FIT Summer School
Novi Sad, June 2009

Concurrent and Global Computing

Part II: A brief introduction
to server-side technologies

Ivan Scagnetto (ivan.scagnetto@dimi.uniud.it)
University of Udine

The notion of computing

- The classical notion of computing (i.e., a sequential application of statements producing some output values from the input ones) has being replaced by a new paradigm where:
  - programs do not always terminate (reactive programs);
  - there are many concurrent computations and communication activities between different processes;
  - the perspective is network-centric rather than computer-centric.

Global Computing (GC)

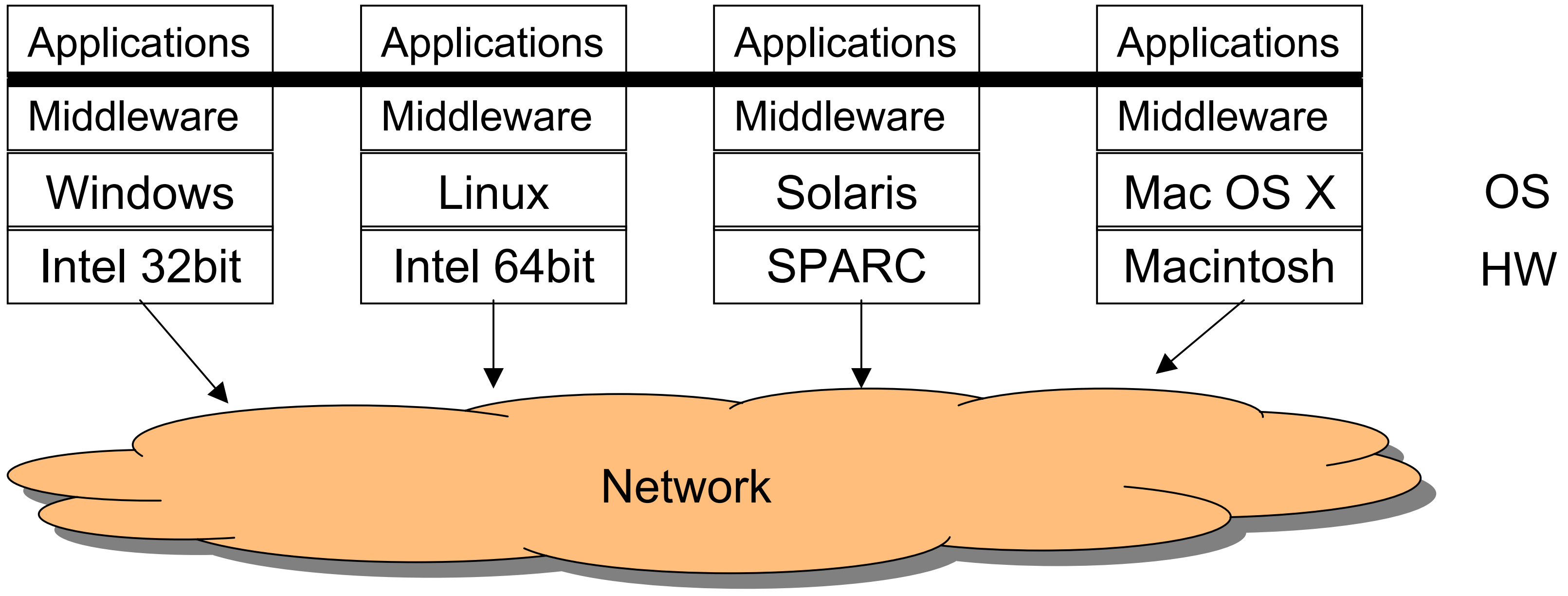
- According to the definition given in the Proactive Initiative of the EU 6th Framework Programme:
  - "*Global computing refers to computation over 'global computers', i.e., computational infrastructures available globally and able to provide* ***uniform services*** *with variable guarantees for communication, co-operation and mobility, resource usage, security policies and mechanisms, etc., with particular regard to exploiting their universal scale and the programmability of their services.*"
  - The notion of "uniform services" is well-known in the computer science literature and it is also referred to as "**middleware**".

Distributed Systems

- The infrastructure needed in order to provide global computing services builds upon the existing **networks** and technologies used to provide **common network services**.
- Moreover, due to the concurrent nature of computation in this framework, we can speak of **distributed systems**, where:
  - a single node of a distributed system is a **complete** computer (CPU, memory, I/O peripherals);
  - the nodes of a distributed system communicate and coordinate their processes through a **network** (e.g., the Internet) by means of protocols based upon a **message passing** paradigm.

Middleware and Distributed Systems

- Middleware is implemented in distributed systems as a software layer between applications and the operating system:



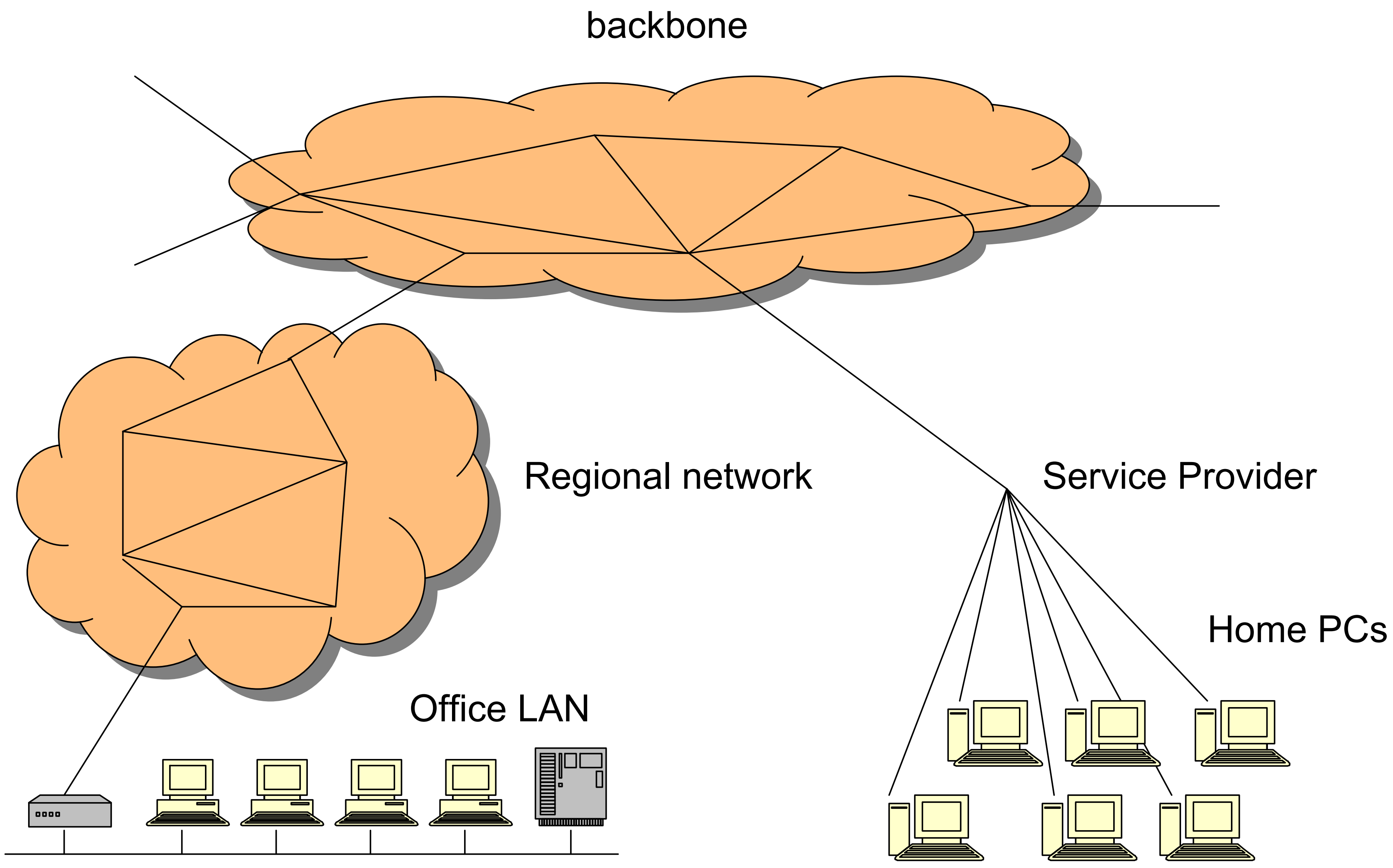
GC vs. GRID

- "GRID" research was originally introduced in order to **increase computing power** by sharing tasks between different computers (e.g., SETI@Home).
- Currently, people refer to GRID as a set of middleware technologies aimed at supporting **resource sharing** between computers.
- Instead, "*GC research provides the foundations for the development of large-scale general purpose computer systems that have dependably predictable behaviour, for the needs of a distributed world*".
- Then, the specific goals might be different:
  - Resource Sharing,
  - Internet commerce (web applications and web services),
  - Ambient Intelligence (e.g. via UMTS).
- GC research is not just middleware, but addresses a range of issues that are not explicitly addresses by GRID:
  - mobility,
  - ubiquity,
  - dynamicity,
  - interactivity.

