EBU6501 - Middleware
Week 4, Day 3 & 4: Client and Server Middleware

Gokop Goteng & Ethan Lau

Original version created by: Dr. Gokop Goteng

Lecture Aim and Outcome

Aim

- » The aim of this lecture is to learn client/server sides application

Outcome

- » At the end of this lecture students should be able to:
  - □ Write a program that communicates over a network
  - □ Write a program for an application that runs on the client side and remote server
  - □ Recap on HTTP protocol and to relate into in the HTTP header

Lecture Outline

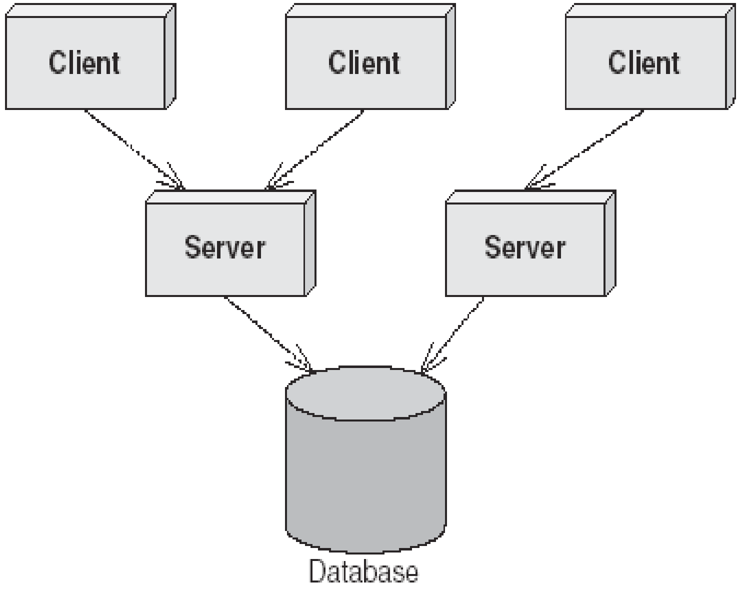
- 3 Tier systems
- Different remote communication models
- WRITE programs to communicate over a network
- Reading characters
- Creating client side socket
- Read from socket to client
- Write from client to socket
- Creating a Server Socket
- Accept method
- HTTP communication & HTTP request
- HTTP Header

Lecture Outline

- Response message
- WWW-Authenticate
- Content-Length
- Location
- Server code
- A simple client server program
- Standards for communicating with remote objects
- RMIC_RMI Compiler
- Stub & Skeleton
- Middleware – application component, interfaces

Many distributed business applications have 3 tier model

- A stock trading system
- A banking application
- A customer call centre
- A procurement system
- An insurance risk analysis application



Standard multitier deployment.

E.g. A browser client , eg http

The server can use e.g. servlets/JSPs/php

Databse can use eg JDBC (Java Database Connectivity) to talk to DB

Three-tier systems attempt to mitigate common bottlenecks

- by managing back-end resources more effectively.
  - » through resource management techniques like **pooling** and **clustering of middle-tier servers**.
- Also offer security models in the middleware
- Dealt with in Information Systems Management

Beyond 3 Tier: Interacting systems

- Sometimes **different applications need to interact differently**
  - » peer to peer communication
    - Either application can initiate a communication
  - » Publish subscribe (built on e.g. AMQP- Advanced Message Queuing Protocol)
    - – Client and subscriber are decoupled
      - They do not need to know who they are
    - – The MOM (Message-Oriented Middleware) handles the interaction
- All can be built using the techniques in this course

Different Remote communication Models

- Client Server using **TCP/ UDP** (Transmission Control Protocol/User Datagram Protocol)
- **RMI** (Remote Method Invocation)
  - Synchronous + serialized data + language restricted (Java)
- **Client and web server communication using HTTP**
  - Multiple synchronous calls + asynchronous
- **Web Services**
  - Data structures passed over the wire using universally agreed schemas supporting mixed language
- **Publish Subscribe Message Oriented Middleware**
  - Asynchronous
  - Delivery assurances

Want to WRITE programs that communicate over a network

- Not write just a server process using http
- Look at
  - How to **write a client** to **send** information to another program
  - How to **write a program** that **listens**, i.e. a **server**.
  - How to make a **server multithreaded**
  - Understand why a **client** can **benefit** from being **multithreaded**
  - How to communicate between peer processes

Network components

```
Socket socket;
BufferedReader inStream;
PrintWriter outStream;
```

Input from and Output to a File: First create the streams

```
FileInputStream theInput=null;
FileOutputStream theOutput=null;
try{
theInput= new FileInputStream (fromFile);
theOutput= new FileOutputStream (toFile); }
catch(FileNotFoundException fnf){
System.out.println(“Couldn’t open file ”+fnf);
System.exit(1);}
```

Reading and writing byte by byte

```
int valueRead;
while ((valueRead =theInput.read()) !=-1)
        theOutput.write(valueRead);
```

