DS LABS

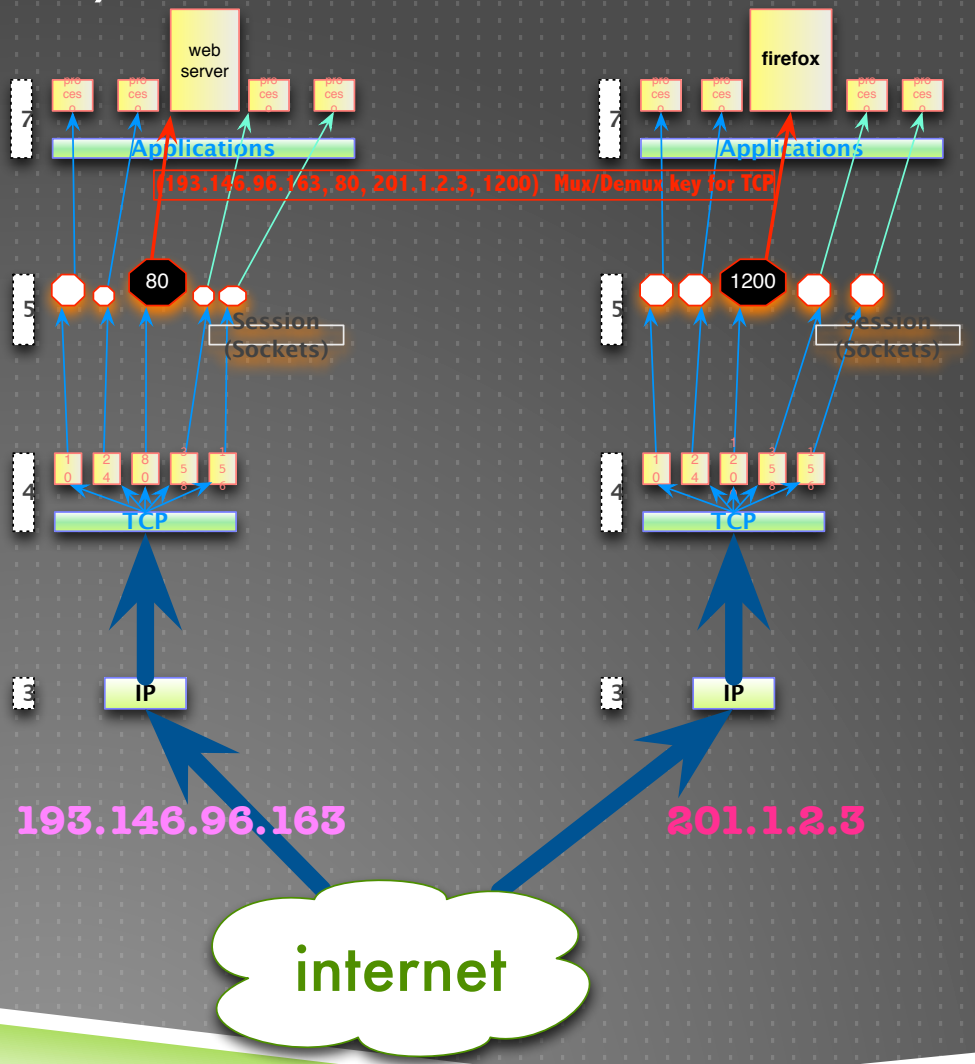
DISTRIBUTED SYSTEMS PRACTICAL EXERCISES

Java Streams and **TCP** Sockets

José María Foces Morán, 2013

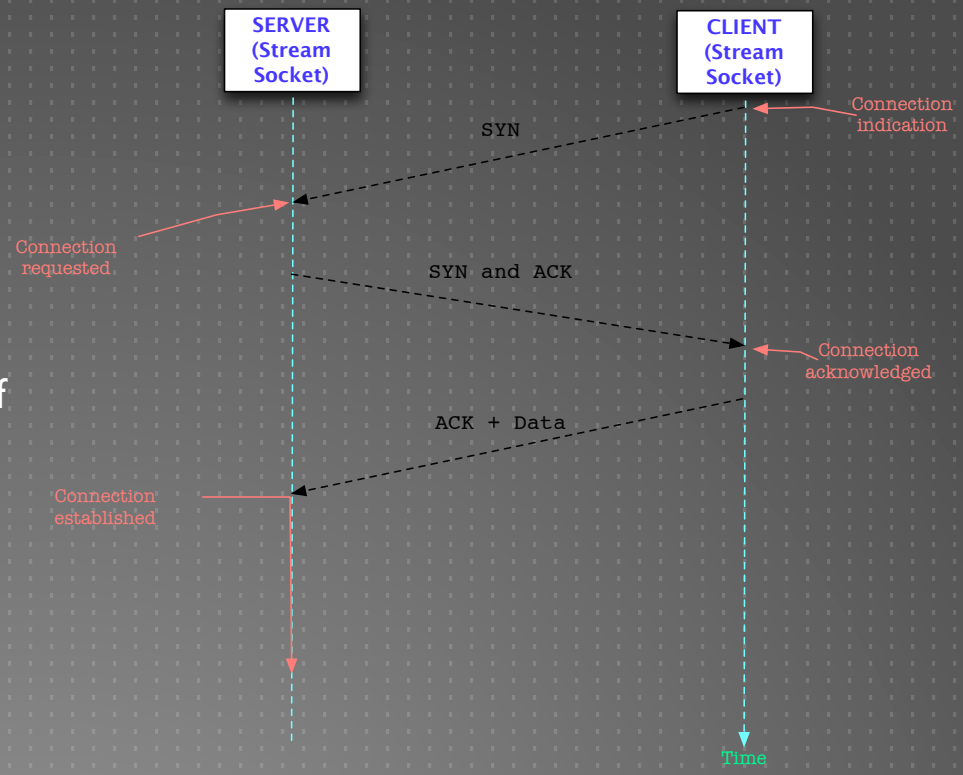
JAVA SOCKETS (TCP)

