= new Date(dNow.getTime() + (long)24*60*60*1000);
```

Outra alternativa é usar a função *add* da classe *Calendar* para realizar a operação:

```
Calendar now= Calendar.getInstance();
now.add(Calendar.DAY_OF_YEAR, +1);
Date d= now.getTime();
```

3.3.3. URLs

A leitura e validação de URLs podem ser feitas usando a classe *java.net.URL*. Um URL tem a estrutura seguinte:

```
<protocolo>://<autoridade><path>?<query>#<fragment>
```

Esta classe tem vários construtores que recebem uma *string* com o URL completo e outro que recebe o url por parâmetros. Depois inclui funções que permitem obter os vários campos do URL:

```
// Construtores - devolvem excepção se url inválido:
public URL(String spec) throws MalformedURLException;
public URL(String protocol, String host, int port, String file)
                        throws MalformedURLException

// Exemplo de métodos desta classe:
URL url= new URL("http://lflb@tele1.dee.fct.unl.pt:8080/servlet/xxx?xpto=ola&xa=xa#2");
url.getProtocol() == "http"
url.getAuthority() == " lflb@tele1.dee.fct.unl.pt:8080"  // "" se omitido
url.getUserInfo() == "lflb"                              // null se omitido
url.getPort() == 8080                                    // -1 se omitido
url.getDefaultPort() == 80
url.getFile() == "/servlet/xxx?xpto=ola&xa=xa"           // "" se omitido
url.getHost() == "tele1.dee.fct.unl.pt"                  // "" se omitido
url.getPath() == "/servlet/xxx"                          // "" se omitido
url.getQuery() == "xpto=ola&xa=xa"                       // null se omitido
url.getRef() == "2"                                      // null se omitido
```

3.4 Ficheiros

A classe *java.io.File* representa um ficheiro ou uma directoria, e define um conjunto de métodos para os manipular. O construtor recebe o nome completo de um ficheiro, permitindo depois a classe saber se o ficheiro existe, se é ficheiro ou directoria, o comprimento do ficheiro, apagar o ficheiro, ou marcar o ficheiro para ser apagado quando o programa termina. Inclui ainda métodos para criar ficheiros temporários com nomes únicos.

```
File f= new File ("/home/pc40/xpto.txt"); // Associa-se a ficheiro
long len= f.length();                     // Comprimento do ficheiro
Date date= new Date(f.lastModified());    // Última modificação
if (f.exists()) …                         // Se existe
if (f.isFile()) …                         // Se é ficheiro
if (f.canRead()) …                        // Se é legível
f.delete();                               // Apaga ficheiro
File temp= File.createTempFile("proxy", ".tmp"); // Cria ficheiro temporário com nome
      único
temp.deleteOnExit();                      // Apaga ficheiro quando a aplicação termina
File.separator                            // '/' ou '\\' dependendo do sistema operativo
```

A leitura a partir de ficheiros de texto é geralmente realizada através da classe *FileInputStream*. Recomenda-se que a escrita de ficheiros recebido a partir de servidores Web seja feita através da classe *BufferedWriter*, por esta também permitir definir o tipo de caracteres ("ISO-8859-1" por omissão). Caso este tipo seja usado nos canais associados a sockets e a ficheiros, o Java nunca faz conversão de tipos, permitindo transmitir dados arbitrários (imagens, aplicações, etc.).

```
// Para leitura
FileInputStream f= new FileInputStream ( file );
// Para escrita
FileOutputStream fos= new FileOutputStream(file);
OutputStreamWriter osr= new OutputStreamWriter(fos, "8859_1");
BufferedWriter os= new BufferedWriter(osr);
// Permitem ler e escrever 'char []' com os métodos 'read' e 'write'
// usando o método 'getBytes()' é possível converter um 'char []' em 'byte []'
```

3.5 Estruturas de dados adicionais

A linguagem Java suporta um conjunto de variado de estruturas de dados que permitem lidar de uma forma eficaz com conjuntos de pares (nome de propriedade, valor de propriedade), onde o campo nome_de propriedade é **único**. Uma das estruturas que permite lidar com este tipo de dados é a classe *Properties*. Esta classe é usada para manter as variáveis de sistema. Uma variável da classe *Properties* pode ser iniciada vazia ou a partir do conteúdo de um ficheiro de texto, pode-se acrescentar ou remover elementos, pode-se pesquisar por nome de propriedade ou exaustivamente, e pode-se exportar o conteúdo para um ficheiro.

