Interprocess Communication

Javier Espinosa, PhD
javier.espinosa@imag.fr

Outline

- **Interprocess communication**
  - Network Protocols
  - Communication Primitives
  - Sockets (UDP and TCP oriented)
  - Exercise: WebSockets

Reminder

Communicating entities (what is communicating)		*Communication paradigms (how they communicate)*		
System-oriented entities	*Problem-oriented entities*	*Interprocess communication*	*Remote invocation*	*Indirect communication*
Nodes	Objects	Message passing	Request-reply	Group communication
Processes	Components	Sockets	RPC	Publish-subscribe
	Web services	Multicast	RMI	Message queues
				Tuple spaces
				DSM

Communication

- Due to the absence of shared memory all communication in distributed systems is based on sending an receiving (low level) messages

- Questions
  - How many volts should be used to signal a O-bit, and how many volts for a I-bit?
  - How does the receiver know which is the last bit of the message?
  - How can it detect if a message has been damaged or lost, and what should it do if it finds out?
  - How long are numbers, strings, and other data items, and how are they represented?

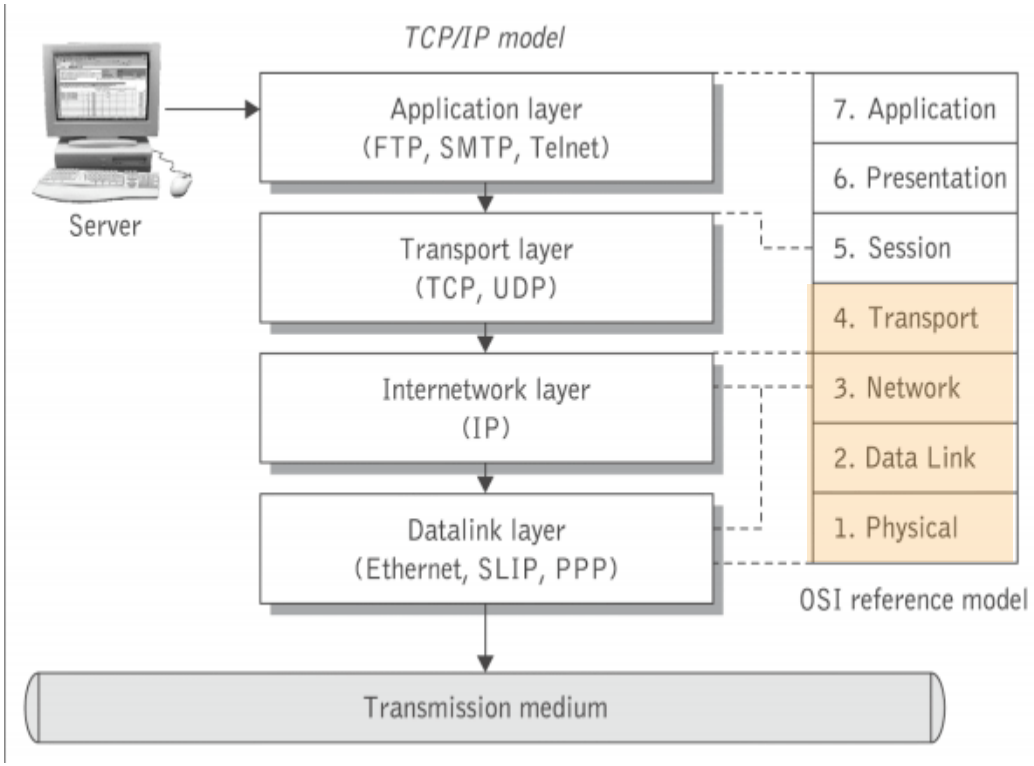
Communication

- Due to the absence of shared memory all communication in distributed systems is based on sending an receiving (low level) messages