- this makes the stream theInput read ***a byte*** at a time
- because the end of file generates -1 you have to use an **int** rather than a **byte type** to **hold the byte that is read**!
- the write method automatically casts the int to a byte and writes a byte

Faster input and output

BufferedInputStream & BufferedOutputStream

- A BufferedInputStream *manages* an input stream and so produces a different behaviour
  - on creation it takes the stream *it manages* as an input parameter
  - e.g. a FileInputStream
- a read of a BufferedInputStream will read as much data as possible *even before the program requests it*
  - so when it is requested access is much quicker
  - *changes* byte by byte read on request to *reading from a buffer of bytes*

Use read() as Before

```
FileInputStream theInputFile =  new
   FileInputStream(theFileName);
 BufferedInputStream theBufferedFile =  new
   BufferedInputStream (theInputFile );
```

- use read() as before
  - » while ((theBufferedFile.read()!=-1){……}
  - » but you may be reading from the buffer (with luck)

Take care when buffering output

- Sometimes we need to **make sure** that output which is buffered (i.e. queued) has been **delivered before** other **processing** can **proceed**.
  - » e.g. when you print a prompt on the screen asking for input
    - – A subsequent read might be performed before the output has appeared
- theBufferedOutput.flush()
  - » forces any **buffered output bytes to be written out to the underlying output stream**
  - » suspends processing till the output is actually delivered
- still use write

Reading **Characters**

```
BufferedReader theInput=null;
BufferedWriter theOutput=null;
try{
FileReader inputFS=new FileReader(fromFile);
theInput=new BufferedReader(inputFS);
FileWriter outputFS=new FileWriter(toFile);
theOutput=new BufferedWriter(outputFS);
}……
```

Can read lines if use characters

A “line” is terminated by a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed

```
String line;

while( (line=theInput.readLine()) !=null)
   {System.out.println(“Read :” +line);
   theOutput.write(line,0, line.length());
   theOutput.newLine();
                          //uses platform’s definition of a new line
   }
```

Creating Client Side Socket

```
Socket socket;
try {socket = new Socket("localhost", 8205);
```

InetAddress of remote machine

loopback device, so as can run without a network card installed

same as 127.0.0.1

Creates a client-side type socket and connects it to the specified port number at the specified IP address, i.e. the server

Creating the instance creates the connection

(Remember that the server is already “listening”)

Options: e.g. setSoTimeout()

- Sets the time in milliseconds that the socket **should block waiting for data**
  - SO_TIMEOUT is set
- InterruptedIOException generated if nothing received in the time
  - socket still exists and connection still exists
- Use, e.g. on an HTTP connection while waiting for a response from a server
  - if nothing received in the set time notify the user
  - need to set SO_TIMEOUT before stream blocks for a read() !

To read from socket to client

- **Since in this application we are reading and writing text use a stream that has text handling capability**

```
inStream = new BufferedReader(
new
   InputStreamReader(socket.getInputStream()));
```

Returns an input stream for reading characters FROM this socket.

To write from client to socket

```
outStream = new PrintWriter(
                    new  OutputStreamWriter(
                                            socket.getOutputStream()));
```

Returns: an output stream for writing characters TO this socket.

```
    }
catch(Exception exc)
    {System.out.println("Error! - " + exc.toString());}
}
```

Take care as output buffered!

- Can **call write** but the data is not sent till the **buffer is full**
- So ***sometimes nothing will be sent*** to the socket (and so to the server) even though you have called write
- need to call flush()
  - to force data in the buffer to be written to the socket

Setting autoflush by using a different constructor

new PrintWriter (socket.getOutputStream(), true)

- Now every time you call *println()* it automatically sends the data

Creating a ServerSocket

```
try{ServerSocket serverSocket = new
                 ServerSocket(8205);
System.out.println("Server Ready");
```

Now create an infinite loop listening

```
while(true){
         Socket incoming = serverSocket.accept();
```

Listens for a connection to be made to serverSocket and accepts it. The method blocks until a connection is made.

```
         System.out.println("Client Connected");
```

Accept method

- When the ServerSocket **receives** the **accept method it waits** until **a client starts up and requests a connection** on the port it is listening to
  - the *listening* port number is known to the client
- When **connection successfully established**
  - returns a socket object which is bound *to a new local port which is different from the port it was (or is still) listening to for connections*
  - the server communicates with the client over this new socket *so server can continue to listen on original port through the ServerSocket*

