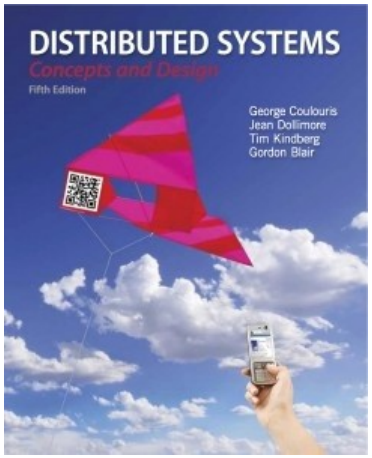
Week 2
Interprocess Communication



Reference:

Chapter 4

Distributed Systems: Concepts and Design

Coulouris, Dollimore, Kindberg and Blair

Edition 5, © Addison Wesley 2011

Learning Objectives

- Recognise characteristics of interprocess communication
- Interpret UDP datagram communication
- Interpret TCP stream communication
- Describe external data representation and marshalling
- Explain request-reply communication protocol.
- Identify features of HTTP as a request-reply protocol

Characteristics of Interprocess Communication

- The interprocess communication involves the activities to communicate data from the sending process to the receiving process and may involve the synchronisation of the two processes.
- The fundamental elements of any communication are:
  - ***Send*** operation is used by a process to send a message to a destination.
  - ***Receive*** operation is used by another process at the destination to receive the message.

Characteristics of Interprocess Communication

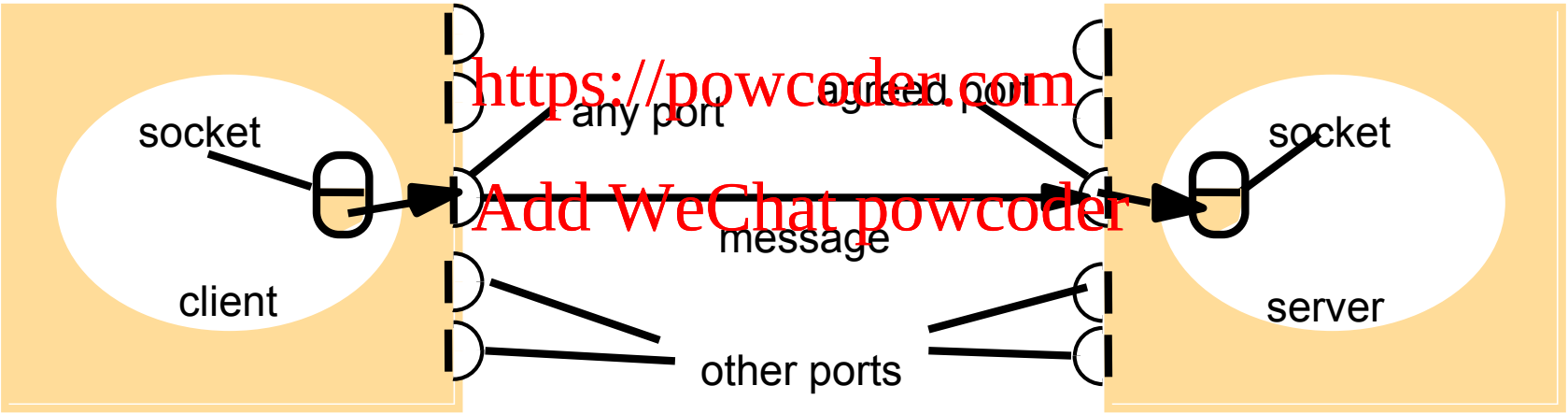
- Communication types include:
  - Synchronous
    - The sender and receiver are synchronised.
    - Both ***send*** and ***receive*** are blocking operations.
  - Asynchronous
    - The ***send*** operation is non-blocking.
    - The sender is allowed to proceed as soon as the message has been copied to a local buffer.
    - Transmission continues in parallel with other processing.
    - In this case the ***receive*** operation may be blocking or non-blocking.