The network

- The hardware of a distributed system can be very heterogeneous when we think of its network infrastructure:
  - LANs: they cover small geographic areas:
    - regular structure: ring, star, bus;
    - speed: 100Mb/s or greater;
    - nodes: personal computers and/or workstations; some servers and/or mainframes.
  - WANs: they connect geographically distant sites:
    - point-to-point connections on long distances lines (e.g., telephone cables);
    - speed: from about 100Kb/s to 34Mb/s;
    - broadcast implementation usually requires multiple messages.

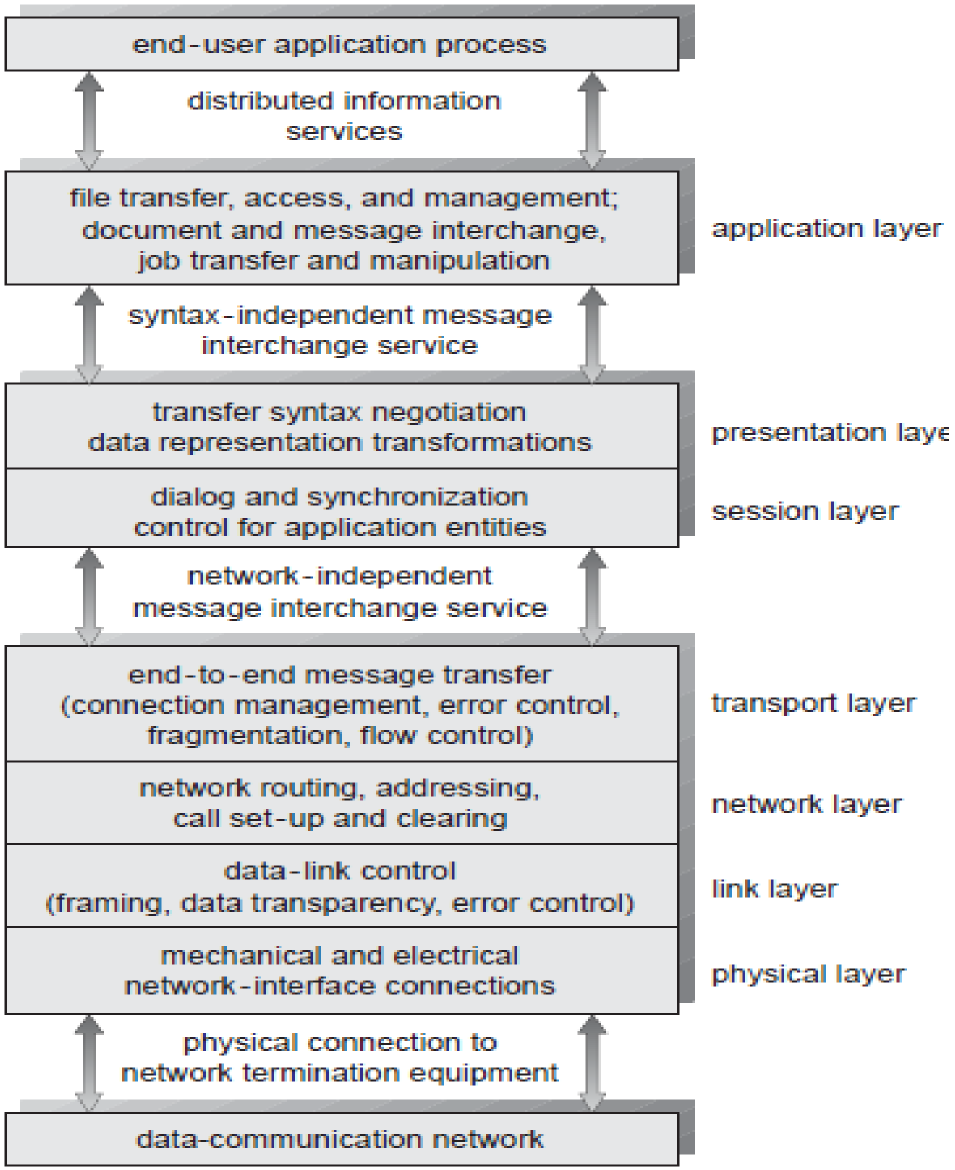
The network


backbone
Regional network
Service Provider
Home PCs
Office LAN

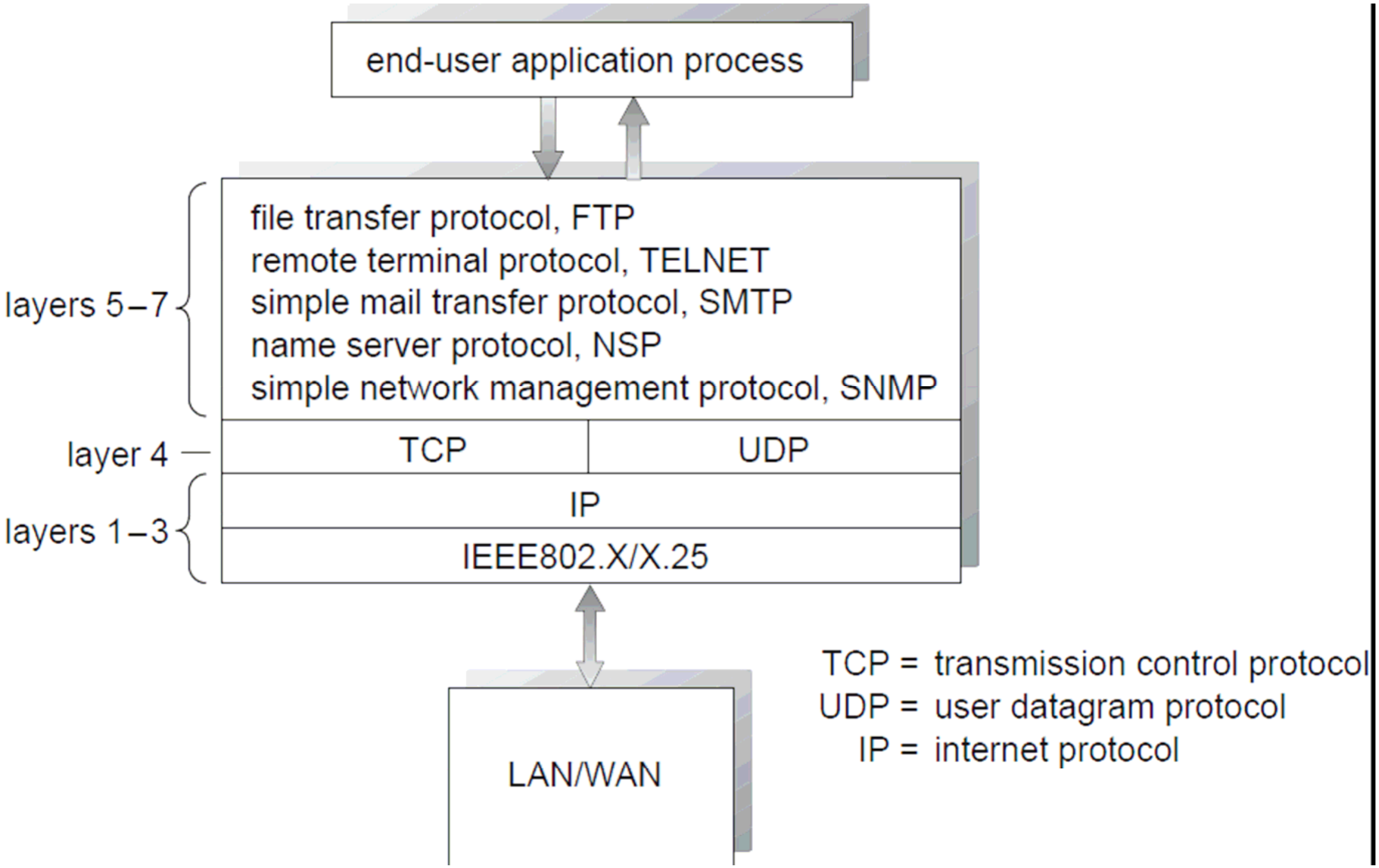
Network protocols

- All communications between nodes are regulated by a common set of protocols (set of rules).
- The implementation of protocols for advanced services is easier if protocols are **layered in a stack**:
  - each layer of the stack implements some new features based upon those of the inferior layers;
  - this rule ensures modularity and simplicity of design and implementation.
- When dealing with distributed systems it is very important to follow **standard protocols**.