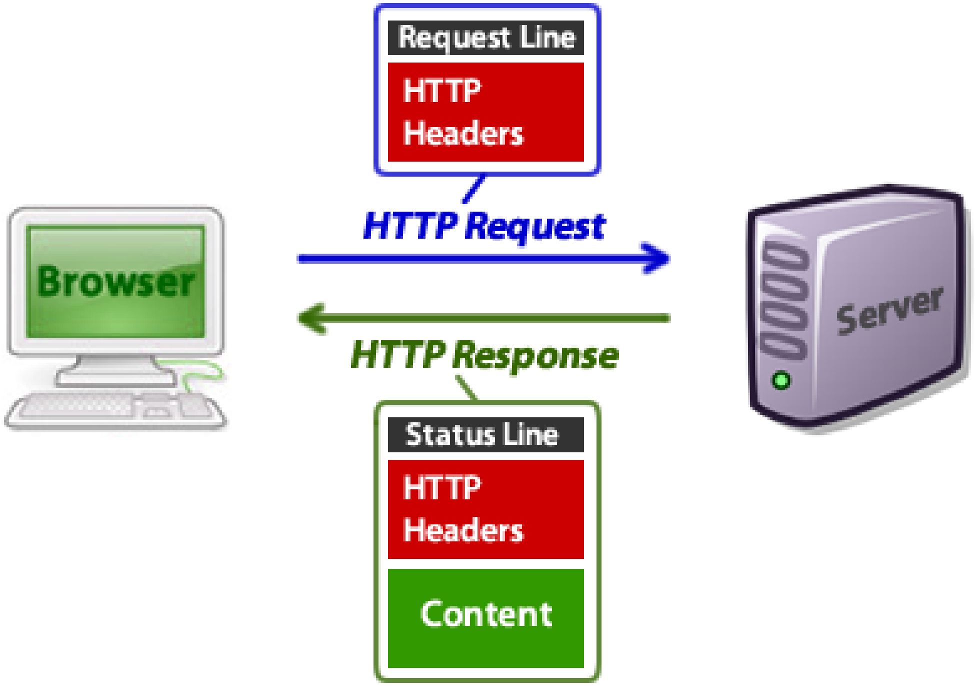
A *very* simple Web server, i.e. simple http server

- Handles only one HTTP request (!!!!!)
  - request is in form GET path&file_name HTTP/1.0
- Accepts and parses the HTTP request
- Gets the requested file from the server's file system
- Creates an HTTP response message consisting of header lines and the requested file.
- Sends the response directly to the client
  - i.e. browser

A quick reminder about HTTP

- You need to know the **HTTP protocol**
  - » Quite well!
- Taught in other modules
- A few slides to help for what is needed here