```
Properties prop= new Properties();
try { prop.load(in) } catch (IOException e) {…}          // Lê ficheiro
prop.setProperty("nome", "valor");                       // define valor
String val= prop.getProperty("nome");                    // obtém valor
String val2= prop.getProperty("nome2", "omisso");        // obtém valor, se n existe
                                                            devolve "omisso"
for (Enumeration p= prop.propertyNames(); p.hasMoreElements();)  // percorre lista
      System.out.println(p.nextElement());
try { prop.store(new FileOutputStream("file"), "Configuração:"); }  // Grava ficheiro
catch (IOException e) { … }
```

3.6 Carregamento de classes em tempo real

A linguagem Java permite que novas classes sejam acrescentadas em tempo real a um programa activo, e que novos objectos dessas classes sejam usados. Uma das maneiras mais simples de lidar com os novos objectos é fazer com que eles herdem os métodos de uma interface ou classe genérica, usando-se os novos objectos através dessa interface.

O código seguinte exemplifica esta utilização com uma classe abstracta pura '*Cooklet*' e uma classe definida "*DemoCooklet*", que implementa todos os métodos indefinidos na classe raiz. A classe '*Cooklet*' define todos os métodos disponíveis para lidar com novos objectos e o comportamento por omissão de cada um dos métodos.

```
public abstract class Cooklet {
    /** Initialization method */
    public void initialize() { /* Default initialization */ }
    /** Work method */
    public abstract void work();
    /** Termination method */
    public void terminate() { /* Default termination */ }
}
```

A classe '*DemoCooklet*' herda da classe '*Cooklet*' estes comportamentos, e redefine todos os métodos específicos.