The ISO/OSI model


end-user application process
distributed information services
file transfer, access, and management; document and message interchange, job transfer and manipulation
application layer
syntax-independent message interchange service
transfer syntax negotiation data representation transformations
presentation laye
dialog and synchronization control for application entities
session layer
network-independent message interchange service
end-to-end message transfer (connection management, error control, fragmentation, flow control)
transport layer
network routing, addressing, call set-up and clearing
network layer
data-link control (framing, data transparency, error control)
link layer
mechanical and electrical network-interface connections
physical layer
physical connection to network termination equipment
data-communication network

The TCP/IP protocols stack


end-user application process
file transfer protocol, FTP
remote terminal protocol, TELNET
simple mail transfer protocol, SMTP
name server protocol, NSP
simple network management protocol, SNMP
layers 5–7
layer 4
TCP
UDP
IP
IEEE802.X/X.25
layers 1–3
LAN/WAN

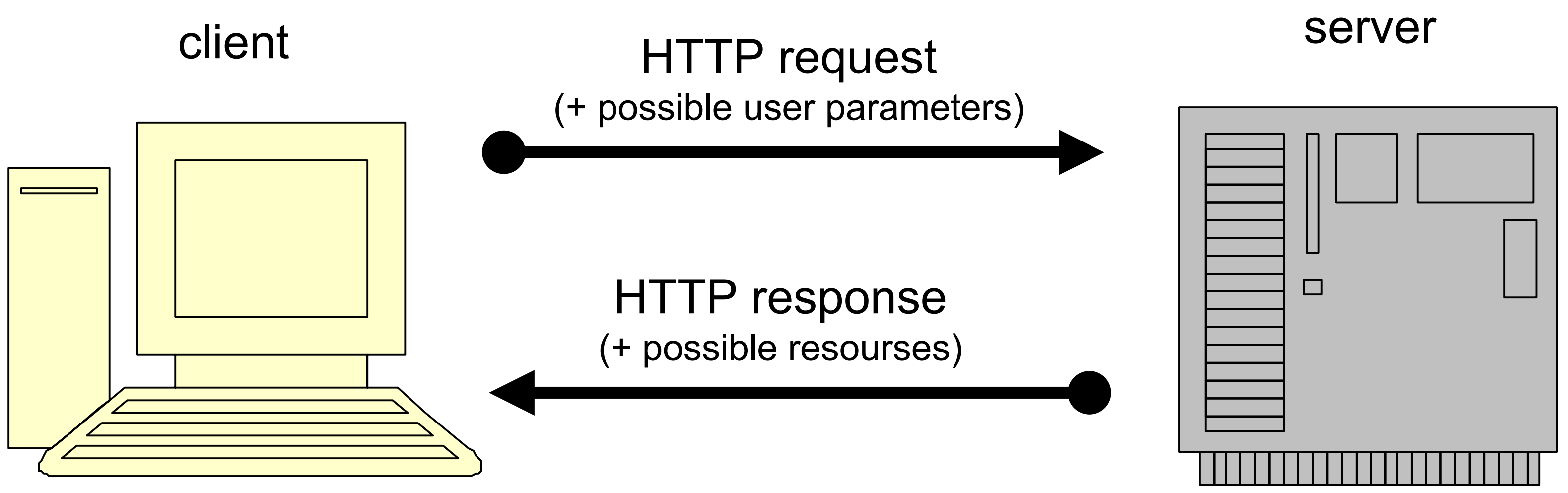
TCP =	transmission control protocol
UDP =	user datagram protocol
IP =	internet protocol

Server Side Technologies

- A common way to design and implement GC applications is to exploit the World Wide Web (WWW) notion of middleware.
- Indeed WWW can be seen as a uniform way to collect and manage a set of distributed and heterogeneous documents.
- This is an example of middleware since the user can require a document without worrying about its physical location.
- As a consequence, at the application level it is straightforward to use the HTTP protocol.

The HTTP protocol

- The HyperText Transfer Protocol (HTTP – RFC 2616) is a very simple one:



HTTP request

- A HTTP request has two components:
  - one header: some text lines of limited length,
  - one body: unlimited.
- Header:
  - first line: one method (GET, HEAD, POST, etc.), the URI (Uniform Resource Identifier) of the requested resource and the protocol version;
  - the other text lines contain some metadata exchanged between the browser and the server.
- Body:
  - it can be either empty…
  - …or it can contain user’s parameters sent through a form (POST method).

A sample HTTP request

- GET /dir1/dir2/file.html HTTP/1.1
- Host:www.domain.com
- Accept: text/xml,application/xml, application/(X)HTML+xml,text/html,text/plain,image/png
- Accept-language: it,en
- Accept-charset: ISO-8859-1
- User-agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; it; rv:1.8.0.5) Gecko/20060723 Firefox/1.5.0.5
- Referer: http://www.google.it/search?q=a+good+book&start=0&ie=utf-8

HTTP response

- The structure is the same of the request (header+body).
- Header:
  - status code (outcome), 3 digits:
    - 2XX: success,
    - 4XX: errors (e.g., 404 NOT FOUND),
    - …
  - metadata from server to client (e.g., Content-Type and Content-Length)
- Body:
  - In case of success, the body contains the requested resource.

A sample HTTP response

- HTTP/1.1 200 OK
- Date: Mon, 31 Jul 2007 15:27:59 GMT
- Server: Apache/2.0.54 (Debian GNU/Linux) mod_python/3.1.3 Python/2.3.5 PHP/4.3.10-16
- Expires: Thu, 19 Nov 2007 08:52:00 GMT
- Content-Type: text/html
- Content-Length: 3451

- <html>
- <head>
- … (all the requested file)

The browser

- Interpretation of user’s requests
- Obtaining the resources:
  - the browser begins a transaction with the host server,
  - sending the appropriate HTTP command to request the resource (GET, POST),
  - waiting for the response, including the resource (if the latter exists),
  - if the resource is a HTML document, the other resources included in it are automatically requested by the browser to the right servers.
- Managing the resources according to their MIME type:
  - the browser activates the most appropriate action in order to manage the resource; often such action produces an output result and, more in detail, such output amount to visualize something to the user.

The browser’s actions

- When performing an action, the browser can use either internal code or other programs:
  - internal code: in the browser there are all the functions needed in order to visualize resources encoded in standard web formats and in other common formats including (HTML, JPEG, GIF, PNG, …);
  - plugin;
  - helper.

The browser’s actions: plugins

- External subprograms loaded when the browser is run.
- Their execution takes place inside the browser’s interface:
  - a standard API links them to the browser’s interface.
- Plugins are usually developed by creators of a peculiar file type.
- Moreover, they often imply the automatic execution of external programs.
- Video viewers (MPEG, Real, WindowsMedia, Quicktime) and media managers (for, e.g., MIDI sound files) are implemented following this pattern.

The browser’s actions: helpers

- Helpers are external programs which are automatically executed when a resource, whose type is associated to a given program, is received.
  - (many plugins can be included in this category).
- The resource is automatically saved in a file which is passed as a parameter to the helper program.
- A classical example is Acrobat Reader, used to view PDF files.
- It is possible to configure the browser in order to use any programs to manage resources according to their MIME type or to the extension of the file name.
  - (e.g., it is possible to use Word as a helper for the documents whose file names end in .doc).