- Java Sockets are an abstraction of a TCP reliable channel
- Sockets offer a *namespace* for processes and TCP channels
  - A way to identify a process running on a computer in internet
- Socket knowledge is essential for learning RMI (Distributed Objects)



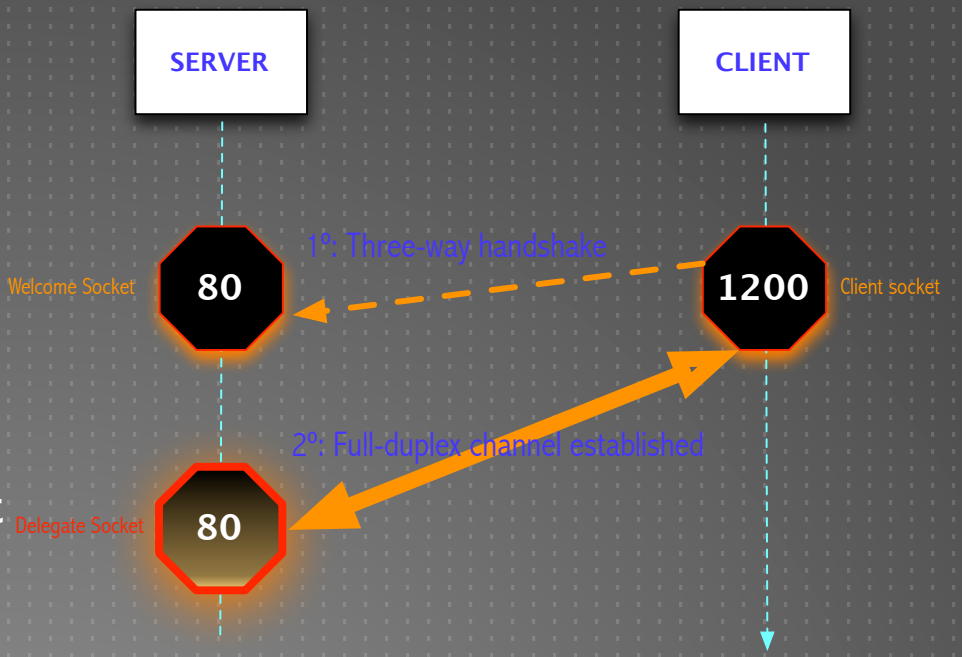
JAVA SOCKETS (TCP)

- Sockets represent a TCP connection but they hide the details regarding the process of connection establishment and teardown
- TCP protocol is virtuallly hidden
  - Except for activation/deactivation of certain features:
  - Nagel's algorithm
- A simple introduction to the Client/Server distributed computing model **(C/S)**



JAVA SOCKET CLIENT/SERVER

- Server creates a ServerSocket on a well known port (80, for example)
  - This is the **Welcome Socket**
  - Welcome Socket Listens *forever* for connection requests from Clients
  - If one arrives, executes 3-way handshake and creates a **delegate socket** to care for the forthcoming i/o operations
- Client creates a Socket and connects it to server's IP/PORT
  - This Socket is adequate for performing i/o operations



JAVA SOCKETS, EXAMPLES

- **SERVER** creates a ServerSocket and binds it to a well known port

```
try {

    ws = new ServerSocket(port);

} catch (IOException ex) {
    Logger.getLogger(ServerFactory_Type0.c
    System.exit(-1);
}
```

- **CLIENT** creates a Socket and connects it to server's ip/port

```
Socket s = null;

try {

    s = new Socket("192.168.2.101", 50001);

    ObjectOutputStream oos = new ObjectOutputStream(s.getOutputStream());

    oos.writeUTF(COMMAND_UPLOAD);
    oos.flush();
```

