Distributed Systems
COMP 212

Lecture 13

Othon Michail

UNIVERSITY OF
LIVERPOOL

HTTP Protocol

- Hypertext Transfer Protocol
- Used to transmit resources on the WWW
  - HTML files, image files, query results, …
  - Identified by Uniform Resource Locators (URLs)
- Uses the client-server model
  - HTTP client: a browser
  - HTTP server: a Web server
- Application layer protocol presuming a reliable underlying transport layer protocol
  - Usually using TCP/IP sockets
- HTTPS: secure version of HTTP

An HTTP 1.0 Protocol

- Request

  <method> <path to file> <HTTP version>

  e.g. GET lecture-notes/index.html HTTP/1.0

- Response

  <HTTP version> <response status> <reason phrase>:

  e.g. HTTP/1.0 200 OK or HTTP/1.0 404 Not Found

  <Date>: e.g. Monday 6 Feb 2017 10:03:49 GMT

  <Content-Type>: e.g. text/html
  <Content-Length>: e.g. 1954 (bytes)
  <server id>
  <blank line>
  <contents of file>

HTTP 1.0 Toy Server Structure

```
public class WebServer {
  public static void main(String args[]) throws Exception{
        < initialisation & listening >
  }
  private static void processRequest(Socket myClient) {
        < read request and reply >
  }
  private static void sendFile(InputStream file, OutputStream out) {
        < copy file to out >
  }
};
```

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

public class WebServer {

  private static ServerSocket initSocket (int port) throws
IOException {
    return new ServerSocket (port);
  };

  public static void main(String args[]) throws Exception {
    if (args.length != 1) {
      System.out.println("Usage: WebServer  <port_number>");
      System.exit(0);
    }

    int port = Integer.parseInt(args[0]);
    System.out.println("Listening on port " + port);

    // Create a ServerSocket to listen on that port.
    ServerSocket ss = null;
    try { ss = initSocket(port);
    } catch (Exception e) {}

    while(true){
      processRequest(ss.accept());
    }
  };
```

```
 private static void processRequest (Socket myClient){
    try {
      // Get input and output streams to talk to the client from the
socket
      BufferedReader in = new BufferedReader(new
                InputStreamReader(myClient.getInputStream()));

      OutputStream out = new
BufferedOutputStream(myClient.getOutputStream());
      PrintWriter pout = new PrintWriter(out,true);

      String request = in.readLine();
      System.out.println("Request: " + request);
      String req = request.substring(4, request.length()-9).trim();
      String path = "html/" + req;
      File f = new File(path);
      if (f.isDirectory()) {
          // if directory, implicitly add 'index.html'
          path = path + "/" + "index.html";
          f = new File(path);
      }
      try {
        InputStream file = new FileInputStream(f);
```

```
        pout.println("HTTP/1.0 200 OK");
        String contenttype =
          URLConnection.guessContentTypeFromName(path);
        System.out.println("Content-Type: " + contenttype);
        if (contenttype != null) {
          pout.println("Content-Type: " + contenttype);
        }
        pout.println("Date: " + new Date());
        pout.println("Server: Toy HTTP Server 1.0");
        pout.println();
        sendFile(file, out); // send raw file
      } catch (FileNotFoundException e) {
          System.out.println("File " + path + " not found");
        }
      out.close();
    } catch (Exception e) { System.err.println(e); }
  };
```