```
public class DemoCooklet extends Cooklet {
    /** Creates a new instance of Cooklet */
    public DemoCooklet() {
    }
    /** Initialization method */
    public void initialize() {
        System.out.println("DemoCooklet initialized");
    }
    /** Work method */
    public void work() {
        System.out.println("DemoCooklet working");
    }
    /** Termination method */
    public void terminate() {
        System.out.println("DemoCooklet shut down");
    }
}
```

O carregamento de uma nova classe é realizado através do método *Class.forName*, que devolve um objecto do tipo *Class*. O ficheiro "*name.java*" contêm o código da class a carregar e deverá estar acessível no *CLASSPATH*. O método *newInstance* cria um novo objecto do tipo 'nome', que é acedido através dos métodos definidos na classe abstracta *Cooklet*.

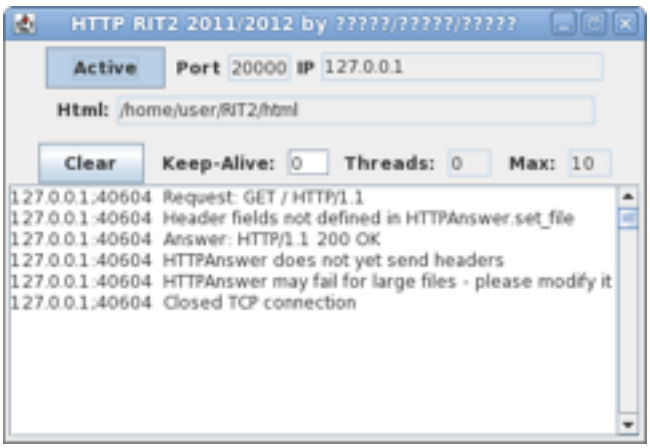
```
private void load_class(String name) {
    Cooklet cooklet = null;
    try {
        Class demoClass= Class.forName("DemoCooklet");   //Loads class 'DemoCooklet'
        Object demoObject= demoClass.newInstance();      // Creates new object
        cooklet= (Cooklet)demoObject;
    }
    catch (Exception e) {
        System.err.println ("Error loading class:"+e);
    }
    if (cooklet == null) {
        Log("null object");
        return;
    }
    cooklet.initialize();
    cooklet.work();
    cooklet.terminate();
}
```

4. ESPECIFICAÇÕES

Pretende-se neste trabalho realizar um servidor web que permita explorar várias funcionalidades da versão 1.1 do protocolo HTTP. Não se pretende uma realização completa do protocolo, mas uma realização "à medida", com uma interface gráfica que permita configurar o servidor de uma forma simples.

4.1 Especificação detalhada

O servidor HTTP deve permitir suportar vários clientes em paralelo, recorrendo a tarefas individuais para responder a cada cliente. Deve ainda manter sempre uma tarefa disponível para receber os eventos gráficos. Na interface gráfica deverá permitir arrancar e parar o servidor, definir o número de porto do servidor, e o número máximo de clientes que pode ter em paralelo. Para facilitar o desenvolvimento, é fornecido o código inicial do servidor web, inspirado no exemplo 3.2.4 (pág. 10), integrado com a interface gráfica representada abaixo. O servidor está funcional para pedidos simples embora, como se pode ver, há várias funcionalidades que foram removidas:



O servidor deve receber pedidos no porto **Port** desde que o *toggleButton* **Active** esteja seleccionado. O campo **Html** define a directoria raiz, a partir de onde se procuram os ficheiros nos acessos ao servidor Web local. URLs em métodos POST são tratados como pedidos para objectos da classe "*nome*" (que trata a parte dinâmica a partir do parâmetros recebidos no POST), não sendo lidos da directoria anterior.

O valor de **Keep-Alive** apenas é usado caso se mantenha uma ligação aberta, em HTTP 1.1. Keep-Alive define o número de segundo máximo de inactividade até que se desligue a ligação (**0** significa que não se desliga).

Threads indica o número de ligações de clientes activas num instante, e **Max** define o número máximo de ligações que podem estar activas. Note que sempre que o número máximo de ligações é atingido, deve ser sempre desligada a ligação TCP (não usando portanto o Keep-alive) para não bloquear o servidor.

Finalmente, o botão **Clear** limpa a janela de texto.

4.2 Envio de respostas a pedidos HTTP

A arquitetura do exemplo 3.2.4 (pág. 10) foi modificada de maneira a poder ter um ciclo principal que pode correr várias funções em paralelo para diferentes tipos de pedidos. Quando as funções de tratamento de pedido terminam, devolvem a resposta HTTP encapsulada num objecto, que depois é enviado para o browser; i.e., o envio da resposta para o browser é feito num local diferente do preenchimento dos dados da resposta.

A classe ***HTTPReplyCode*** é usada para memorizar o conteúdo do código de resposta e a versão do protocolo HTTP usada. Suporta um conjunto de constantes com os códigos de estado da resposta que podem ser usados no trabalho, mais funções para preencher e ler os códigos, e para obter uma descrição textual de cada código.

```
public class HTTPReplyCode {
    public static final int NOTDEFINED= -1;
    public static final int OK= 200;
    public static final int NOTMODIFIED= 304;
    public static final int TMPREDIRECT= 307;
    public static final int BADREQ= 400;
    public static final int UNAUTHORIZED= 401;
    public static final int NOTFOUND= 404;
    public static final int PROXYAUTHENTIC=407;
    public static final int NOTIMPLEMENTED= 501;

    private int code;                                   // HTTP Answer Code
    private String code_txt;                            // HTTP Answer Code Description
    private String version;                             // HTTP Version

    /** Default constructor */
    public HTTPReplyCode(int code, String version);
    public HTTPReplyCode();                             // Code NOTDEFINED, HTTP/1.1
    public HTTPReplyCode(HTTPReplyCode src);            // Copy constructor

    public int get_code();
    public String get_code_txt();
    public String get_version();

    public void set_code(int code);             // Sets default answer code description
    public void set_code_txt(String code_txt);  // Overwrites default code description
    public void set_version(String version);
```