HTTP Communication



An HTTP GET message request

GET /somedir/index.html HTTP/1.1

Host: www.chamelion.elec.qmw.ac.uk

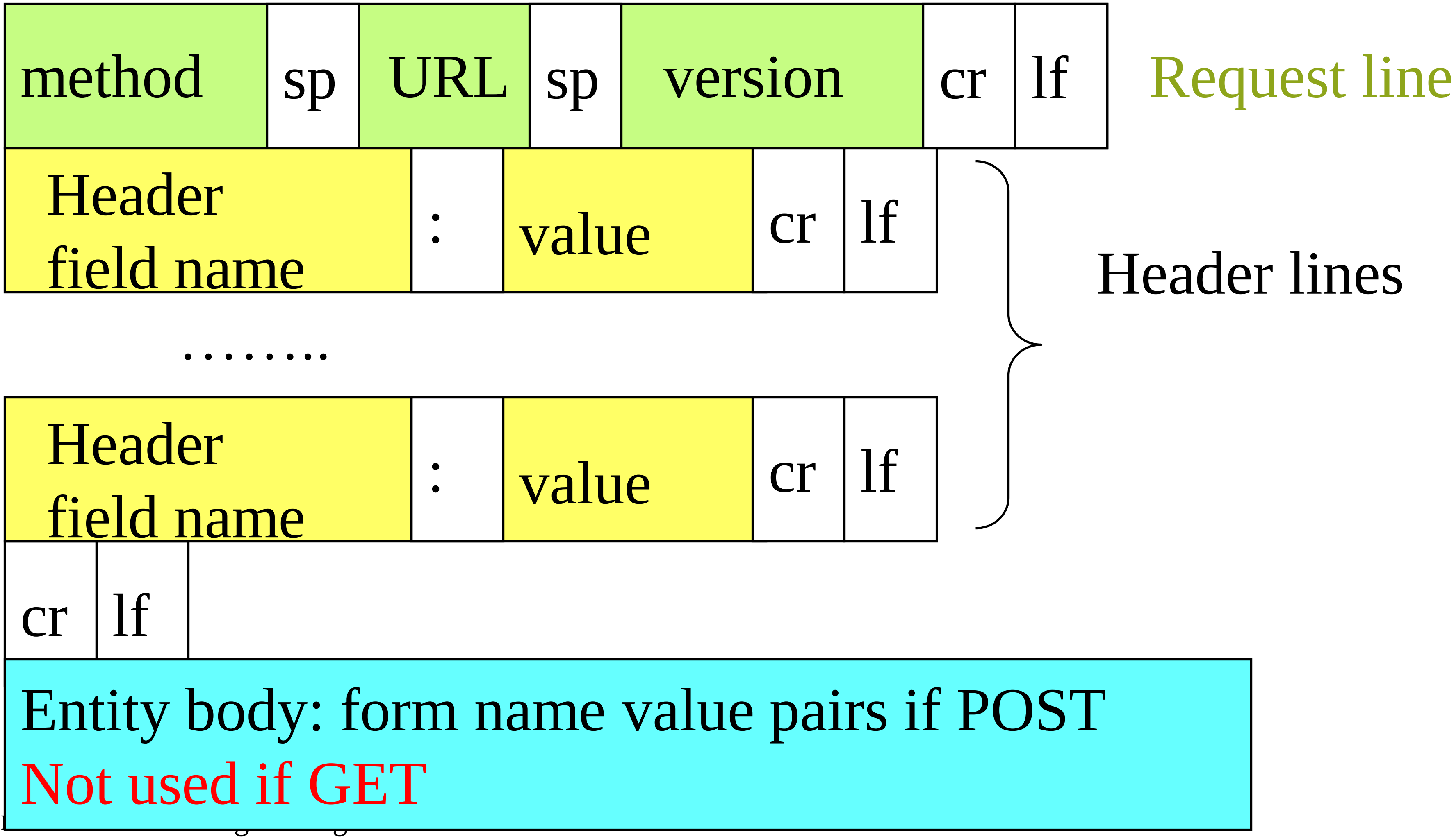
Connection: close

User-agent: Mozilla/4.0

Accept-language:fr

extra carriage return and line feed

HTTP request message format



method sp URL sp version cr lf

Request line

Header field name : value cr lf

……..

Header field name : value cr lf

Header lines

cr lf

Entity body: form name value pairs if POST

Not used if GET

Example request headings: Accept-Encoding

```
Accept-Encoding: gzip,deflate
```

- Most modern browsers support gzip, and will send this in the header.
- The web server then can send the HTML output in a **compressed format**. This can reduce the size by up to 80% to save bandwidth and time.

Accept-Language

```
Accept-Language: en-us,en;q=0.5
```

- This header displays the default language setting of the user. If a website has different language versions, it can redirect a new surfer based on this data.
- ***It can carry multiple languages, separated by commas. The first one is the preferred language, and each other listed language can carry a "q" value, which is an estimate of the user's preference for the language (min. 0 max. 1).***

Host

- An **HTTP Request** is sent to a specific IP Addresses.
- But since most servers are capable of hosting multiple websites under the same IP, they must know which domain name the browser is looking for.

```
Host: net.tutsplus.com
```

- This is basically the host name, including the domain and the subdomain.

User-Agent

```
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1.5) Gecko/20091102 Firefox/3.5.5 (.NET CLR 3.5.30729)
```

- This header can carry several pieces of information such as:
  - Browser name and version.
  - Operating System name and version.
  - Default language.
- This is how websites can collect certain general information about their surfers' systems. E.g. they can detect if the surfer is using a cell phone browser and redirect them to a mobile version of their website that works better with low resolutions.

If-Modified-Since

- If a web document is already cached in your browser, and you visit it again, your browser can check if the document has been updated by sending this:

```
If-Modified-Since: Sat, 28 Nov 2009
06:38:19 GMT
```

- If it was not modified since that date, the server will send a "304 Not Modified" response code, and no content - and the browser will load the content from the cache.

Referer Header: example

- when a user clicks a hyperlink in a browser, the browser sends a request to the server corresponding to the destination webpage.
- The request headers ALSO includes the referer header,
  - which indicates the last page the user was on
  - **i.e. the one where they clicked the link**
- For example, if I visit the Nettuts+ homepage, and click on an article link, this header is sent to my browser:

```
Referer: http://net.tu
```

HTTP Referer header contains the referring url.

- Your browser will let a site know which site it visited last.
- The "Referer" header is frequently considered a privacy concern.
  - **If the site was coded carelessly**, your browser may communicate sensitive information (session tokens, usernames/passwords and other input sent as part of the URL).

Authorization

- **When a web page asks for authorization,** the browser opens a login window. When you enter a username and password in this window, the browser sends another HTTP request, but this time it contains this header.

```
Authorization: Basic bXl1c2VyOm15cGFzcw==
```

- The data inside the header is base64 encoded.

  E.g., base64_decode('bXl1c2VyOm15cGFzcw==') would return 'myuser:mypass'

A response message

Server is going to close the TCP connection

HTTP/1.1 200 OK
Connection : close
Date: Fri, 17th September 2010 12:01:14 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Thur, 16 September 2010 08:44:01 GMT
Content-Length: 5993
Content-Type: text/html

Request succeeded and the info. is in the response

Important for caching (both client and network)
Conditional GET

The official indicator of type, not the file extension

Blank line

data data data…

entity body - could be a page to display

The status line

- The first piece of data is the protocol. This is again usually `HTTP/1.x or HTTP/1.1` on modern servers.
- The next part is the status code followed by a short message. `Code 200` means that our **GET request** was successful and the server will return the contents of the requested document, right after the headers.