```
 private static void sendFile(InputStream file, OutputStream
out) {
    try {
      byte[] buffer = new byte[1000];
      while (file.available() > 0)
         out.write(buffer, 0, file.read(buffer));
    } catch (IOException e) { System.err.println(e); }
  }
};
```

HTTP 1.0 Toy Server

- Lacks many features
  - HEAD, POST, PUT, DELETE,…
  - Forms, CGI, dynamic features
  - Cookies
  - …
- Lacks security
- Non-persistent communication (defined in 1.1) not supported

- Fully functional Jetty HTTP server
  - http://www.eclipse.org/jetty/
- Tutorial:
  - http://wiki.eclipse.org/Jetty/Tutorial/Embedding_Jetty

Sockets with Java: Summary (1)

Server	Client
1. Create a server socket using the ServerSocket constructor (and catch the exceptions)	
2. Create the socket object that must be used to communicate with the client and use the accept method to assign the correct value to this socket	2. Create a client socket using the Socket constructor (and catch the exceptions)

Sockets with Java: Summary (2)

3. Create the appropriate input and output streams
4. Write the code that communicates with the client
5. Close the input and output streams
6. Close all the sockets

Advantages and Disadvantages

- Main advantage:
  - Sockets are perfect if you want to develop application-specific protocols
  - They are fast and efficient
- Disadvantages:
  - big developing effort if functionality changes
  - application-specific protocols are difficult to maintain