```
    public boolean isError();                           // true if it is an error code
    public boolean isUndef();                           // true if it is NOTDEFINED

    public String toString();            // Returns a string with the 1st header line

    public static String code_text (int code); // Auxiliary function with default code text
}
```

As respostas a pedidos HTTP são encapsuladas em objectos da classe ***HTTPAnswer***. Esta classe memoriza os dados da primeira linha da resposta (*code*), do conjunto de operações *Set-Cookie* (em *set_cookies*) e dos restantes campos de cabeçalho (*param*), e dos dados a enviar. Os dados podem ser guardados na forma de uma string (*text*) ou de um descritor de ficheiro (*file*).

Para preparar uma resposta, é criado um objecto do tipo *HTTPAnswer* que é passado para as funções que preparam a resposta. Depois, estas funções usam os métodos *set_code*, *set_version*, *set_property* e *set_cookie* respectivamente para definir o código da resposta, a versão de HTTP, acrescentar um campo de cabeçalho genérico e acrescentar um *Set-Cookie*. Utilizam ainda os métodos *set_file* ou *set_text*, respectivamente quando preparam uma *string* ou um ficheiro para enviar para o browser. Existem dois métodos adicionais para preencher conteúdos de respostas: o método *set_error* prepara uma resposta com um código de erro e o método *set_Date* preenche o campo de cabeçalho *Date*.

A classe *HTTPAnswer* define ainda métodos para aceder aos campos da resposta, permitindo obter o código da resposta (*get_code*), uma string com a primeira linha (*get_first_line*), um iterador sobre os campos de cabeçalho (*get_Iterator_parameter_names*) e a lista de *Set-Cookies*.

Por fim, define o método *send_Answer*, que é usado para enviar a resposta para o browser, que pode ou não enviar os dados, dependendo do valor do parâmetro *send_data*. O código fornecido com o trabalho inclui uma realização parcial das duas últimas classes apresentadas.

```
public class HTTPAnswer {

    private HTTPReplyCode code;           // Reply code information
/**  Reply headers data   */
    private Properties param;             // Header fields
    public ArrayList<String> set_cookies; // Set cookies header fields
/**  Reply contents  */
    private String text;                  // buffer with reply contents
    private File file;                    // used if text == null
    Log log;                              // Log object
    String id_str;                        // Thread id - for logging purposes

    /** Constructors */
    public HTTPAnswer(Log log, String id_str, String server_name);
    public HTTPAnswer(HTTPAnswer a);

    void Log(boolean in_window, String s);

    public void set_code(int _code);
    public void set_version(String v);
    public void set_property(String name, String value);
    public void set_cookie(String setcookie_line);
    public void set_file(File _f, String mime_enc);
    public void set_text(String _text);

    public void set_Date();
    public void set_error(int _code, String version);
```

```
    public int get_code();
    public String get_first_line();
    public Iterator<Object> get_Iterator_parameter_names();
    public ArrayList<String> get_set_cookies();

    /** Sends the HTTP reply to the client using 'pout' text device */
       public void send_Answer(PrintStream pout, boolean send_data, boolean echo) throws
      IOException;
}
```

4.3 Interface de programação para objectos JavaREST que implementam a RESTful API.

Qualquer classe que realize um objecto JavaREST (*Classe que tratará os pedidos à API*) deve estender a classe ***JavaREST***, seguindo-se a abordagem apresentada na secção 3.6. A classe *JavaREST* define os métodos representados na figura seguinte.