Status code

- If the **GET request** would be made for a path that the server **cannot find**, it would respond with a **404** instead of 200.
- **HTTP Status Codes**
  - » 200's are used for successful requests.
  - » 300's are for redirections.
  - » 400's are used if there was a problem with the request.
  - » 500's are used if there was a problem with the server.

WWW-Authenticate

- A website may **send** this **header** to **authenticate a user through HTTP**. When the browser sees this header, it will open up a login dialogue window.

```
WWW-Authenticate: Basic realm="Restricted Area"
```

Content-Length

- When content is going to be transmitted to the browser, the server can **indicate the size** of it (in bytes) using this header.

```
Content-Length: 89123
```

- This is especially **useful** for **file downloads**. That's how the **browser can determine the progress of the download**.

Location

- This header is used for redirections.
- If the response code is 301 or 302, the server must also send this header.
- For example, when you go to http://www.net.tutsplus.com your browser will receive this:

```
HTTP/1.x 301 Moved Permanently

...

Location: http://net.tutsplus.com/
```

Examples of requests from the browser

We will make server listen on port 6666

`http://localhost:6666/myfile.html`

```
where file.html is in same directory as
  the server class
```

```
or
```

`http://localhost:6666/nep3/background.gif`

```
where nep3 is a directory inside the
  directory where the server is running
```

- Before making a request must set the simple server running

Structure of code

```
public class SimpleWebServer
{String requestMessageLine;
         ….  .
 SimpleWebServer(){……}

 public static void main(String args[]){
          SimpleWebServer sws=new
                      SimpleWebServer();
               }
}
```

The server code

```
import java.net.*;
import java.io.*;
import java.util.*;

public class SimpleWebServer
{String requestMessageLine;
 String fileName;
     // to hold name of file requested
 ServerSocket listenSocket;
 Socket connectionSocket;
```

Server Code…

```
SimpleWebServer(){try
   {listenSocket= new ServerSocket(6666);
   System.out.println("Server Ready");
   connectionSocket= listenSocket.accept();
// set up input from socket
   BufferedReader in =new BufferedReader(
   new
   InputStreamReader(connectionSocket.getInputStream()));
// set up output to socket
   DataOutputStream out = new
   DataOutputStream(connectionSocket.getOutputStream());
```

Server Code…

```
requestMessageLine = in.readLine();
StringTokenizer tl = new
           StringTokenizer(requestMessageLine);
if(tl.nextToken().equals("GET")){
   fileName = tl.nextToken();
   if(fileName.startsWith("/")==true)
               fileName=fileName.substring(1);
   File file = new File(fileName);
                    //so as to find length
   int numOfBytes = (int)file.length();
   FileInputStream inFile = new
                    FileInputStream(fileName);
```

Server Code…

```
byte[] fileInBytes= new byte[numOfBytes];
inFile.read(fileInBytes);
out.writeBytes("HTTP/1.0 200 Document Follows\r\n");
if(fileName.endsWith(".jpg")) out.writeBytes("Content-
  Type: image/jpeg\r\n");
if(fileName.endsWith(".gif")) out.writeBytes("Content-
  Type: image/gif\r\n");
  out.writeBytes("Content-Length: " + numOfBytes +
  "\r\n");
  out.writeBytes("\r\n");
  out.write(fileInBytes,0,numOfBytes);
  connectionSocket.close();
        } else System.out.println("Bad Request
  Message");
}
```

Server Code…

```
catch(Exception
  exc){System.out.println("Error! - " +
  exc.toString());}

}// end of constructor


public static void main(String args[]){
SimpleWebServer sws=new
  SimpleWebServer(); }//end of main
}//class definition
```

A very simple client server program

- Server returns the sentence sent by the client in upper case

A simple client

```
class TCPClient {  public static void main(String argv[]) throws Exception  {
String sentence;   String modifiedSentence;
BufferedReader inFromUser = new BufferedReader( new InputStreamReader(System.in));
 Socket clientSocket = new Socket("localhost", 6789);
DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());
BufferedReader inFromServer = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
 sentence = inFromUser.readLine();
outToServer.writeBytes(sentence + '\n');
modifiedSentence = inFromServer.readLine();  //blocks till reply comes - synchronous
System.out.println("FROM SERVER: " + modifiedSentence);
clientSocket.close();  } }
```

A simple server: listening

```
class TCPServer {    public static void main(String argv[]) throws Exception      {
String clientSentence;
String capitalizedSentence;
ServerSocket welcomeSocket = new ServerSocket(6789);
while(true)
{Socket connectionSocket = welcomeSocket.accept();
BufferedReader inFromClient =   new BufferedReader(new
InputStreamReader(connectionSocket.getInputStream()));
DataOutputStream outToClient = new
DataOutputStream(connectionSocket.getOutputStream());
clientSentence = inFromClient.readLine();
 capitalizedSentence = clientSentence.toUpperCase()+'\n';    // If long, blue lines would be in a thread
//and we would return immediately to the start of the while loop
 outToClient.writeBytes(capitalizedSentence);
}      } }
```