  - do not fit easily into an object-oriented paradigm
  - does not encourage reuse
  - using an application-specific protocol means that the client code can be quite large

Questions

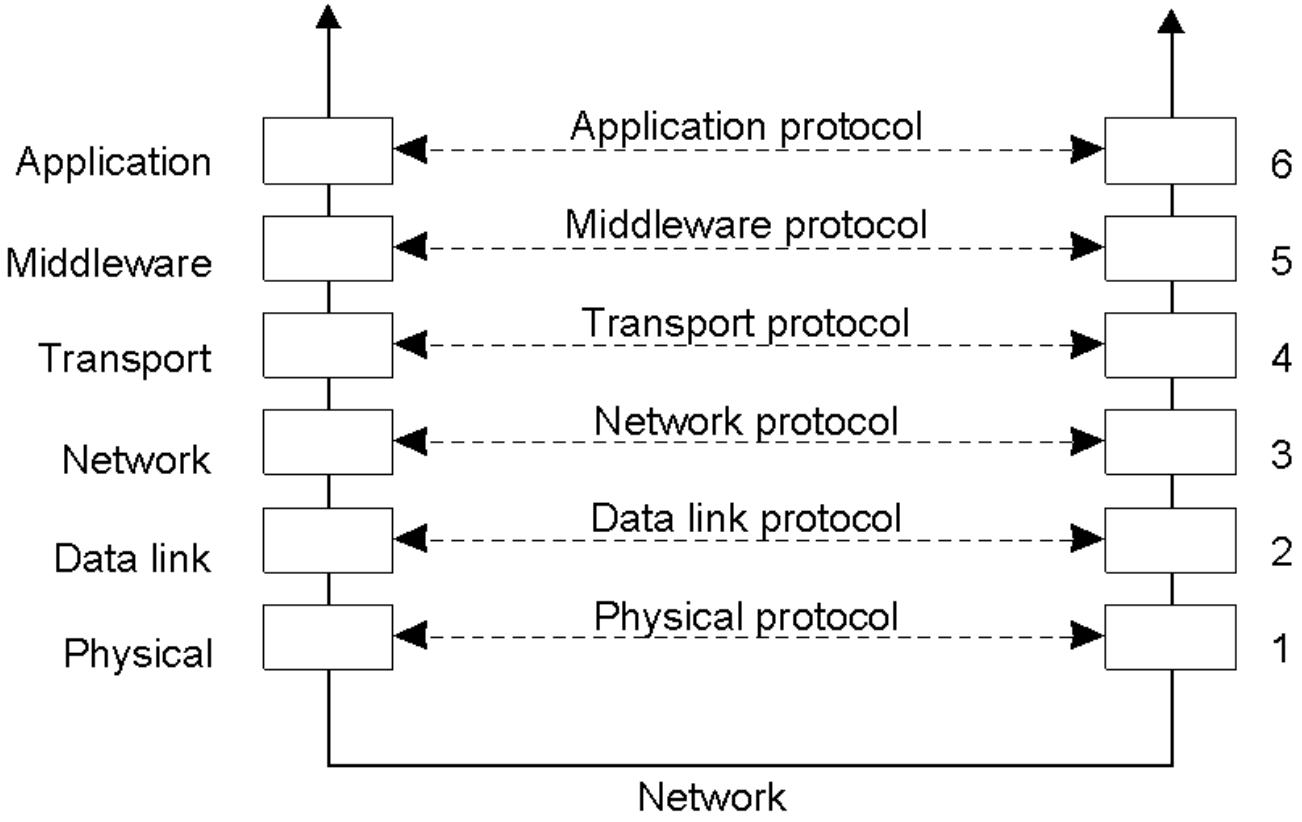
- In this simple example we passed strings
- How to pass an object?
  - Serialisation (converts any object into a string)
- How do we know then what is sent across the network?
  - Integer, String, Object,…?

Middleware Layer

- Middleware has been invented to provide common services and protocols that can be used by a rich set of communication protocols, but which allow different applications to communicate

- Naming protocols, so that different applications can easily share resources
- Security protocols, to allow different applications to communicate in a secure way
- Scaling mechanisms, such as support for replication and Caching

Middleware Protocols



An adapted reference model for networked communication

Middleware Communication

- RPC (remote procedure calls)
- RMI (remote method invocation)
  - Java RMI
- Message-based communication
- Streams

RPC: Remote Procedure Call

- Birrell and Nelson (1984)

"To allow programs to call procedures located on other machines."

- Effectively removing the need for the Distributed Systems programmer to worry about all the details of network programming
  - i.e. no more sockets
- Conceptually simple, but …

Complications: More on RPC