Beyond visualization: programs execution

- A possible category of browser’s actions is the execution of programs received as resources:
  - for the sake of security, the effects due to these programs must be controlled and limited,
  - in order to avoid the diffusion of viral contents through the Web.
- In this sense we can speak about client-side programming.

Client-side programming

- Writing programs which will run into a particular environment: the browser.
- Currently, there are three main approaches for client-side programming:
  - scripting,
  - Java applets,
  - Flash objects.

Client-side scripting

- Embedding brief code fragments (called scripts) into the HTML tags of a page.
- Scripts are interpreted by the browser.
- Scripting languages: automation, manipulation, and personalization of the system features:
  - they are less “general-purpose” (i.e., more focused) of traditional languages, acting on the features of the hosting systems.
- In the cases where the hosting system is the browser,
  - a scripting language supplies primitives and mechanisms to interact with the objects visualized by the browser (windows, HTML and CSS documents) and to manage user’s input and the output;
  - the standard language is ECMAscript (Javascript), with DOM libraries.

Java applets

- Java applets are compiled programs,
  - instances of a suitable class
- The applet is indeed constituted by a separate (**`.class`**) file containing the intermediate code:
  - Such file is inserted into a web page by means of the **`<object>`** tag.
- A Java applet can be a complete program, but it is run into the browser interface.
- Its actions are limited:
  - no access to the file system,
  - communications can only address the originating server,
  - …
  - example: **`http://rsb.info.nih.gov/ij/`**

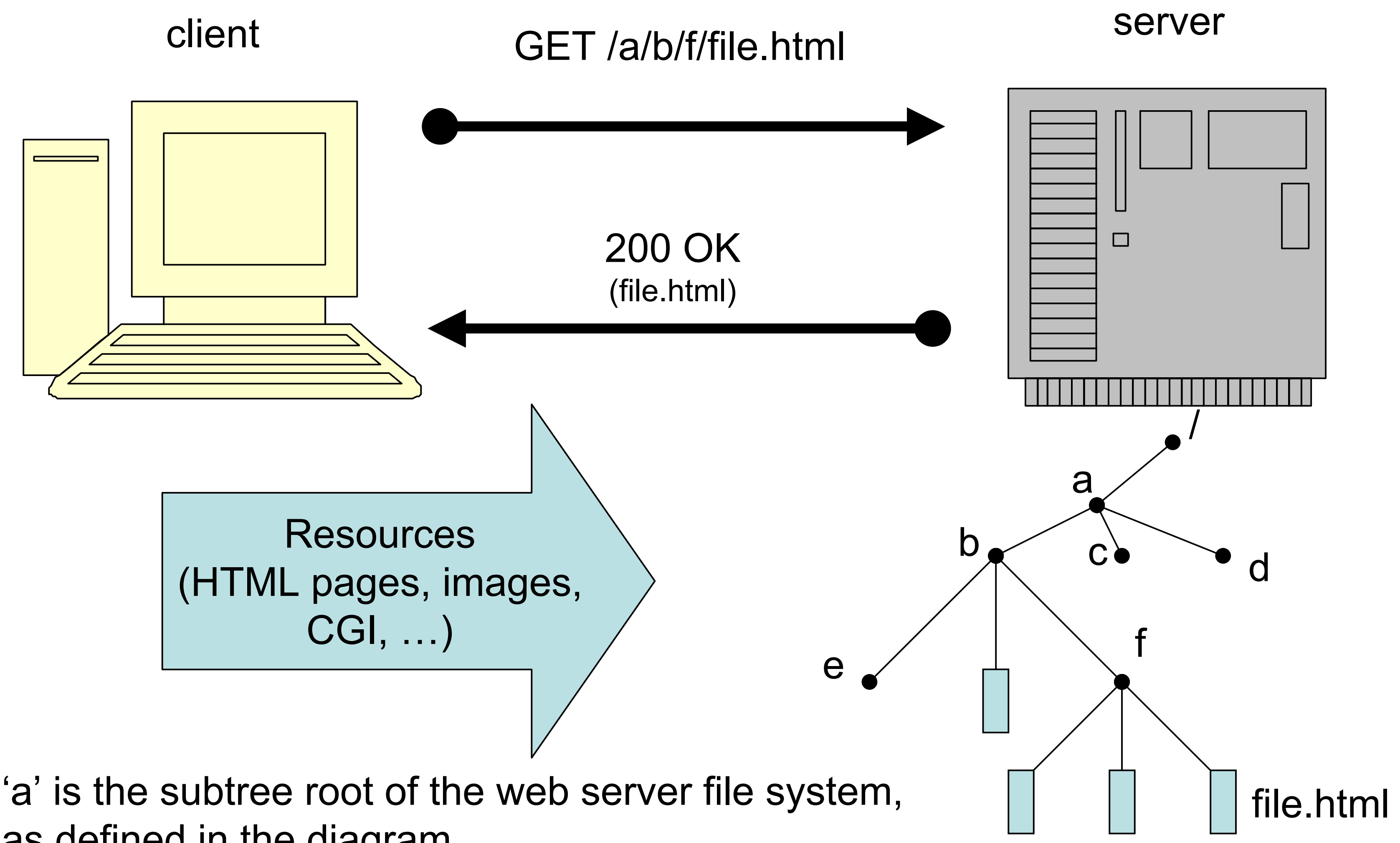
Flash

- Programs in a compiled multimedia oriented language:
  - embedded into web pages by means of the **`<object>`** tag,
  - run by an appropriate plugin installed in the browser.
- Flash contents are usually oriented towards multimedia presentations
  - they are used to create animated pages with non-standard interaction features (i.e., different from the classical browser interface).
- Flash is proprietary software (the plugin is free, but the development tools are not).

The web server

- Basically, the web server is the (reactive) program, managing the HTTP protocol.
- Its main activity is distributing resources, according to clients' requests.
- Distributed resources are usually ready and hosted in a given subtree of the file system (static pages).
- However, they can also be generated at the moment the requests arrive (dynamic pages), usually on the basis of user's parameters specified by means of a HTML form.

The web server



‘a’ is the subtree root of the web server file system, as defined in the diagram

Web applications

- They are applications which can be used through a web browser: **(X)HTML** substitutes the classical **GUI**.
- They allow to run programs on a remote server.
- Examples:
  - Web mail
  - E-Commerce
  - CMS (Content Management Systems)
  - etc.

Advantages

- The installation phase has to be carried out once on a single computer (the server).
  – This fact implies easiness of management and maintenance.
- However, the application can be used concurrently by several clients.
  – Unique requirements are the availability of a browser and connectivity.
  – No needs to install/manage ad-hoc software on the clients.

Developing web applications with J2EE

- J2EE is a version of the traditional Java SDK oriented towards the development of **distributed** (≡ **enterprise**) **applications**.
- J2EE features several components:
  - servlets,
  - JSP,
  - JDBC,
  - XML support,
  - RMI, Corba,
  - JNDI, JMS, JavaMail,
  - EJB etc.
- For the development of web applications the main technology is represented by servlets.