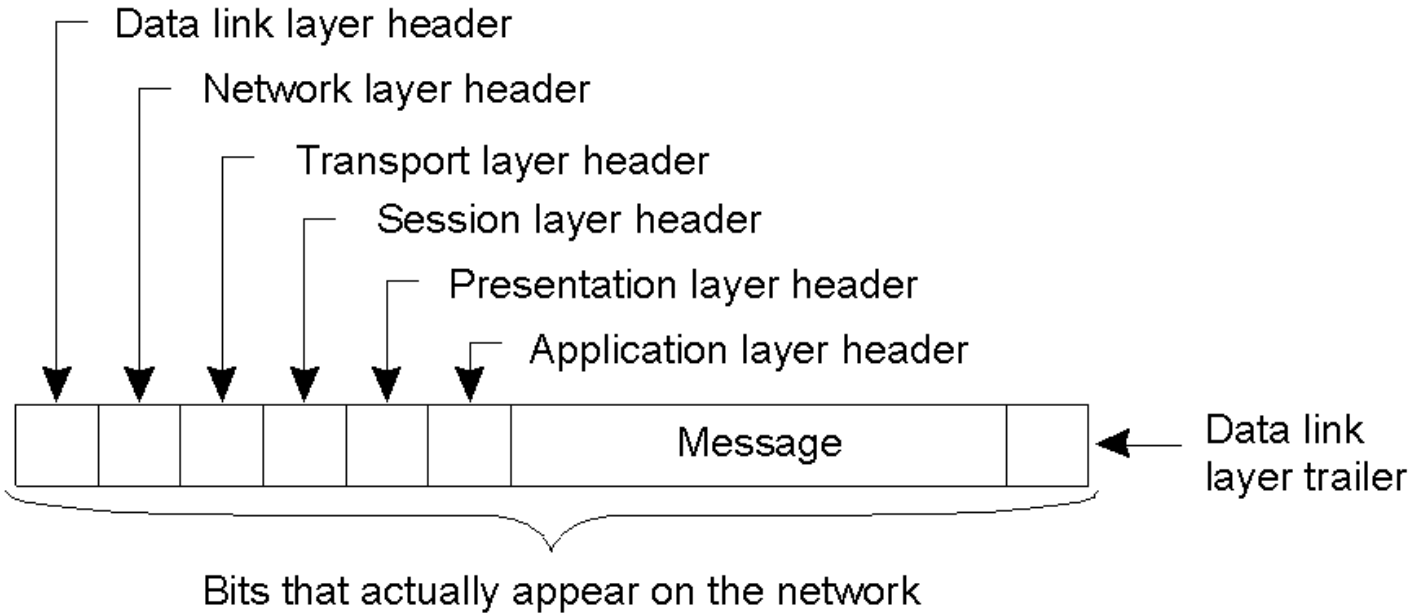
Agreements are needed at a **variety of levels** varying from the low-level details of bit transmission to the high-level details of how information is to be expressed

  - How long are numbers, strings, and other data items, and how are they represented?

Layered Protocols



Layered Protocols



Interprocess Communication

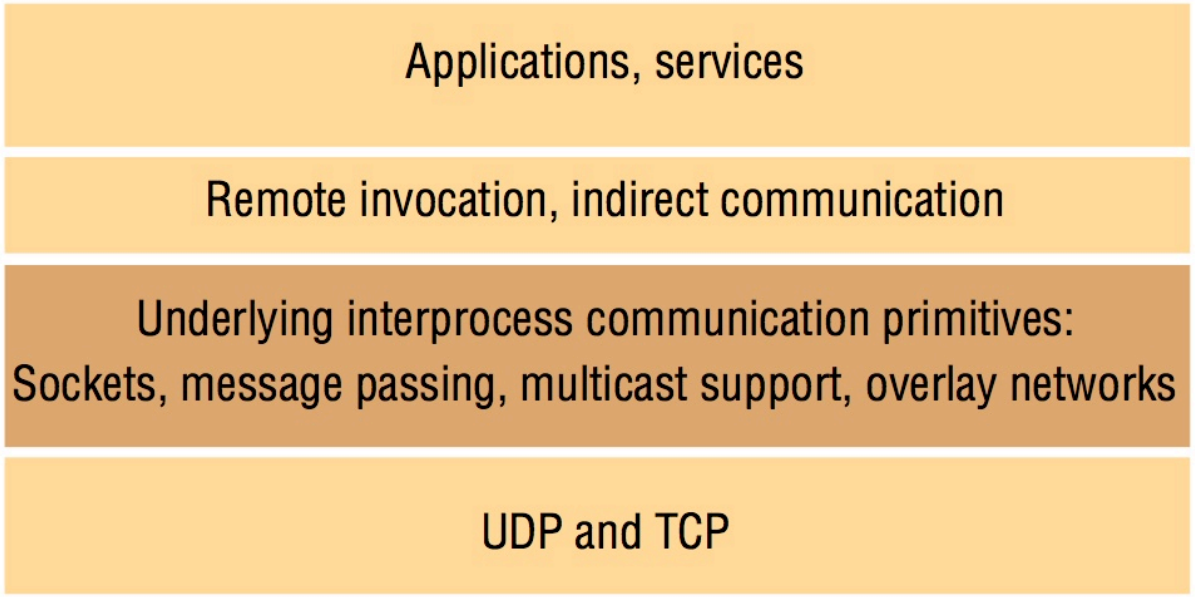
- Provides the basic building blocks for communication over the Internet
- **Datagram oriented**
  - Message passing abstraction
  - Simplest form of interprocess communication