- Two machine architectures may not (or need not) be identical
- Each machine can have a different address space
- How are parameters (of different, possibly very complex, types) passed to/from a remote procedure?
- What happens if one, the other, or both of the machines crash while the procedure is being called?

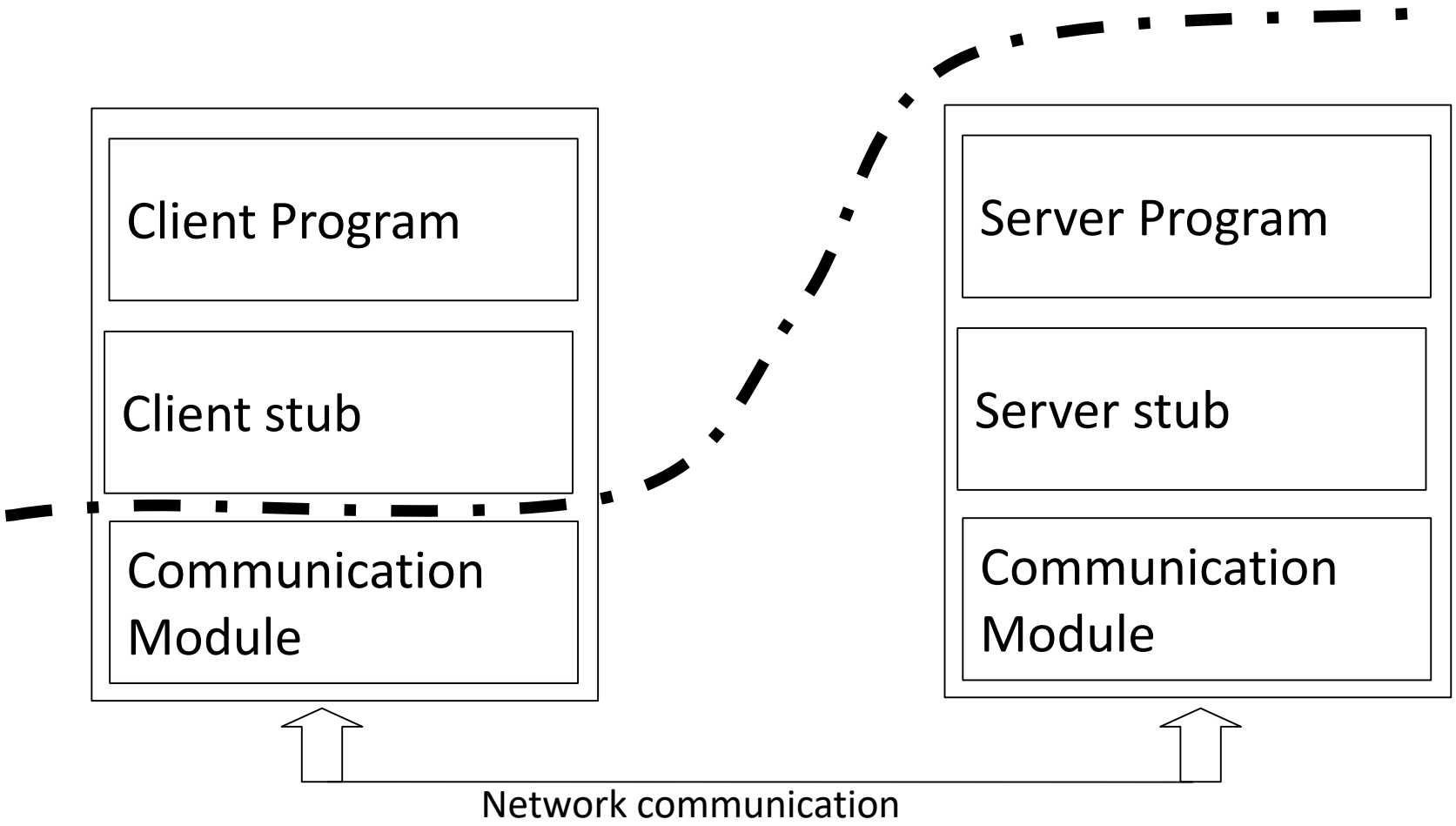
How RPC Works: Part 1

- As far as the programmer is concerned:

  A “remote” procedure call looks and works identically to a “local” procedure call.

- In this was, transparency is achieved

Client and Server "Stubs"



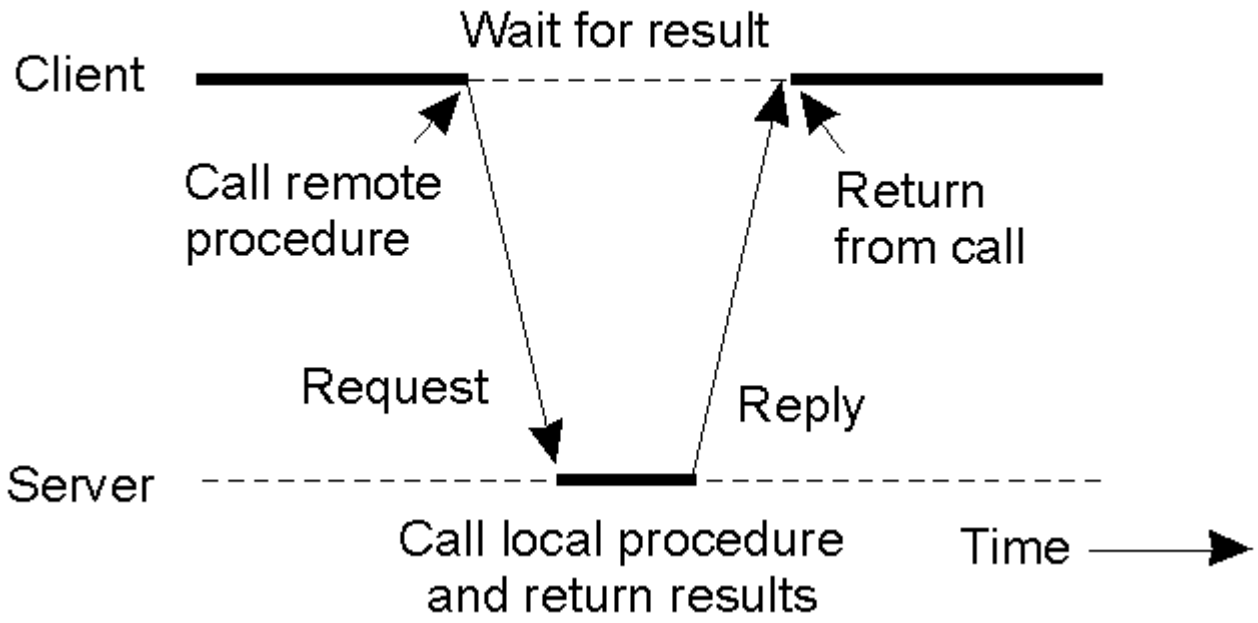
Client program only communicates with the client stub

How RPC Works: Part 2

- The procedure is "split" into two parts:
  - The CLIENT "stub" – implements the interface on the local machine through which the remote functionality can be invoked
  - The SERVER "stub" – implements the actual functionality, i.e. does the real work!

- Parameters are "marshalled" by the client prior to transmission to the server.

Flow Chart

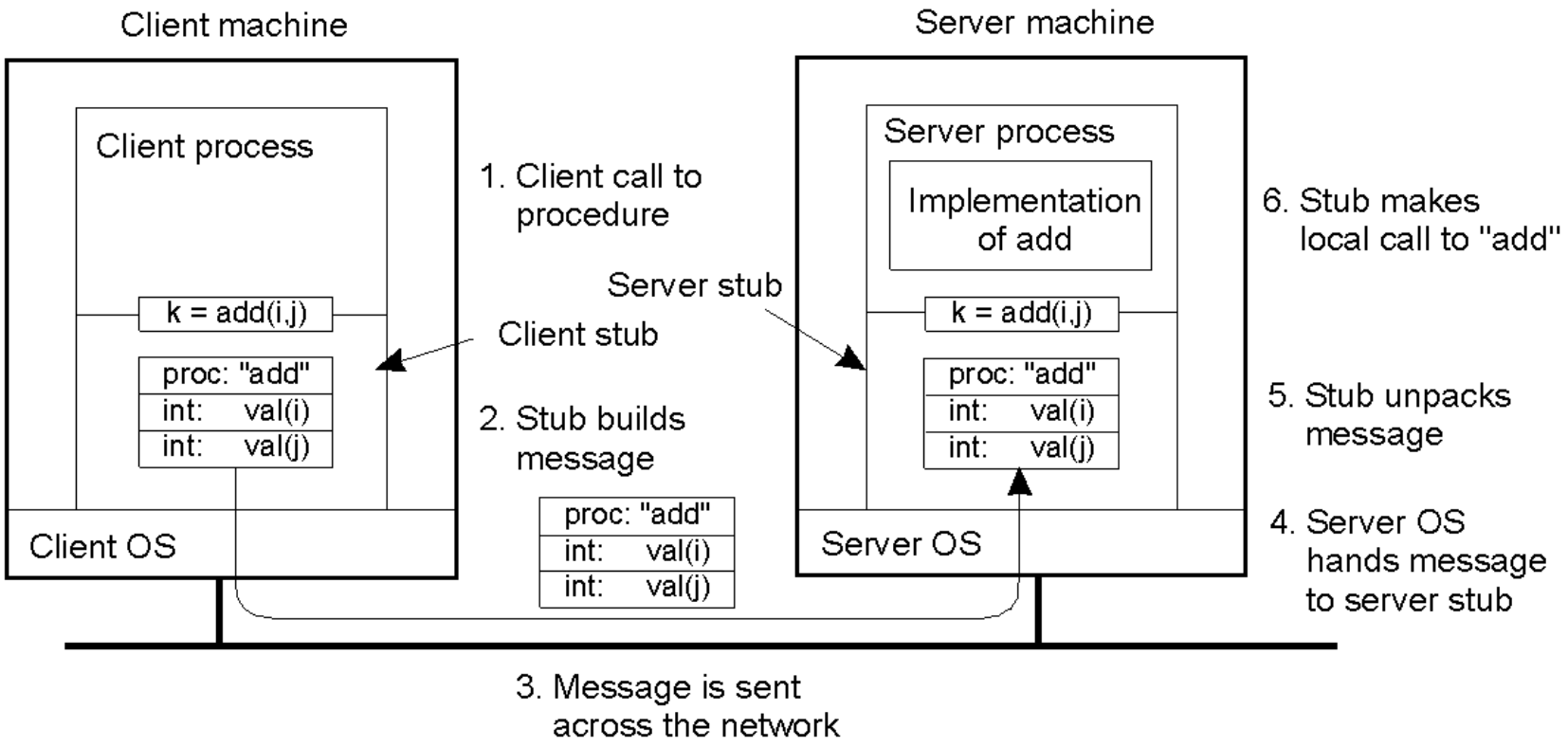


- Principle of RPC between a client and server program
  - no big surprises here …

The Ten Steps of a RPC

1. Client procedure calls client stub in a normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Passing Value Parameters



Steps involved in doing remote computation through RPC

RPC Problems

- RPC works really well if all the machines are homogeneous