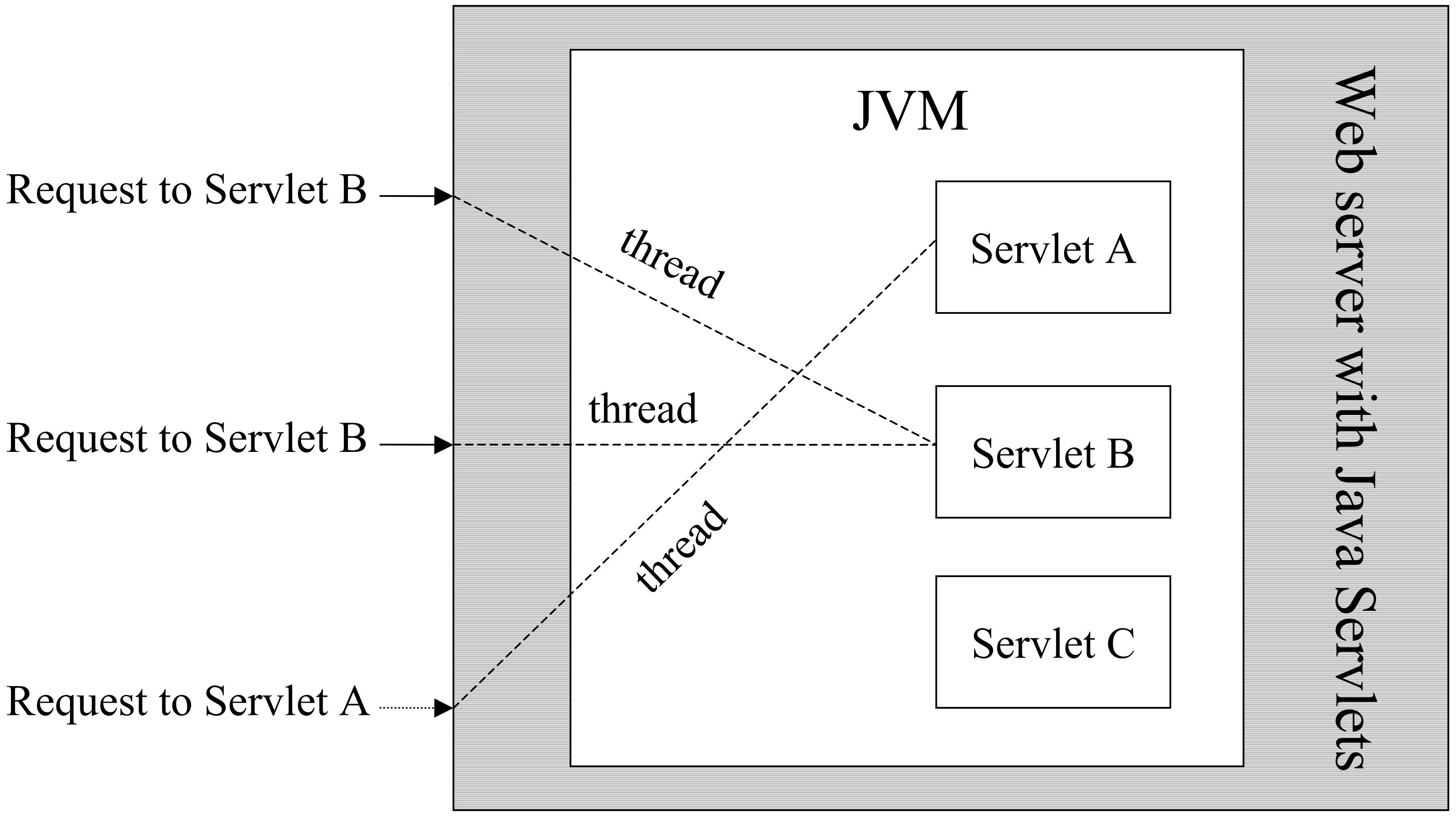
Java servlets

- A Java servlet is a **server extension**, i.e., a Java class which can be dynamically loaded to extend the server features.
- Usually, servlets are mainly used in web servers (as an alternative to CGI scripts).
- However, nothing forbids to use the servlet technology with other kinds of servers: FTP server, mail server etc.
- Servlets run on a server's JVM, hence they are secure and portable.
- Differently from applets, they **do not require** any Java support in the web browser.

Servlets advantages

- All the requests addressing servlets are implemented as distinct threads of the same (server) process: this implies **efficiency** and **scalability**.
- Servlets are **portable**:
  - between different operating systems;
  - between different web servers (all the most popular web servers support servlets, either directly or through plug-ins, add-ons etc.).

Java servlets and the web server



Java servlet support

- In order to use the Java Servlet technology we need:
  - JVM;
  - Servlet API (**`javax.servlet`** and **`javax.servlet.http`** classes), supplied in bundle with the JSDK or incorporated in some web servers;
  - a Servlet Engine (e.g., Tomcat); the available types of engines are the following:
    - standalone;
    - add-on;
    - embeddable.

Servlet engine types

- **Standalone Server Engine**: a standalone server with native support for Java Servlets.
- **Add-on Servlet Engine**: plug-in adding servlet support for existing servers.
- **Embeddable server engine**: development platform supporting servlets, embeddable in other applications.
- Servlet engines can substantially differ in features, performance, etc.
- Before choosing a specific versione, it is wise to test it, in order to see if it meets the requirements for the applications one has in mind.
- An updated list with the servlet engines is mainteined by Sun at the following URI: **`http://java.sun.com/products/servlet/industry.html`**

Servlet features

- Portability (**write once, serve everywhere**).
- Power (networking, multithreading, database connectivity…)
- Efficiency and persistency (thread implementation, native notion of state)
- Security (type safety, garbage collection, exception, Java Security Manager)
- Elegance (cookie, session management, …)
- Integration (in-server)
- Extendibility and flexibility (server-side include, JSP)

Servlet API

- Servlets use classes and interfaces of two packages:
  - **`javax.servlet`** (generic and protocol-independent servlets);
  - **`javax.servlet.http`** (HTTP specific servlets).
- Each servlet has to implement the interface **`javax.servlet.Servlet.`** This usually happens by extending:
  - **`javax.servlet.GenericServlet`** (generic and protocol-independent servlets);
  - **`javax.servlet.http.HttpServlet`** (HTTP specific servlets).

Servlet API

- Servlets do not have the **`main()`** method.
- The server calls specific methods, in response to a request:
  - **`service()`** is called whenever a request addresses a servlet.
- Thus, a generic servlet must override the **`service()`** method.
- Instead, a HTTP servlet overrides methods **`doGet()`** and **`doPost()`**. The **`service()`** method in this case coordinates the forwarding of the requests to the abovementioned methods and it must not be modified.

Tomcat

- The servlet engine we will use is Tomcat: it can work either as a **stand-alone** product or as a **module** of the Apache web server.
- It can be freely downloaded (for many popular operating systems such as Linux, Windows, Mac OS X) from either `http://jakarta.apache.org` or `http://tomcat.apache.org`
- The default installation accepts incoming requests on port 8080: `http://<server-address>:8080`
- When it receives the first request for a servlet, it creates an **instance** of the corresponding **Java class**.
- Moreover, at each request, it creates a **thread** executing the appropriate method of the servlet, according to the kind of HTTP request.
- The class instance is destroyed in correspondence with one of the events of **Stop**, **Reload**, **Undeploy**, **Shutdown** of Tomcat.

Tomcat – directory structure

- **`$CATALINA_HOME`** (or **`$TOMCAT_HOME`**) represents the root directory of the Tomcat installation and it contains the following subdirectories:
  - bin
  - lib
  - conf
  - logs
  - temp
  - webapps
  - work

Tomcat – **`bin`** directory

- This directory contains binary files and management scripts of Tomcat.
- Two of them are very important:
  - **`startup.sh`** (**`startup.bat`** in Windows): starts Tomcat
  - **`shutdown.sh`** (**`shutdown.bat`** in Windows): stops Tomcat
- During the development phase of a web application it is very common stopping and restarting the service.

Tomcat – **`lib`** directory

- In this directory there are the common packages relative to Tomcat and to the web applications deployed in the directory **`webapps`**:
  - **`lib/`**
    - **`.jar`**
    - **`servlet-api.jar`**: to add to the classpath:
      - **`javac -classpath $TOMCAT_HOME/lib/servlet-api.jar <file.java>`**
    - **`tools.jar`**
    - **`...`**

Tomcat – **`conf`** directory

- This directory contains the configuration files of Tomcat.
- In particular there are the following ones:
  - **`server.xml`**: general configuration parameters of Tomcat (in case of modifications it is necessary to restart the server).
  - **`tomcat-users.xml`**: informations about the users of Tomcat.