```
public abstract class JavaREST {
    /** Runs GET method  ---  By default returns "not supported" */
    public boolean doGet(Socket s, Properties param, Properties cookies, HTTPAnswer ans);

    /** Runs POST method  ---  By default returns "not supported" */
       public boolean doPost(Socket s, Properties param, Properties cookies, Properties
      fields, HTTPAnswer ans);

    /** Converts POST string into Java string (ISO-8859-1) (removes formating codes) */
    public static String postString2string(String in_s);

    /** Converts java string (ISO-8859-1) to HTML format */
    public static String String2htmlString(String in_s);

    /** Convert POST string (ISO-8859-1) to HTML format */
    public static String postString2htmlString(String in_s);
}
```

O método ***doGet*** é usado sempre que é invocada uma operação GET sobre um URI que seja referente à API. Este método recebe como argumentos o socket da ligação TCP (***s***), a lista de cabeçalhos recebidos no pedido HTTP excluindo os *cookies* (***param***), a lista de cookies recebidos no pedido http (***cookies***) e o objecto *HTTPAnswer* onde vai ser escrita a resposta (***ans***). Por omissão o método retorna uma página com o código 501 (não implementado), mas o método pode ser redefinido nas classes que a estendem.

O método ***doPost*** é usado sempre que é invocada uma operação POST sobre um URI que seja referente à API. Este método tem quase todos os argumentos semelhantes ao método *doGet*, excepto os dados (parâmetros do formulário com a estrutura representada na página 5) que são passados sobre a forma de uma lista (***fields***).

A classe *JavaREST* define ainda um conjunto de funções para conversão do formato de *strings*.

O código do servidor HTTP deve ler integralmente todos os campos dos pedidos HTTP, e criar os objectos *Properties* que passa para o objecto *JavaREST*. Por sua vez, as funções devem escrever o conteúdo da página HTML dinâmica criada com resposta e os campos de cabeçalho no objecto *ans*.

4.4 Aplicação rit2REST

Pretende-se que seja desenvolvida uma aplicação de manipulação de uma base de dados de grupos de alunos. Para esse efeito é fornecida a classe *groupDB* que suporta uma base de dados simplificada.

```
public class groupDB {
    private String bdname;
    private static Properties pgroups= new Properties(); // Group data
    private static SortedSet<String> set = new TreeSet<String>();  // Sorted set of names

    public groupDB (String _bdname);              // Constructor, reads contents from file
    public void save_group_db();                  // Saves group database to file
    public SortedSet<String> get_sorted_set();    // Returns sorted list with names

    public String table_group_html();             // Returns a HTML table with the group
    public String get_group_info(String group, String prop);     // Get group information