Characteristics of Interprocess Communication

- Message destinations
  - In distributed systems, a process is identified by:
    - ***Internet address***: the location of the node that a process resides.
    - ***Port***: the unique number ($0 \sim 2^{16}$) corresponding to a single sending/receiving process.
- Sockets
  - Sockets are a software abstraction which provides a communication endpoint for processes.
  - A socket encapsulates:
    - An IP address
    - A Port number
    - A protocol, e,g, UDP or TCP

Characteristics of Interprocess Communication

- Client/Server communication via sockets



Internet address = 138.37.94.248

Internet address = 138.37.88.249

UDP Datagram Communication

- UDP communication is unacknowledged and unreliable.
- A datagram is transmitted between processes when one process sends it and the other receives it.
- Sending is non-blocking but receiving is blocking although timeouts can be set.
- Arriving messages are placed in a queue bound to the receiver's port.

UDP Datagram Communication

- The ***receive*** method returns the Internet address and port number of the sender.
- Issues relating to UDP communication include:
  - Message size
    - IP has a limit of 64kbytes on packets and a UDP datagram must fit into this size.
    - Typical applications use limit of 8kbytes.
  - Omission Failures
    - Datagrams can be dropped because of full buffers, checksum failure, either send-omission or receive-omission.
  - Ordering
    - Messages can arrive out of order because the underlying IP routes packets independently.

UDP Datagram Communication

- Overheads relating to reliable message delivery are costly.
  - The need to store state information at sender or receiver.
  - The need to retransmit messages.
  - Latency for the sender.
- It is acceptable by applications where failures are tolerable but overheads are not tolerable.
  - Domain Naming Service
  - Voice over IP

Java APIs for UDP Communication

- Class ***DatagramPacket*** encapsulate:
  - A message and the length of the message
  - The internet address and the port number of the receiver.
- Class ***DatagramSocket***
  - Supports for sending and receiving UDP datagrams.
  - Can be bound to a particular port or allow to choose a free local port.
- Throw ***SocketException*** or ***IOException*** to reflect omission failure.

An Example of Java UDP Client

```
import java.net.*;
import java.io.*;
public class UDPClient{
  public static void main(String args[]){
          //args give message contents and server hostname
    DatagramSocket aSocket = null;
       try{
         aSocket=new DatagramSocket();
         byte[] m=args[0].getBytes();
         InetAddress aHost=InetAddress.getByName(args[1]);
         int serverPort=6789;

         DatagramPacket request=new DatagramPacket(m,
                      args[0].length(), aHost, serverPort);
         aSocket.send(request);
```


☞ This program continues on the next slide

An Example of Java UDP Client

```
        byte[] buffer=new byte[1000];
        DatagramPacket reply=new DatagramPacket(buffer,
                                    buffer.length);
        aSocket.receive(reply);
        System.out.println("Reply: " + new
                                String(reply.getData()));
        }catch (SocketException e)
          {System.out.println("Socket: " + e.getMessage());
        }catch (IOException e)
          {System.out.println("IO: " + e.getMessage());}
        }finally {if(aSocket != null) aSocket.close();}
    }
}
```


The end of this program

An Example of Java UDP Server

```
import java.net.*;
import java.io.*;
public class UDPServer{
  public static void main(String args[]){
    DatagramSocket aSocket = null;
      try{
          aSocket = new DatagramSocket(6789);
          byte[] buffer = new byte[1000];
```