- Complications arise when the two machines use different character encodings, e.g. EBCDIC or ASCII
- Byte-ordering is also a problem:
  - Intel machines are “little-endian”
  - Sun Sparc’s are “big-endian”
- Extra mechanisms are required to be built into the RPC mechanism to provide for these types of situations –
  - This adds complexity

Python RPC Server Example

```
import xmlrpclib
from SimpleXMLRPCServer import SimpleXMLRPCServer


def myadd(x, y):
    return (x + y)

server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_function(myadd, "myadd")
server.serve_forever()
```

Save the file as server.py and execute it by

$ python server.py

from the terminal ($ is the command prompt)

Python RPC client example

```
import xmlrpclib

proxy =
xmlrpclib.ServerProxy("http://localhost:8
000/")
print proxy.myadd(2, 3)
```

Save the file as client.py and execute it by

$ python client.py

Interface Definition Language (IDL)

- RPCs typically require development of custom protocol interfaces to be effective
- Protocol interfaces are described by means of an Interface Definition Language (IDL)
- IDLs are "language-neutral" – they do not presuppose the use of any one programming language
- That said, most IDLs look a lot like C …
  - Protocol Buffers (Google)

RPC Summary

- The DSs de-facto standard for communication and application distribution (at the procedure level)

- It is mature
- It is well understood
- It works!