    The sending process sends a message (i.e. datagram) to the receiving process
- **Stream oriented**
  - Abstract a bi-directional communication between a pair of processes
  - Basis for producer-consumer communication (*e.g. HTTP, FTP, SMTP*)

Interprocess Communication

- **Stream oriented (continuation)**
  - Data sent by producers to consumers are queued on arrival at the receiving host until the consumer is ready to receive them.
    - *Consumer must wait when no data items are available*
    - *The producer must wait if the storage used to hold the queued data items is exhausted*

Interprocess Communication Primitives



Interprocess Communication Primitives

- Message passing between 2 processes is supported by 2 operations
  - **Send**

    Sends a message (a sequence of bytes) to a destination process
  - **Receive**

    Waits for a message in the destination process

- A queue message is associated with each message destination
  - Send causes a message to be added to remote queues
  - Receive causes a message to be remove from local queues

(!!) Interprocess communication involves processes synchronization

(A)Synchronous Communication

- Sending and receiving processes may be synchronous and asynchronous
- **Synchronous communication**
  - Ending and receiving processes synchronize at every message
  - *Send* and *receive* are *blocking* operations
    - *When a **send** is issued the sending process (or thread) is blocked until the corresponding **receive** is issued*
    - *When a **receive** is issued by a process (or thread), it blocks until a message arrives*

(A)Synchronous Communication

- **Asynchronous communication**
  - The *send* operation is *non-blocking*

    Sending process is allowed to proceed as soon as the message has been copied to a local buffer, and the transmission of the message proceeds in parallel with the sending process
  - The *receive* operation can have blocking and non-blocking variants
    - **Non-blocking variant**

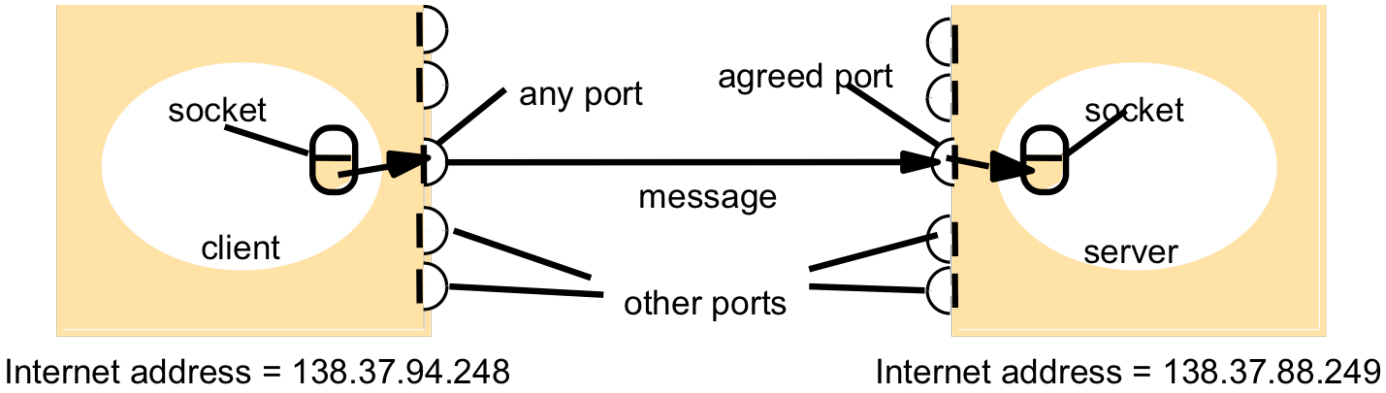
      Receiving process proceeds after issuing a receive operation (a buffer is used in the background), but it must separately receive notification that its buffer has been filled, by polling or interrupt.
    - In multi-threading environments *blocking receive* has no disadvantages.
      - Why?