☞ This program continues on the next slide

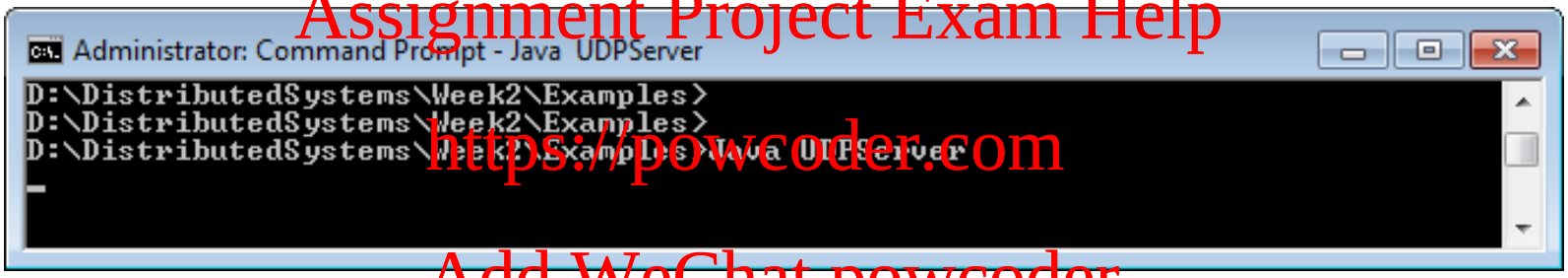
An Example of Java UDP Server

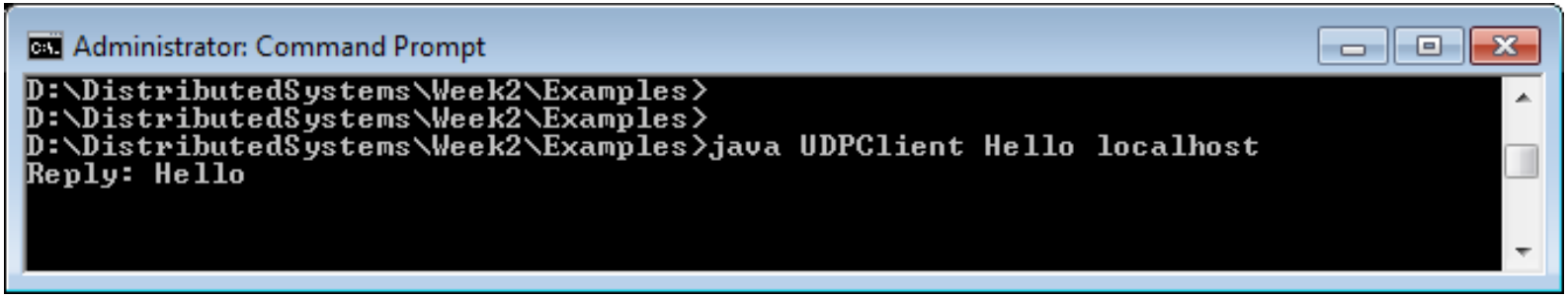
```
        while(true){
          DatagramPacket request=new DatagramPacket
               (buffer, buffer.length);
          aSocket.receive(request);
          DatagramPacket reply=new DatagramPacket
               (request.getData(), request.getLength(),
                request.getAddress(), request.getPort());
          aSocket.send(reply);
        }
      }catch (SocketException e)
        {System.out.println("Socket: " + e.getMessage());
      }catch (IOException e)
        {System.out.println("IO: " + e.getMessage());}
      }finally {if(aSocket != null) aSocket.close();}
    }
  }
```


The end of this program

An Example of Java UDP communication

- Outputs of the ***DUPClient*** and the ***UDPServer***





TCP Stream Communication

- TCP uses a stream of bytes to effect communication.
- To enable reliable communication, TCP uses:
  - Checksums to detect and reject corrupt packets.
  - Acknowledgements to confirm the arrivals of valid messages.
  - Sequence numbers to detect and reject duplicated packets (from retransmission).
  - Timeouts and retransmissions to deal with lost packets.

TCP Stream Communication

- TCP hides the networking issues:
  - Message sizes
  - Lost messages
  - Duplicated and incorrectly ordered messages
  - Flow control issues
  - Message destinations (after initial setup)
- TCP tries best effort to reliably deliver messages even if some packets are lost.
- TCP has no guarantee to message delivery only when connection is broken or processes crash.

Java APIs for TCP Communication

- Distinction is made between a client (using ***Socket***) and a server (using ***ServerSocket***).
- Connection must first be established, and the client sends a connect request.
- When the server accepts the request, a new stream ***Socket*** is created for communication with this client.
- The ***ServerSocket*** keeps listening for new connect requests.

Java APIs for TCP Communication

- The established pair of sockets then support streams in both directions.
- The sender writes onto its output stream via its socket and the receiver reads from its input stream via its socket.
- When a process closes a socket, any remaining data is transmitted together with an indicator that the connection is now broken.
- Throw ***UnknownHostException***, ***EOFException*** and ***IOException*** to reflect omission failure.