Tomcat – `server.xml` (I)

- Structure of the file:
  - <Server> (principal tag)
    - <Service> (type of server, e.g., stand-alone)
      - Connector (TCP port where Tomcat listens to requests)
      - Engine (requests manager)
        - » Host (virtual host)
        - » Context (web application)
        - » Realm

Tomcat – **`server.xml`** (II)

- The **`<Context>`** tag allows one to define the “context” of a new web application:

```
<Context
    path="/examples"
    docBase="examples"
    debug="0"
    reloadable="true" />
```

URI

Position in the file system

Example: defining an application outside the standard dir **`webapps`**

- With the following Context tag specification, one can define a new application whose files are located in **`C:\newapp`** and responding to requests whose URI is **`http://<host>:8080/new`**

```
<Context
    path="/new"
    docBase="c:\newapp"
    debug="0"
    reloadable="true" />
```

Tomcat – `tomcat-users.xml` (I)

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="tomcat"/>
  <role rolename="role1"/>
  <role rolename="manager"/>
  <role rolename="admin"/>
  <user username="tomcat" password="tomcat" roles="tomcat"/>
  <user username="both" password="tomcat"
   roles="tomcat,role1"/>
  <user username="role1" password="tomcat" roles="role1"/>
  <user username="admin" password="adminpwd"
   roles="admin,manager"/>
</tomcat-users>
```

- Notice that the users’ passwords are in **clear text**!

Tomcat – **`logs`** directory

- This directory contains the “logbooks” of the Tomcat’s activity.
- There are many files whose names always have a suffix indicating a datestamp. For instance, some of them are:

  `localhost.YYYY-MM-DD.log`

  `host-manager.YYYY-MM-DD.log`

  `manager.YYYY-MM-DD.log`
- What is registered in those files?
  - events relative to the Tomcat server (startup, shutdown, …);
  - possible errors;
  - …

Tomcat – **`temp`** directory

- Temp is a work directory where Tomcat stores temporary files during his activity.
- This directory is very important for the correct behaviour of Tomcat and it should never be removed (even if sometimes it is empty).

Tomcat - **`webapps`** directory

- This directory contains the web applications managed by Tomcat:
  - users’ applications;
  - predefined applications installed together with Tomcat (e.g., the examples stored in **`examples/`**).

Tomcat – `work` directory

- This is another directory for temporary files.
- In particular, Tomcat uses it to store servlets generated from JSP (Java Server Pages) files.

The first servlet: Hello, world!

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {
        public void doGet(HttpServletRequest req, HttpServletResponse res)
                        throws ServletException, IOException {

                res.setContentType("text/html");
                PrintWriter out = res.getWriter();

                out.println("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML "+
                "1.1//EN\" \"http://www.w3c.org/TR/xhtml11/DTD/xhtml11.dtd\">");
                out.println("<html xmlns=\"http://www.w3.org/1999/XHTML\""+
                "xml:lang=\"en\" lang=\"en\">");
                out.println("<head><title>Hello, world!</title></head>");
                out.println("<body>");
                out.println("<strong>Hello, world!</strong>");
                out.println("</body></html>");
        }
}
```

Deployment on Tomcat

- Compile the servlet, including in the classpath (through either the ambient variable **`CLASSPATH`** or the **`–classpath`** option of **`javac`**) the package **`servlet-api.jar`** present in the directory **`$CATALINA_HOME/lib`**:
  - **`cd <path_to_HelloWorld.java>`**
  - **`javac -classpath "$CATALINA_HOME\lib\servlet-api.jar" HelloWorld.java`**
- Write the deployment descriptor file **`web.xml`**.
- Copy **`HelloWorld.class`** and **`web.xml`** on the server.
- Restart the service.

Deployment descriptor file

- Since a servlet is a Java program, the servlet engine (e.g., Tomcat) must know how to map incoming requests to the corresponding servlet.
- Such information is provided by the deplyment descriptor file (**`web.xml`**).
- There is one **`web.xml`** for each application managed by the container.
- For each **`<servlet>`** there is a corresponding **`<servlet-mapping>`**. The latter specifies the class file to associate to a given request pattern (URL).

web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
    <servlet>
        <servlet-name>FirstTest</servlet-name>
        <servlet-class>HelloWorld</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>FirstTest</servlet-name>
        <url-pattern>/servlet/First</url-pattern>
    </servlet-mapping>
</web-app>
```

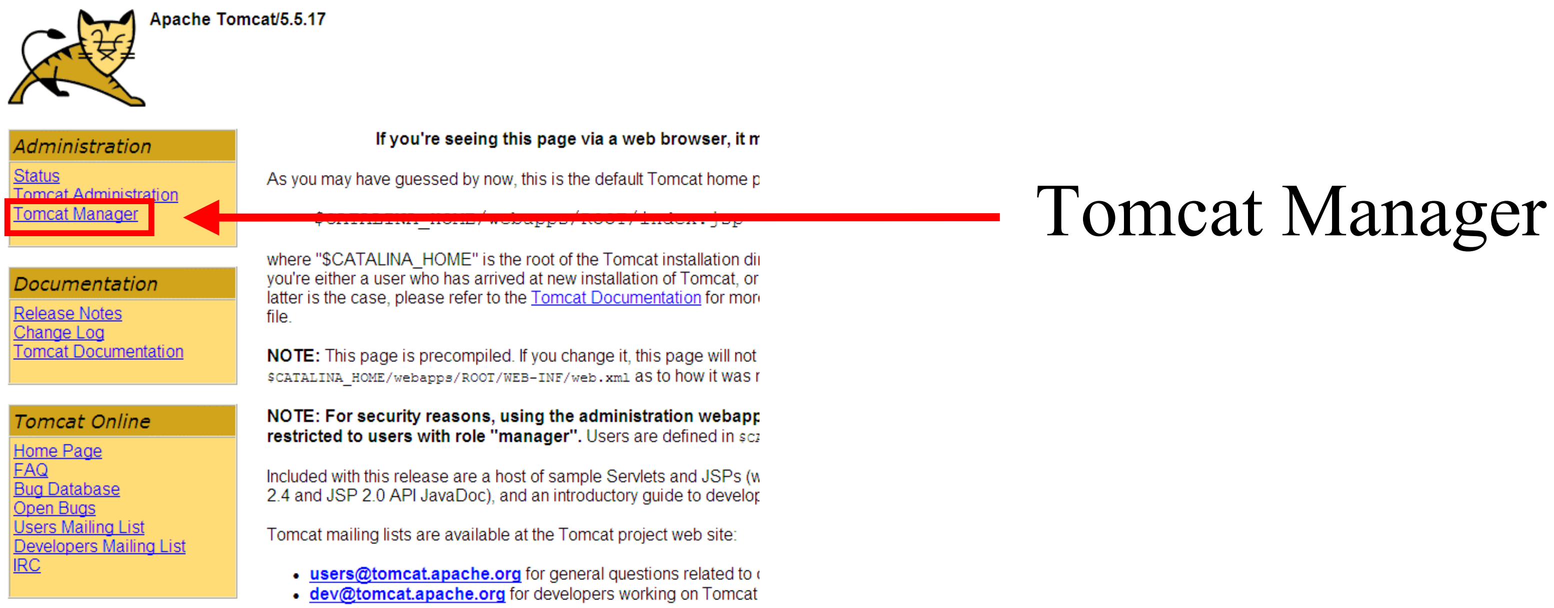
These names must match

Deployment on Tomcat