- **SERVER** waits for new connection request

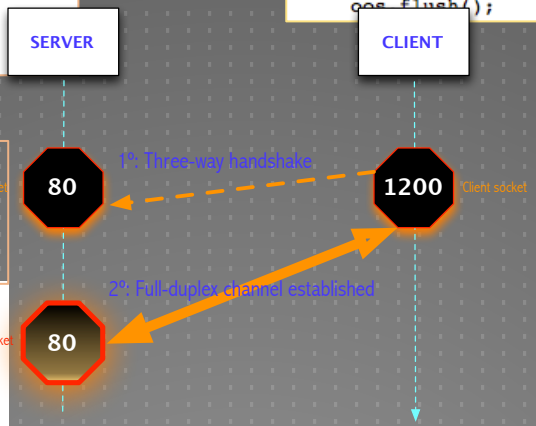
```
while(true){

    try {

        ds = ws.accept();

        //Configure socket options

        // Dispatch a request serve
        rs = new RequestServer(ds);
        rs.run();
```



JAVA STREAMS, ABSTRACT CLASSES

- Streams, an abstraction:
  - Ordered sequence of bytes
  - Storage and retrieval are done sequentially
  - Adequate for almost any external device
- Abstract classes
  - InputStream
    - Methods for reading data and stream navigation
    - OS must allocate resources beyond memory
    - IOException, checked exception (try/catch)
  - OutputStream
    - Methods for writing data…
    - flush(): output streams usually allocate a buffer to store the data being written

JAVA STREAMS LAYERING

- Streams can be **wrapped** in other streams to provide incremental functionality
  - Decorator/Wrapper patterns
- Primitive: Talk to external devices (underlying streams)
  - FileInputStream / FileOutputStream
  - ObjectInputStream / ObjectOutputStream
- Intermediate streams: Wrap around already existing streams
  - If you close a stream that encloses a socket, close() and flush() propagate to sockets
  - DataInputStream / DataOutputStream (Binary, byte streams)
  - Readers / Writers (Unicode characters and strings)
- Other possibilities:
  - Classes for buffered streams
  - Compressed streams
  - Others

JAVA STREAMS LAYERING

The OutputStream of Socket s gets **wrapped** into an ObjectOutputStream instance whose name is oos

```
Socket s = null;

try {

    s = new Socket("192.168.2.101", 50001);

    ObjectOutputStream oos = new ObjectOutputStream(s.getOutputStream());

    oos.writeUTF(COMMAND_UPLOAD);
    oos.flush();
```

Now, we use **oos** to transmit data much more easily than with **s**

OBJECT OUTPUT STREAM

- Java ObjectInputStream() and ObjectOutputStream() classes serve for transmitting Java objects directly in a seamlessly manner over these streams
- When Java transmits an object it send a series of ordered bytes over a stream, that ordered sequence, upon reception is deserialized and trnasformed into **a copy** of the original object in the **addressing space** of the receiving JVM
  - Object serialization example:

```
Socket s = null;

try {

    s = new Socket("192.168.2.101", 50001);

    ObjectOutputStream oos = new ObjectOutputStream(s.getOutputStream());

    oos.writeUTF(COMMAND_UPLOAD);
    oos.flush();
```

TWO-SESSION LABORATORY EXERCISE

- Design and build a simple sockets-based File Service in Java
  - Server listens on **TCP** port 50001
  - Serves Clients one by one, serially
  - Main protocol functions for now –we will extend it
    - File upload
    - File download
- Exercise consists of completing and adapting the provided software
  - Today:
    - Get familiar with the software provided
    - Study the *file upload command* then
    - Implement the *file download command* by using FileInputStream and FileOutputStream in CommandDispatcher.java
  - Next Monday:
    - Provide a **multithreaded** implementation of the server
    - Study the advantages of multithreading, try to estimate the server's throughput increase

BASIC LABORATORY EXERCISE (TODAY)

- Download source source code from:
  - paloalto.unileon.es/asd/asdfileservice.zip
- Setup and Compile project according to java package name (asdfileservice.server and asdfileservice.client):
- Run server with parameters (port and 0)
  - $ java asdfileservice.server.FileServerDriver 50001 0
- Then, extend UploadClient.java so such that it retrieves the server IP and port from the command line, then run UploadClient, which, will connect and send file /tmp/anyfile to the server
  - Server will honor this request (Observe the peer-to-peer protocol messages interchanged between cleint and server)
- Now, your task consists of writing the file download method
  - FileOutputStream, new file on server

ADVANCED LABORATORY EXERCISE (NEXT SESSION)

- Obtain a multithreaded implementation of the server
- Design an experimental setup to compare the throughput in single-threaded vs. Throughput multithreaded