An Example of Java TCP Client

```
import java.net.*;
import java.io.*;
public class TCPClient {
  public static void main (String args[]) {
     //arguments supply message and hostname of destination
    Socket s=null;
      try{
        int serverPort=7896;
        s=new Socket(args[1], serverPort);
        DataInputStream in=new
                    DataInputStream(s.getInputStream());
        DataOutputStream out=new
                    DataOutputStream(s.getOutputStream());
        out.writeUTF(args[0]);
```


☞ This program continues on the next slide

An Example of Java TCP Client

```
        String data = in.readUTF();
        System.out.println("Received: "+ data) ;
        }catch (UnknownHostException e){
           System.out.println("Sock:"+e.getMessage());
        }catch (EOFException e){
           System.out.println("EOF:"+e.getMessage());
        }catch (IOException e){
           System.out.println("IO:"+e.getMessage());}
        }finally {
           if(s!=null)
             try {s.close();}
             catch (IOException e){
               System.out.println("close:"+e.getMessage());}}
        }
}
```


The end of this program

An Example of Java TCP Server

```
import java.net.*;
import java.io.*;
public class TCPServer {
  public static void main (String args[]) {
    try{
      int serverPort=7896;
      ServerSocket listenSocket=new
                    ServerSocket(serverPort);
      while(true) {
        Socket clientSocket=listenSocket.accept();
        Connection c = new Connection(clientSocket);
      }
    } catch(IOException e) {
       System.out.println("Listen :"+e.getMessage());}
    }
}
```


☞ This program continues on the next slide

An Example of Java TCP Server

```
class Connection extends Thread {
  DataInputStream in;
  DataOutputStream out;
  Socket clientSocket;
  public Connection (Socket aClientSocket) {
    try {
      clientSocket = aClientSocket;
      in=new DataInputStream(
            clientSocket.getInputStream());
      out=new DataOutputStream(
             clientSocket.getOutputStream());
      this.start();
    } catch(IOException e){
       System.out.println("Connection:"
                              +e.getMessage());}
}
```


☞ This program continues on the next slide

An Example of Java TCP Server

```
        public void run(){
          try { // an echo server
            String data = in.readUTF();
            out.writeUTF(data);
          }
          catch(EOFException e) {
               System.out.println("EOF:"+e.getMessage());}
          catch(IOException e){
               System.out.println("IO:"+e.getMessage());}
          }
          finally {
            try {clientSocket.close();}
            catch(IOException e){/*close failed*/}}
        }
}
```


The end of this program

An Example of Java TCP Communication

- Outputs of the ***TCPClient*** and the ***TCPServer***

Administrator: Command Prompt - Java TCPServer

```
D:\DistributedSystems\Week2\Examples>
D:\DistributedSystems\Week2\Examples>
D:\DistributedSystems\Week2\Examples>
D:\DistributedSystems\Week2\Examples>Java TCPServer
```


Administrator: Command Prompt

```
D:\DistributedSystems\Week2\Examples>
D:\DistributedSystems\Week2\Examples>
D:\DistributedSystems\Week2\Examples>
D:\DistributedSystems\Week2\Examples>
D:\DistributedSystems\Week2\Examples>java TCPClient Hello localhost
Received: Hello

D:\DistributedSystems\Week2\Examples>
```

TCP Applications

- Where overheads can be tolerated, TCP is used to ensure high level of reliability.
  - HTTP is used for communication between web browsers and web servers.
  - FTP allows directories on a remote computer to be browsed and files to be transferred from one computer to another.
  - Telnet provides access by means of a terminal session to a remote computer.
  - SMTP is used to send mails between computers.

External Data Representation

- Information stored in processes is represented in memory as interconnected data structures e.g. arrays, objects etc.
- Different platforms/environments use different representations for primitive data types such as integers (big/little endian), floating point numbers, characters.
- Information in messages is represented as a flat sequence of bytes.
- Agreement is needed on the format for the data on the wire.