```
$CATALINA_HOME/webapps
   └──► test
          └──► WEB-INF
                  ├──► web.xml
                  └──► classes/
                          └──► HelloWorld.class
```

The Tomcat Manager

- Connect to `http://<server-address>:8080` and click the link “Tomcat Manager” (user and password are specified in the configuration file **`tomcat-users.xml`**):



Apache Tomcat/5.5.17

Administration
Status
Tomcat Administration
Tomcat Manager

Documentation
Release Notes
Change Log
Tomcat Documentation

Tomcat Online
Home Page
FAQ
Bug Database
Open Bugs
Users Mailing List
Developers Mailing List
IRC

If you're seeing this page via a web browser, it m

As you may have guessed by now, this is the default Tomcat home p

`$CATALINA_HOME/webapps/ROOT/index.jsp`

where "$CATALINA_HOME" is the root of the Tomcat installation di
you're either a user who has arrived at new installation of Tomcat, or
latter is the case, please refer to the Tomcat Documentation for mor
file.

NOTE: This page is precompiled. If you change it, this page will not
`$CATALINA_HOME/webapps/ROOT/WEB-INF/web.xml` as to how it was r

**NOTE: For security reasons, using the administration webapp
restricted to users with role "manager".** Users are defined in `$c`

Included with this release are a host of sample Servlets and JSPs (w
2.4 and JSP 2.0 API JavaDoc), and an introductory guide to develop

Tomcat mailing lists are available at the Tomcat project web site:

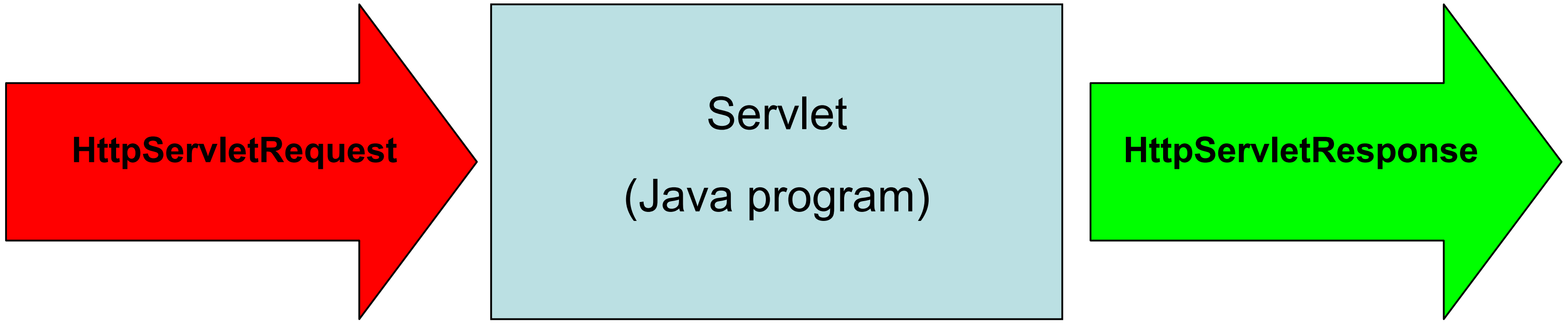
- **users@tomcat.apache.org** for general questions related to c
- **dev@tomcat.apache.org** for developers working on Tomcat

The Tomcat Manager

- Among the several application managed by the Tomcat Manager, find the one named test, then
  - either click Stop, then click Start
  - or click Reload
  - (important: do not click Undeploy).
- Access to the servlet either clicking the relative link or through the following URI:

```
http://<server-address>:8080/test/servlet/First
```

The “input/output” classes



The interface with the HTTP protocol is represented by the classes **`HttpServletRequest`** (input) and **`HttpServletResponse`** (output).

HttpServletRequest

- It is an interface, defined in `javax.servlet.http`, adding to the "superinterface" `ServletRequest` (for generic servlet), defined in `javax.servlet`, specific methods for HTTP requests.
- It represents the client's request to a servlet.
- The corresponding object is created by the container (e.g., Tomcat) when the request is received and it is passed to the appropriate method of the servlet.
- Main methods:
  - `getInputStream`: to read the data sent by the client in the request in "raw" mode;
  - `getParameter`: to gather the request's parameters.

HttpServletResponse

- It is an interface, defined in `javax.servlet.http`, adding to the “superinterface” `ServletResponse` (for generic servlets), defined in `javax.servlet`, specific methods for HTTP requests.
- It represents the client’s response of the servlet.
- The corresponding object is created by the container (e.g., Tomcat) when the request is received and passed to the appropriate method of the servlet.
- Main methods:
  - `setContentType`: to specify the MIME type of the content which will be sent to the client;
  - `getWriter`: to have a reference to the data stream towards the client.

Experimenting with HTML forms (I)

```
…
<body>
<p>
Widgets in a form
</p>
<form action="servlet/HandleForm" method="post">
<p>text box sample: <input name="textBox1" type="text" size="10" /></p>
<p>Check boxes sample: <br />
  choice 1 <input name="check1" type="checkbox" value="1" /><br />
  choice 2 <input name="check2" type="checkbox" value="2" /><br />
  choice 3 <input name="check3" type="checkbox" value="3" /><br />
  choice 4 <input name="check4" type="checkbox" value="4" /><br />
</p>
<p>Radio box sample:<br />
  choice 1 <input name="radio1" type="radio" value="true" checked="true" /><br />
  choice 2 <input name="radio1" type="radio" value="false" /><br />
</p>
…
```

Experimenting with HTML forms (II)

```
…
<p>
  Selection sample:<br />
  <select name="select1">
    <option value="opt1">first choice</option>
    <option value="opt2">second choice</option>
    <option value="optno" selected>-</option>
  </select>
</p>
<p>
  <input name="resetButton" type="reset" />
  <input name="submitButton" type="submit" />
</p>
</body>
</html>
```

Experimenting with HTML forms
The servlet (I)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HandleForm extends HttpServlet {
  public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    out.println("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML "+
      "1.1//EN\" \"http://www.w3c.org/TR/xhtml1/DTD/xhtml11.dtd\">");
    out.println("<html xmlns=\"http://www.w3.org/1999/XHTML\""+
      "xml:lang=\"en\" lang=\"en\">");
    out.println("<head><title>Hello, world!</title></head>");
    out.println("<body>");
    …
```

Experimenting with HTML forms
The servlet (II)

```
  out.println("<strong>You typed/selected the following values:</strong>");
    out.println("<br />Text box value: "+(req.getParameter("textBox1").equals("") ?
                     "no value" : (req.getParameter("textBox1"))));
    out.println("<br />Check box 1 value: "+(req.getParameter("check1")==null ?
                     "not checked" : req.getParameter("check1")));
    out.println("<br />Check box 2 value: "+(req.getParameter("check2")==null ?
                     "not checked" : req.getParameter("check2")));
    out.println("<br />Check box 3 value: "+(req.getParameter("check3")==null ?
                     "not checked" : req.getParameter("check3")));
    out.println("<br />Check box 4 value: "+(req.getParameter("check4")==null ?
                     "not checked" : req.getParameter("check4")));
    out.println("<br />Radio button 1 value: "+req.getParameter("radio1"));
    out.println("<br />Selection value: "+req.getParameter("select1"));
    out.println("</body></html>");
  }
}
```

web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
    …
    <servlet>
        <servlet-name>HandleForm</servlet-name>
        <servlet-class>HandleForm</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HandleForm</servlet-name>
        <url-pattern>/servlet/HandleForm</url-pattern>
    </servlet-mapping>
</web-app>
```

Tracking the client

- Usually, a web application is composed by more than a single page (resource).
- Thus, it is useful keeping track of the clients connecting to the service, in order to make easier the application development.
- However, **the HTTP protocol is “stateless”**:
  - the client makes a request;
  - the server satisfies the request (if certain conditons are verified);
  - the client/server interaction is “forgotten” and the next request is considered.

Tracking the client

- In order to tracking the client’s status in a web application, common solutions are the following:
  - inserting status information directly in the URIs (a complicated and inherently insecure solution);
  - Inserting status information in form hidden fields (insecure solution);
  - cookies;
  - sessions (server-side).

Cookies

- Cookies allow to store small quantities of data in the client’s browser on behalf of a web application.
- The information exchanged between server and client is embedded in the headers of the requests and responses defined by the HTTP protocol.
- The servlet API defines the class **`javax.servlet.http.Cookie`** which allows the programmer to easily manage cookies.
- **`HttpServletResponse`** features the **`addCookie()`** method to create new cookies.
- **`HttpServletRequest`** features the **`getCookies()`** method to read cookies from the HTTP request.

Creating cookies

- Cookies can be initialized in order to be valid only in a given domain, in a given path and for a given amount of time:

```
String userid = createUserID();
Cookie c = new Cookie("userid",userid);
c.setDomain(".company.com");
c.setPath("/");
c.setMaxAge(900); // in seconds
resp.addCookie(c);
```

Reading cookies

- The **`getCookies()`** method gives access to the whole set of cookies.
- To read the value of a given cookie, it is necessary to scan all the cookies set:

```
Cookie[] cookies;
cookies = req.getCookies();
String userid = null;
for (int i = 0; i < cookies.length; i++)
    if (cookies[i].getName().equals("userid"))
                userid = cookies[i].getValue();
```

Sessions

- The servlet API offers a “session tracking” mechanism delegating to the server the tracking management of the client.
- At the first request of a client to a servlet supporting sessions a new object of type **`javax.servlet.http.HttpSession`** is created.
- The **`getSession()`** method allows to obtain a reference to the current session.
- In order to associate the right session to the relative client a session ID (unique for each client) is stored in a cookie or embedded in URIs.

Useful methods of HttpSession

- **`getID()`**: returns the current session ID;
- **`setAttribute()`**: associates an object to the current session:

```
session.setAttribute("myservlet.hitcount",
                              new Integer(34));
```

- **`getAttribute()`**: recovers an object from the current session:
- **`Integer hits = (Integer)session.getAttribute("myservlet.hitcount")`**
- To avoid name conflicts, objects stored in sessions are named according to the following pattern: `<application-name>.<object-name>`

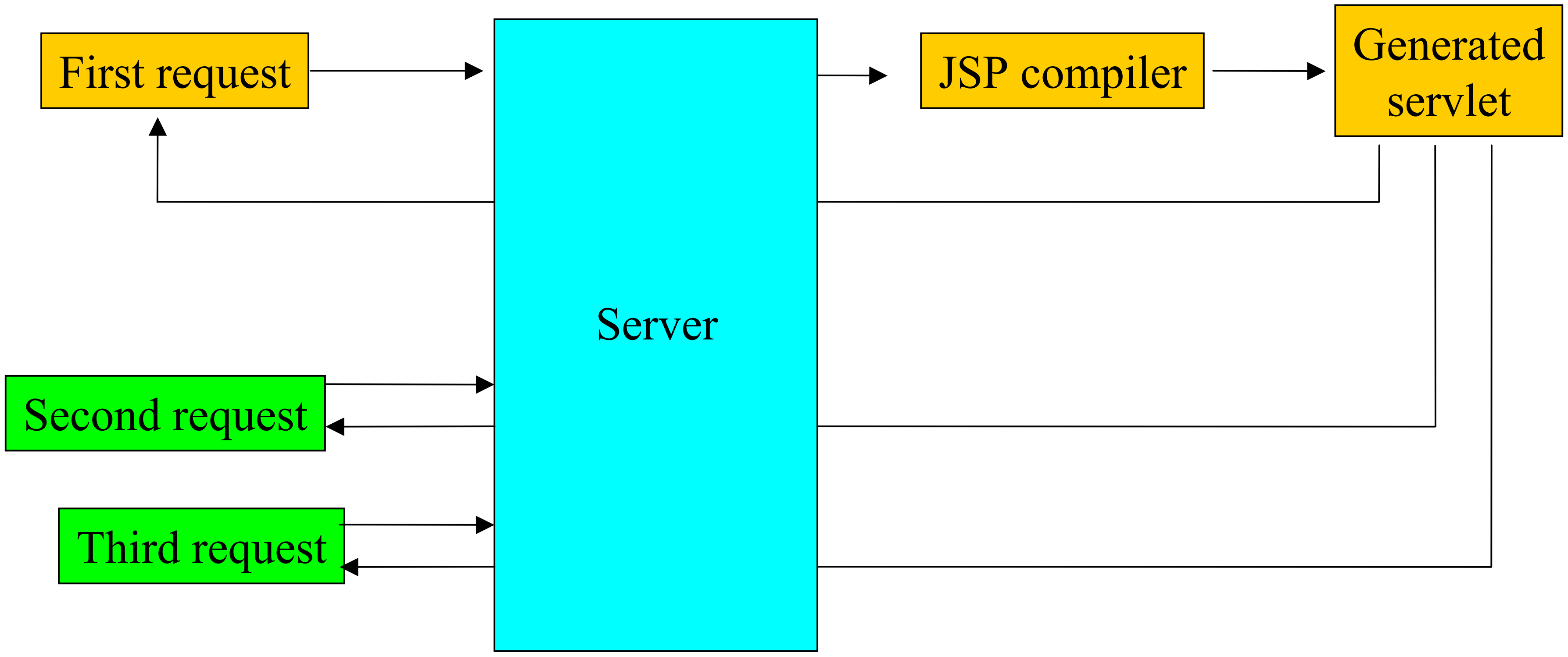
Servlet drawbacks

- Servlets are the main technology of J2EE for the development of web applications.
- However, there are several drawbacks from the point of view of the programmers:
  - servlets “merge” the business logic with the presentation;
  - “embedding” (X)HTML code in Java by means of **println** statements is cumbersome;
  - “rapid prototyping” is not possible: every modification requires recompiling the code and/or updating the **web.xml** file, restarting the application or the whole server.

Java Server Pages (JSP)

- JSP is a technology based upon the servlet API, allowing to embed Java code fragments into a (X)HTML template:
  - at the first request the jsp page is translated into a servlet by the container (e.g., Tomcat);
  - then, the servlet is compiled and deployed automatically by the container;
  - finally, the appropriate method of the servlet is run and the resulting output (e.g., a (X)HTML page) is sent to the client.
  - JSP pages are stored in **`jsp`** files.
  - Official site: **`http://java.sun.com/products/jsp/`**

JSP


First request
Server
JSP compiler
Generated servlet
Second request
Third request

A sample JSP page

- Let us write a simple JSP page displaying current the date:

```
<%@ page import="java.util.Date" import="java.text.*"%>
<%! DateFormat formatDate = new SimpleDateFormat("dd/MM/yyyy"); %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3c.org/TR/xhtml1/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/XHTML"
  xml:lang="en" lang="en">
  <head>
    <title>my first JSP page</title>
  </head>
  <body>
    The current date is
    <%= formatDate.format(new Date()) %>.
  </body>
</html>
```

Storing JSP files

- The previous code can be stored in a file named `Date.jsp` in the root directory of a web application managed by Tomcat (e.g., test, used in our previous examples).
- URI: `http://<server-address>:8080/test/Date.jsp`.
- The generated servlet is stored by Tomcat in `$TOMCAT_HOME/work/Catalina/localhost/<nome-utente>/org/apache/jsp`:
  - `Date_jsp.java`
  - `Date_jsp.class`

The generated servlet (I)

```
package org.apache.jsp;
import javax.servlet.*; import javax.servlet.http.*; import
javax.servlet.jsp.*; import java.util.Date; import java.text.*;

public final class Date_jsp extends
org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

 DateFormat formatDate=new SimpleDateFormat("dd/MM/yyyy");
...
 public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws java.io.IOException, ServletException {
    JspFactory _jspxFactory = null;
    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    JspWriter _jspx_out = null;
    PageContext _jspx_page_context = null;
...
```

The generated servlet (II)

```
...
    try {
      _jspxFactory = JspFactory.getDefaultFactory();
      response.setContentType("text/html");
      pageContext = _jspxFactory.getPageContext(this, request,
       response, null, true, 8192, true);
...
      application = pageContext.getServletContext();
      config = pageContext.getServletConfig();
      session = pageContext.getSession();
      out = pageContext.getOut();
...
      out.write('\r');
      out.write('\n');
      out.write("\r\n");
      out.write("<html>\r\n");
      out.write("  <body>\r\n");
      out.write("    The current date is \r\n");
      out.write("    ");
      out.print( formatDate.format(new Date()) );
```

The generated servlet (III)

```
      out.write("\r\n");
      out.write("  </body>\r\n");
      out.write("</html>\r\n");
    } catch (Throwable t) {
      if (!(t instanceof SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
          out.clearBuffer();
        if (_jspx_page_context != null)
          _jspx_page_context.handlePageException(t);
      }
    } finally {
      if (_jspxFactory != null)
        _jspxFactory.releasePageContext(_jspx_page_context);
    }
  }
}
```

Final considerations

- JSP pages are more user-friendly for web-developers:
  - they allow to delegate the translation, compilation and deployment phases to the servlet container;
  - they allow rapid prototyping (modifications to the source code only need to push the refresh button in the browser);
  - embedding Java code fragments in (X)HTML tags is easier than doing the opposite (like in servlets).
- Writing servlets can still be useful:
  - they allow the programmer to protect source code (only class files need to be stored on production servers);
  - servlets can be combined in chains in order to form the building blocks of data transformation sequences;
  - servlets are the natural implementation of tasks which do not need to produce “visible” output.
- Other server-side technologies may differ in many details, but the programming interface always provides the same notions and mechanisms for handling:
  - HTTP requests and responses,
  - form widgets and parameters,
  - cookies and sessions,
  - connections and manipulation of datastores (e.g., file repositories, databases, …).