Message Destination

- Process messages are sent to ***<internet address, local port>***
- **Local port**
  - Integer representing a message destination on a computer
  - Has exactly one receiver but multiple senders
- Processes may use multiple ports to receive messages
  - Any process that knows the number of a port can send a message to it
  - Servers generally publicize their port numbers for use by clients

Sockets

- Endpoint of an interprocess communication
  - Communication between processes consists of transmitting a message between a socket in one process and a socket in another process



Sockets

- Remarks
  - Associated to a particular protocol (*i.e. UDP or TCP*)
  - Processes may use the same socket for sending and receiving messages
  - Each computer has 2^{16} of possible port numbers for use by a local process

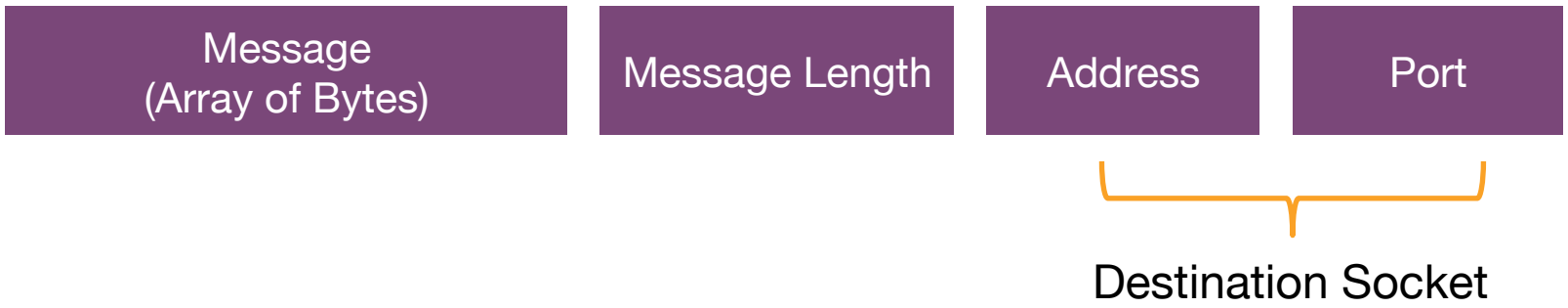
UDP Datagram Communication

- Unreliable by definition
  - No guarantee of delivery, ordering, or duplicate protection
- Messages are size-bounded
  - If the message is too big for the array, the message is truncated on arrival
  - Applications requiring larger messages must fragment the message into chunks
- Sockets normally provide non-blocking sends and blocking receives
  - The send operation returns when it has handed the message to the underlying UDP and IP protocols
- Receive messages from any origin

Example: Java Datagram Communication

Classes

- **Datagram Socket**
  - Support sockets for sending and receiving UDP datagrams
- **Datagram Packet**
  - Represents a UDP datagram as an array of bytes



Example: Java Datagram Communication

UDP Server

Repeatedly receives a request and sends it back to client

```
DatagramSocket socket = null;
int port = 6789;

try {

    socket = new DatagramSocket(port);
    System.out.println("Server listening on port " + port);

    while(true) {

        // 1. Receive UDP request
        DatagramPacket request = new DatagramPacket( new byte[1000], new byte[1000].length );
        socket.receive(request);

        // 2. Print UDP message content
        System.out.print("[localhost:" + request.getPort() + "] ");
        System.out.println(new String(request.getData()));

        // 2. Reply client with same message
        DatagramPacket reply = new DatagramPacket(
                request.getData(),
                request.getLength(),
                request.getAddress(),
                request.getPort()
        );

        socket.send(reply);

    } // while

} catch(Exception ex) {
    System.out.println(ex.getMessage());
```