External Data Representation

- The ***Person*** structure defined in Java.

```
public class Person implements java.io.Serializable {
        private String name;
        private String address;
        private int year;
        public Person(String aName, String aPlace, int
aYear)
                {
                        this.name = aName;
                        this.place = aPlace;
                        this.year = aYear;
                }
        ............
        ............  //methods for accessing the instance variables
}
```

External Data Marshalling

- Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.
- Unmarshalling is the complementary process of re-assembling the data structure at the destination.
- Three alternative approaches to external data representation and marshalling are:
  - CORBA's CDR (Common Data Representation)
  - Java's Object Serialization
  - XML (eXtensible Markup Language)

External Data Marshalling

- The ***Person*** structure expressed in XML


```
<person pers:id="123456789"
    xmlns:pers = "http://www.cdk4.net/person">
    <pers:name> Smith </pers:name>
    <pers:place> London </pers:place >
    <pers:year> 1934 </pers:year>
</person>
```


☞ This definition continues on the next slide

External Data Marshalling

- The ***Person*** structure expressed in XML

```
<xsd:schema  xmlns:xsd = URL of XML schema definitions    >
   <xsd:element name= "person" type ="personType" />
      <xsd:complexType name="personType">
         <xsd:sequence>
            <xsd:element name = "name"  type="xs:string"/>
            <xsd:element name = "place"  type="xs:string"/>
            <xsd:element name = "year"
                                    type="xs:positiveInteger"/>
         </xsd:sequence>
         <xsd:attribute name="id"
                                    type="xs:positiveInteger"/>
      </xsd:complexType>
</xsd:schema>
```


The end of definition

External Data Marshalling

- Marshalling or unmarshalling requires the consideration of all the finest details and is error-prone if carried out manually.
- Software for marshalling and unmarshalling is available for all commonly used platforms and programming environment.
- Java ***Serialization*** and ***Deserialization*** are examples of automated Marshalling or unmarshalling techniques.

Java Object Serialization

- The Java interface ***Serializable*** has no methods but is used to mark classes which may be serialized and deserialized: that is converted into a format suitable for transmission or for storing on disk, and back.
- Serialization is achieved by creating an instance of the class ***ObjectOutputStream*** and invoke its ***writeObject*** method passing the object concerned as an argument.
- Deserialization is achieved by creating an instance of the class ***ObjectInputStream*** and using its ***readObject*** method to reconstruct the object.

Java Object Serialization

- Java Object Serialization


```
String filename="person";
Person person1 = new Person("Smith", "London", 1934);
FileOutputStream fos = null;
ObjectOutputStream out = null;
try {
  fos = new FileOutputStream(filename);
  out = new ObjectOutputStream(fos);
  out.writeObject(person1);
  out.close();
  System.out.println("Object Persisted");
  }catch(IOException ex)
     {ex.printStackTrace();}
```

Java Object Serialization

- Java Object Deserialization

```
String filename="person";
Person person1 = null;
FileInputStream fis = null;
ObjectInputStream in = null;
  try {
    fis = new FileInputStream(filename);
    in = new ObjectInputStream(fis);
    person1 = (Person)in.readObject();
    in.close();
  }catch(IOException ex) { ex.printStackTrace();
  }catch(ClassNotFoundException ex){ex.printStackTrace();}
  System.out.println("Person Name: " + person1.getName());
  System.out.println("Person Place: "+ person1.getPlace());
  System.out.println("Person Year: " + person1.getYear());
}
```