The server typically spawns threads to manage its TCP queue better

Standards for communicating with remote *objects*

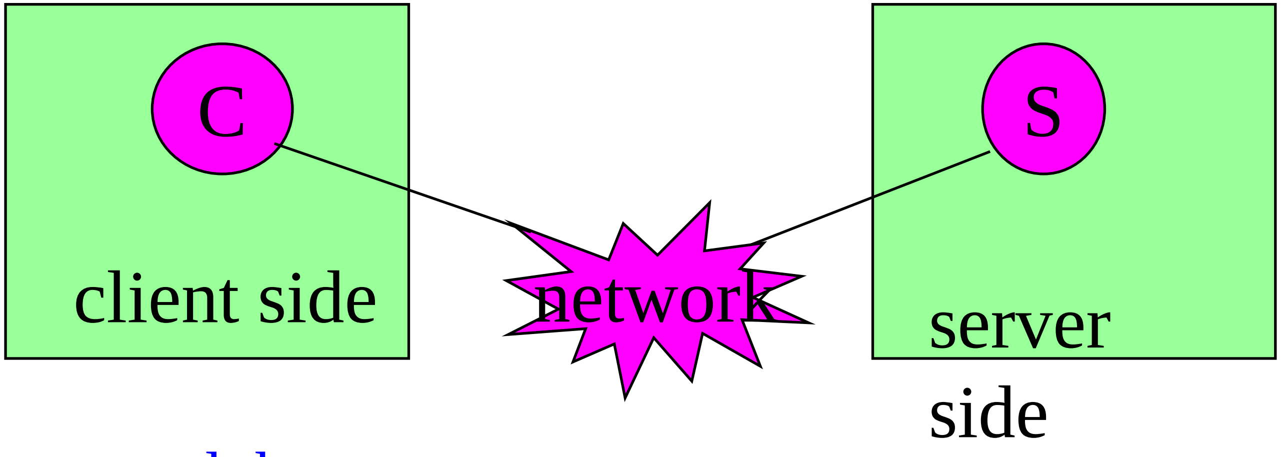
- **CORBA - common object request broker**
  - language neutral
    - e.g. client and server can be in different languages
    - services are described in an Interface Definition Language (IDL)
      - standardised
    - The IDL is mapped into implementation language
      - libraries exist for most languages
- **DCOM - distributed component object model**
  - proprietary
- **RMI - Remote Method Invocation**

RMI

- Limited to one language & currently incompatible with CORBA, **but**…...
- Can **pass object instances between remote objects**
  - » as parameters or return values
    - – c.f. CORBA restricted to primitive types or arrays of them
- Do not need to **know how object gets to remote object method**
  - » implementation uses sockets but we **do not need to set them up**
  - » object serialization **automatically creates a sequence of characters** to be sent via the sockets but we **do not need to compose or parse a string**

RMI Architecture

- Object C wants to call a method provided by Object S.
  - Because the object S is on a different machine (or more to the point on a different JVM heap) its method cannot be invoked directly.



- In RMI can pretend that we can
- Definition S is a *remote object* if it *implements* a *remote interface.* (that we define)

RMIC-RMI Compiler

- You define your server classes as usual and compile using **javac**
  - » but you also need *to set up the mechanism* for accessing the server methods
- RMIC takes **the server side implementation of the code** and produces two extra files called
  - » **skeleton** that you **locate at the server**
  - » **stub** that you **locate at the client**
  - » these files contain low level networking code
    - – All that is needed

RMI : what is a stub & skeleton?

- **Stub** is a **client** that **invokes a TCP connection** to the **skeleton**
- The **skeleton** is a **server listening** for **TCP requests**
- The client hangs waiting for the response
  - » This is **synchronous** communication
- Note that the skeleton usually does not exist as a separate entity. Its function exists always though. We treat as separate for clarity of roles

The **stub** must:

- present the same (remote) interface as S, so from the perspective of C the stub is equivalent to S.
- work with JVM1 & RMI system on machine 1 to serialize any arguments to the method calls and send this information to machine 2
  - can be another JVM on same machine
- receive any results from the remote invocation of the method, deserialize it, and return this to C

The **skeleton** should:

- be able to receive remote method calls for S and their parameters, and with the JVM and RMI system on machine 2 deserialize them.
- Invoke the appropriate method on S with these arguments
- Receive any returned value from the method call, and with the JVM and RMI system on machine 2, serialize them and send this information back to machine 1.

Issues in enterprise computing

- **In a distributed application issues to be considered are:**
- **Load balancing.**
  - » Clients must be directed to the server with the lightest load.
    - – If a server is overloaded, a different server should be chosen.
- **Back-end integration.**
  - » Need to persist business data into databases as well as integrate with legacy systems that may already exist

Still more issues…

- **Threading**
  - Have many clients connecting to a server
    - This means the **server must be multi-threaded**.
  - **How many threads can you support**?
    - Should you be i**nterweaving**?