Example: Java Datagram Communication

UDP Client

Sends message and wait for response

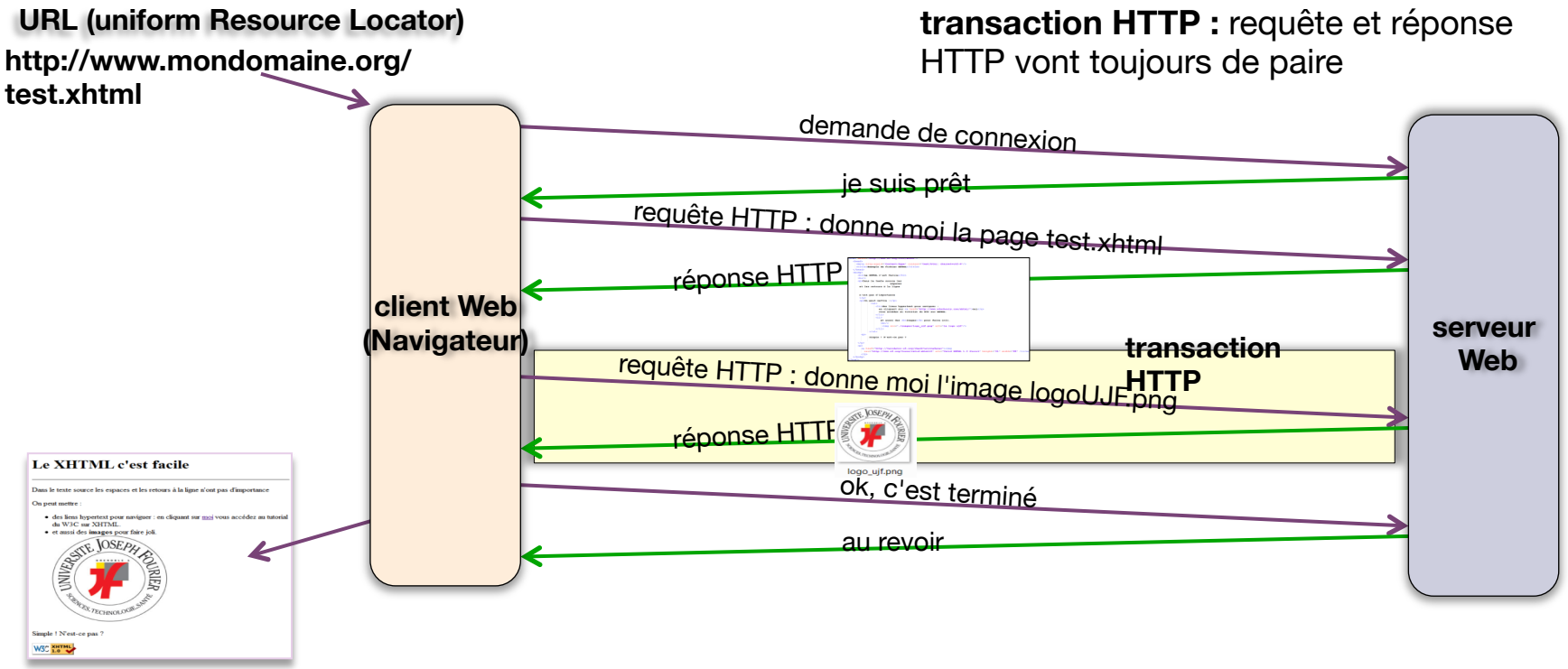
```
DatagramSocket socket = null;

try {
    String message = "Hello World!";
    int serverPort = 6789;

    // 1. Create socket
    socket = new DatagramSocket();

    // 2. Create UDP Request
    DatagramPacket request = new DatagramPacket(
            message.getBytes(),
            message.getBytes().length,
            InetAddress.getLocalHost(),
            serverPort
    );

    // 3. Send request to server
    socket.send( request );

    // 4. Wait for server reply (UDP)
    DatagramPacket reply = new DatagramPacket( new byte[1000], new byte[1000].length );
    socket.receive(reply);

    System.out.println("Reply: " + new String(reply.getData()) );

} catch(Exception ex) {
    System.out.println(ex.getMessage());
}
```

DEMO

TCP Stream Communication

- Abstraction of a stream of bytes to which data may be written/read
- **No message size limitation**
  - A TCP stream decides how much data to collect before transmitting it as IP packets
  - Applications can force data to be sent immediately (if necessary)
- **Connection oriented**
  - Communicating processes establish a connection before they can communicate
    Client process ask for connection and server process accept request
- A socket has 2 streams:
  - 1 input stream
  - 1 output stream

Connection Example



TCP Stream Communication

- Connection Principle
  1. Server process creates a listening socket bound to a server port and waits for request connections
  2. Client creates a stream socket bound to any port and then makes a connect request
  3. When the server accepts a connection, a new stream socket is created for the server to communicate with a client
  4. Server continue listening for new request connections
- Disconnection
  - When an application closes a socket, this indicates that it will not write any more data
  - Process failure or completion automatically closes a socket