    public void store_group(String group, boolean contar, String n1, String nam1,
            String n2, String nam2, String n3, String nam3); // Stores a group information
    public void remove_group(String group);       // Removes a group from the database
}
```

A classe *groupDB* mantém uma lista ordenada de grupos, onde para cada grupo memoriza o nome do grupo, o número e nome de até três alunos, um contador de modificações, e uma data. Para isso, memoriza cada um dos campos em *pgroups* com uma chave única[2]. A classe permite escrever o conteúdo da base de dados (BD) para ficheiro (com *save_group_db*) e ler o conteúdo, a partir do ficheiro com nome *bdname* no construtor. Permite ainda obter uma lista ordenada de nomes (*get_sorted_set*), o conteúdo na forma de tabela HTML (*table_group_html*) e um dos campos memorizados para um grupo (*get_group_info*).

A introdução e remoção de grupos é feita com as funções *store_group* e *remove_group*. É fornecida a classe *test_DB* com uma interface gráfica local para a classe *groupDB*, que ilustra como a classe pode ser usada para gerir a base de dados de grupos de alunos.

A interface web para aceder à base de dados vai utilizar a RESTful API do servidor que vai ser implementada na classe ***rit2REST***, que devolve uma página dinâmica de teste que permite manter uma lista de grupos, com os membros de cada grupo, e conta o número de actualizações para cada grupo. A classe suporta os três métodos (GET/HEAD e POST), gerando a página HTML com o formulário que permite a introdução e remoção de grupos.

É fornecido aos alunos o ficheiro ***rit2REST.java*** que suporta todas as funcionalidades de criação da página HTML, faltando programar o tratamento das funções do POST e o tratamento dos cookies (definir o formato dos cookies e programar a geração de *Set-Cookie*) de forma a que:

- Um browser que inscreva um grupo na página /*rit2REST*, veja os dados do grupo sempre que tornar a visitar a página (GET) no futuro;
- Que o browser pare de visualizar a página apenas quando apagar o grupo;
- Que sempre que o contador esteja seleccionado, conte mais uma actualização e apresente o número de vezes que o grupo foi actualizado;
- Apresente a data da última actualização;
- A sessão HTTP deve-se poder manter aberta durante os comandos POST.

[2] Num sistema profissional, seria usada uma base de dados para realizar esta funcionalidade. No segundo trabalho usaremos uma.

4.5 Testes

O servidor web deverá ser compatível com qualquer browser, com qualquer das opções ligadas. Assim, recomenda-se que sigam as especificações indicadas na norma HTTP e evitem adoptar configurações específicas de algum browser. Sempre que se seleccione ou cancele uma opção, o servidor deve cancelar os efeitos da configuração anterior, sempre que possível (e.g. arrancar e parar o servidor).

No mínimo, o servidor deverá suportar pedidos (GET/HEAD) de ficheiros normais **mantendo a sessão aberta** sempre que o browser o indicar. Deve também ler todos os campos de cabeçalho e interpretar no mínimo os seguintes cabeçalhos (*Connection*, *Keep-Alive*, *If-Modified-Since*). O servidor deverá sempre devolver os campos *Date*, *Server*, *Last modified*, *Content-Type*, *Content-Length*, e *Content-Encoding*.

Para ambicionarem a uma boa nota, dever-se-á também suportar o método POST e desenvolver a interacção com páginas geradas pela RESTful API.

Chama-se a atenção para os problemas de **segurança** que um servidor web poderá representar para o conteúdo da vossa área. Assim, recomenda-se que se devolva um erro sempre que:

- for pedido o conteúdo de um directório
- o nome do ficheiro incluir *“..”*
- o nome do ficheiro incluir *“.java”*

Atendendo à grande difusão de realizações de servidores e procuradores Web disponíveis na Internet, alerta-se os alunos que não devem usar código que não conheçam ou consigam interpretar, pois a discussão vai focar todos os aspectos do código e da realização do servidor.

4.6 Desenvolvimento do trabalho

O trabalho vai ser desenvolvido em cinco semanas. Propõe-se que sejam definidas as seguintes metas para a realização do trabalho:

1. **antes da primeira aula** deve ler a documentação e o código fornecido;
2. **no fim da primeira semana** deve ter realizado a leitura de todos os campos de cabeçalho do pedido e preparado um cabeçalho para as páginas retornadas pelo servidor, na classe HTTPAnswer;
3. **no fim da segunda semana** deve ter programado a reutilização de ligações com HTTP/1.1, e a validação do "*If-Modified-Since*";
4. **no fim da terceira semana** deve ter programado a RESTful API com o carregamento de classes rit2REST que a implementam, e a passagem de parâmetros para os respectivo objectos;
5. **no fim da quarta semana** deve ter realizado todo o código de leitura de Cookies, e programado a resposta a pedidos e a gestão de Cookies no ficheiro rit2REST.java;
6. **no fim da última semana** deve ter acabado todas as tarefas anteriores, e ter realizado vários testes.

4.7 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.

BIBLIOGRAFIA

[1] James Marshall, “HTTP made really easy”, http://www.jmarshall.com/easy/http
[2] HTTP 1.0, http://www.ietf.org/rfc/rfc1945.txt
[3] HTTP 1.1, http://www.ietf.org/rfc/rfc2616.txt
[4] Andrew S. Tanenbaum e David J. Wetherall, “Computer Networks 5th Edition”
[5] HTTP State Management Mechanism, http://ftp.di.fct.unl.pt/pub/documents/rfc/rfc2109.txt
[6] Hypertext Markup Language - 2.0, http://www.ietf.org/rfc/rfc1866.txt
[7] Nik Silver, "CGI Tutorial", http://www.comp.leeds.ac.uk/Perl/Cgi/start.html