- **Security.**
  - The servers and databases need to be **shielded from saboteurs**.
  - Known users must be allowed to **perform only operations** that they **have rights to perform**.

Finally....

- **Logging and auditing**.
  - » If something goes **wrong**, is there **a log** that we can **consult** to **determine** the **cause** of the **problem**?
    - – A log would help us debug the problem so it doesn’t happen again.
- **Systems Management.**
  - » Need **monitoring software** to **page a system administrator** if a **catastrophic failure occurs**.

Middleware is the name for these services

- needed in any business problem
- Each of these services **requires a lot of thought**, a lot of experience, and often a lot of work to resolve

High-end middleware is very complicated to build and maintain

- requires expert-level knowledge, and
- is completely orthogonal to most companies’ core business.

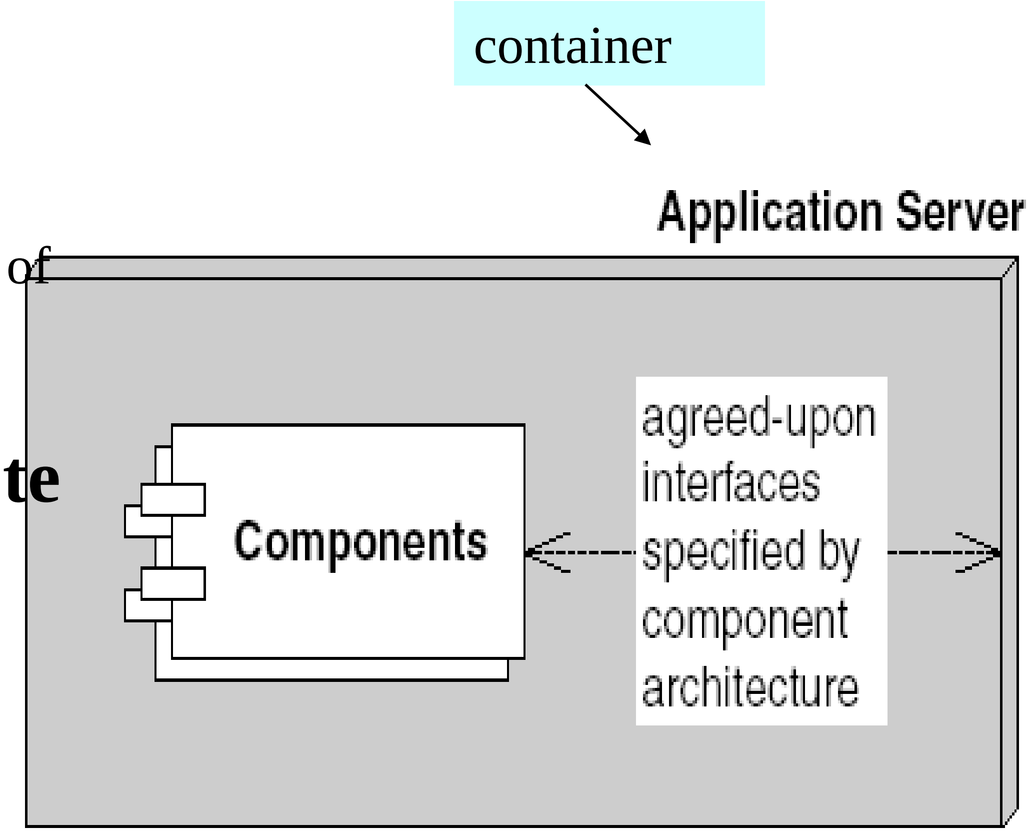
- Buy instead of build?
  - Roman…. these days, companies that build their own middleware risk setting themselves up for failure.

Middleware uses application **components**

- DEFINITION: a component is code that implements a set of well-defined interfaces used by the container.
  - It encodes a manageable, discrete chunk of logic.
- need to build your application out of *components using the interface required by the middleware*

In a component *architecture*…….

- components are **not entire** applications
  - they **cannot run alone**.
  - components to be switched in and out of different application servers
    - without having to change code
- **reusable components promote rapid application development.**



There are two ways you could use interfaces

- You provide everything
- You can write the interface definition and classes that implement the interface.
  - » Then anyone can just call the methods
- But might just as well have written the classes!
  - » So interface redundant
- **Not** the style of programming if you are a middleware provider

Other Way: How we use interfaces

- When you as a container provider (middleware provider) e.g. provide **part of a big system**
  - You write classes for your part
  - e.g. UI for Swing, Tomcat server
- You provide interface definitions for what others have to implement.
  - Others (**the application developers**) write the implementation classes for these
  - E.g. what to do if a button is clicked
  - You cannot do this as you are not the end application developer. The others use the system (you are the middleware provider)