TCP Stream Communication

- Remark
  - Communicating processes must agree as to the contents of the data transmitted over a stream

    e.g. if one process writes an int followed by a double to a stream, the reader must read an int followed by a double.
  - When processes do not cooperate correctly in their use of streams, the reading process may experience errors when interpreting the data or may block due to insufficient data in the stream

Example: Java Stream Communication

TCP Server

```
public class TCP_Server {

    public static void main (String args[]) {

        ServerSocket listenSocket = null;
        int port = 7896;

        try{

            // 1. Create socket
            listenSocket = new ServerSocket(port);
            System.out.println("Listening in port " + port);

            // 2. Listen for connections
            while(true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket);
            }

        } catch(IOException ex) { System.out.println("Listen :" + ex.getMessage()); }

    } // method

} // class
```

Example: Java Stream Communication

TCP Client-Server Connection

```
class Connection extends Thread {

    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;

    public Connection (Socket clientSocket) {

        try {

            // 3. Establish connection with client
            this.clientSocket = clientSocket;
            this.in  = new DataInputStream( this.clientSocket.getInputStream());
            this.out = new DataOutputStream( this.clientSocket.getOutputStream());
            this.start();

        } catch(IOException e) { System.out.println(e.getMessage()); }

    } // method

    public void run(){

        try {
            // 4. Receive incoming message
            String data = this.in.readUTF();

            System.out.print("[localhost:" + clientSocket.getPort() + "] ");
            System.out.println(data);

            // 5. Echo incoming message
            this.out.writeUTF(data);

        } catch(Exception e) { System.out.println(e.getMessage()); }

    } // method
```

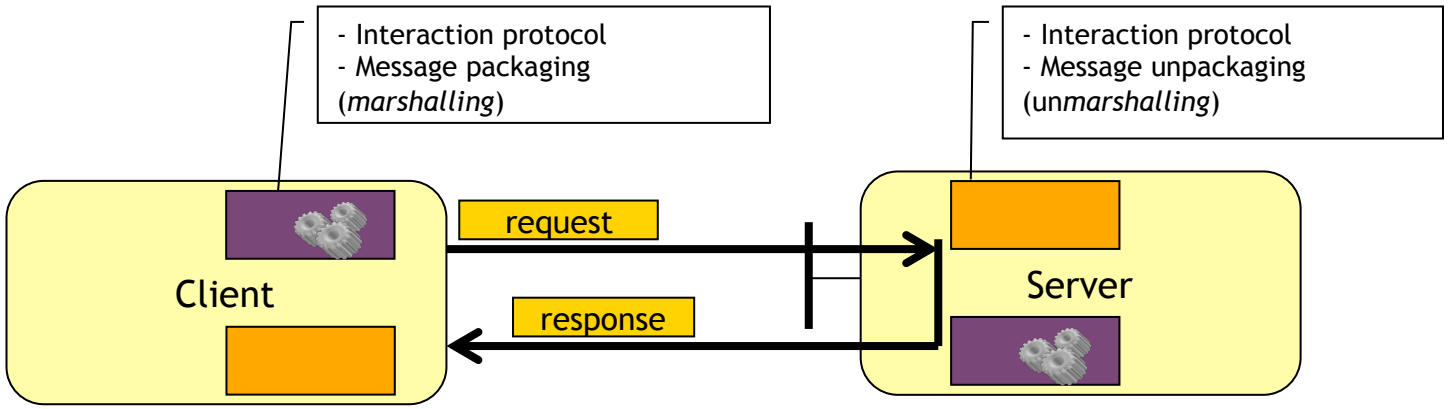
DEMO

Data Representation

- Information stored in running programs is represented as data structures
- Information in network messages consists of sequence of bytes
- Implication
  - Data structures must be flattened (converted to a sequence of bytes) before transmission and rebuilt on arrival
- Why?
  - Not all computers store primitive values as integer and in the same order
  - The representation of floating-point numbers differs between architectures
  - Character representation can also be different (e.g., ASCII, Unicode, etc)

External Data Representation

- Solution for enabling communication among any two computers
  - Convert values to an agree external format before transmission (***marshalling***) and converted to the local from on receipt (***unmarshalling***)
  - Values are transmitted in the sender's format together with an indication of the format used. The recipient converts the values if necessary.



External Data Representation

- Examples
  - Java's object serialization
    - Flattens an object of a a set of interconnected objects into an external representation that may be transmitted in a message or stored on disk
  - XML
    - Textual format for representing structured data
  - JSON
    - External data representation based on a subset of JavaScript

?