Developers of containers use interfaces a lot

- **They assume that users** of the container (the application developers) will implement these interface signature methods
- The container is in control – it is where the main thread starts.
- The user provided methods (for THEIR classes that implement the interfaces)

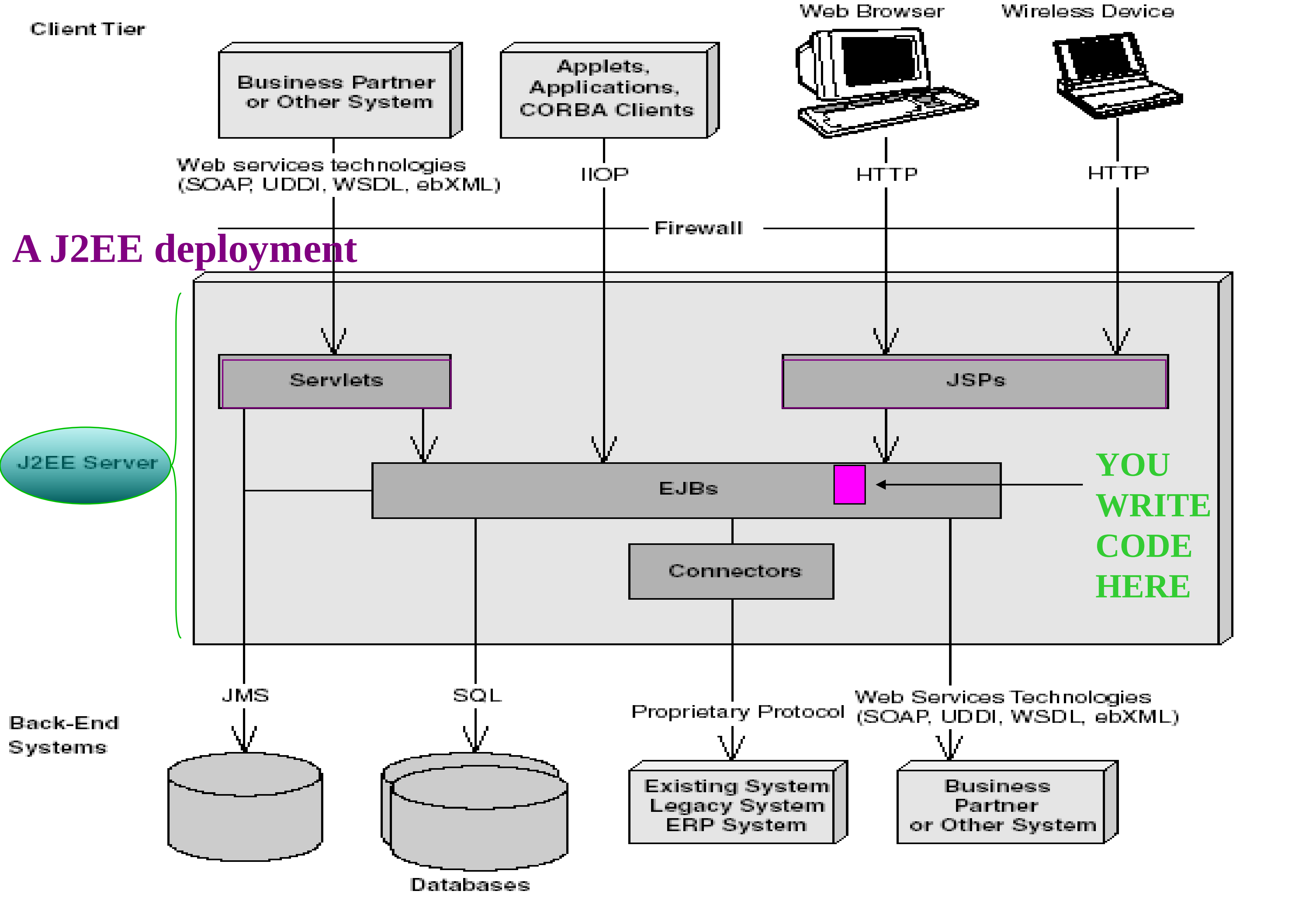
Developers of containers use interfaces a lot

- Many systems, e.g. Android, use interfaces a lot , where the OS (“container”) is issuing commands ,
  - e.g. saying that a WiFi connection has been made

Not only need components but also communication

- Enterprise computing requires inter and intra process communication
  - » Often in a heterogeneous environment
- + security … later
- Look at different models for communication first.

A J2EE deployment



Client Tier

Business Partner or Other System

Applets, Applications, CORBA Clients

Web Browser

Wireless Device

Web services technologies (SOAP, UDDI, WSDL, ebXML)

IIOP

HTTP

HTTP

Firewall

J2EE Server

Servlets

JSPs

EJBs

Connectors

YOU WRITE CODE HERE

JMS

SQL

Proprietary Protocol

Web Services Technologies (SOAP, UDDI, WSDL, ebXML)

Back-End Systems

Databases

Existing System Legacy System ERP System

Business Partner or Other System