Java Object Serialization

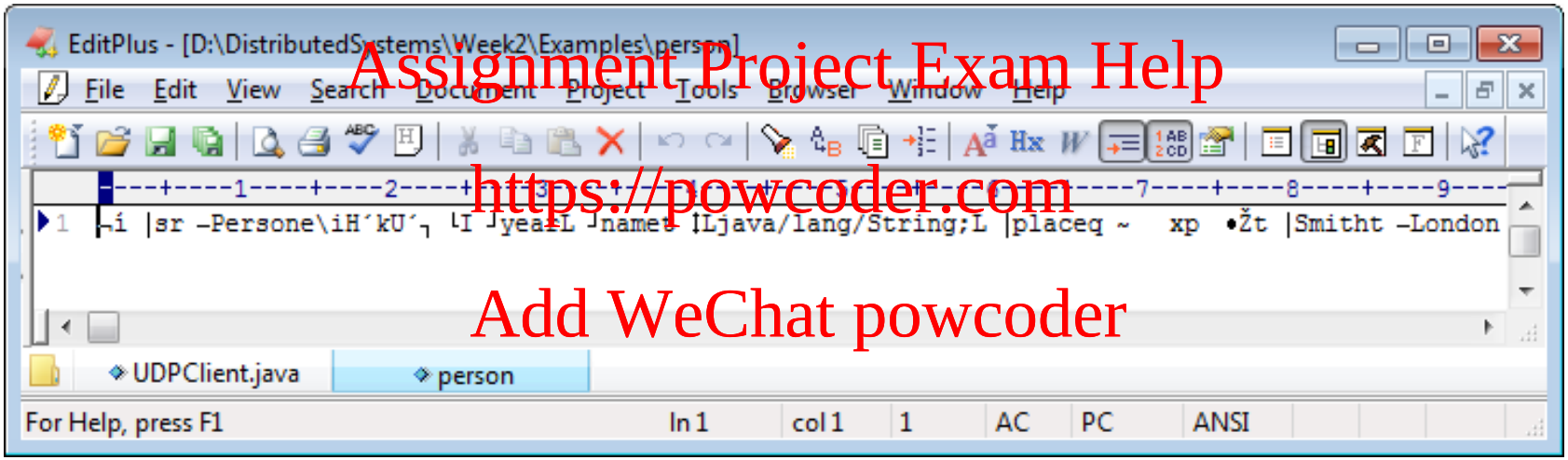
- Screenshots of the example

Select Administrator: Command Prompt

```
D:\DistributedSystems\Week2\Examples>
D:\DistributedSystems\Week2\Examples>
D:\DistributedSystems\Week2\Examples>Java PersonPersist
Object Persisted

D:\DistributedSystems\Week2\Examples>java GetPersonDetails
Person Name: Smith
Person Place: London
Person Year: 1934

D:\DistributedSystems\Week2\Examples>dir
 Volume in drive D is Data
 Volume Serial Number is 24BA-0BF3

 Directory of D:\DistributedSystems\Week2\Examples

02/02/2012  09:13 AM    <DIR>          .
02/02/2012  09:13 AM    <DIR>          ..
02/02/2012  09:08 AM             1,630 Connection.class
02/02/2012  09:08 AM             1,377 GetPersonDetails.class
30/11/2007  11:19 AM               881 GetPersonDetails.java
02/02/2012  09:08 AM             1,699 MulticastPeer.class
07/12/2007  04:42 PM               988 MulticastPeer.java
02/02/2012  09:13 AM                96 person
02/02/2012  09:08 AM               784 Person.class
30/11/2007  11:04 AM               650 Person.java
02/02/2012  09:08 AM               989 PersonPersist.class
30/11/2007  11:19 AM               638 PersonPersist.java
02/02/2012  09:08 AM             1,804 TCPClient.class
07/12/2007  04:42 PM               910 TCPClient.java
02/02/2012  09:08 AM               908 TCPServer.class
07/12/2007  04:42 PM             1,239 TCPServer.java
```

Java Object Serialization

- Screenshots of the example



HTTP as a Request-Reply Protocol

- The request-reply protocol has the following three forms:
  - R (request) is used when:
    - A client sends a single request message to a remote server.
    - There is no value to be returned from the remote server.
    - The client requires no confirmation.
  - RR (request-reply)
    - A client sends a request message to a remote server.
    - A response (also as the acknowledgement) from the server is sent to the client.
  - RRA (request-reply-acknowledge)
    - The client makes a request.
    - The server responds.
    - The client acknowledges the response.

HTTP as a Request-Reply Protocol

- HTTP refers to Hypertext Transfer Protocol.
- A browser is a HTTP client because it sends requests to a HTTP server (Web server).
- The HTTP server sends responses back to the client.
- Each message sent is acknowledged by HTTP.
- HTTP severs manage resources (identified by URLs) in different ways.
  - Data such as web pages, image files
  - Programs such as Java servlets

HTTP as a Request-Reply Protocol

- HTTP supports for:
  - Content negotiation – what data representation can be accepted.
  - Authentication – user name and password style logon.
- HTTP supports for methods:
  - ***GET*** sends request to a server and requires reply.
  - ***HEAD*** is identical to ***GET*** except only information about data is replied.
  - ***POST*** sends data to a server, requesting to perform a special function.

HTTP as a Request-Reply Protocol

- HTTP supports for methods:
  - ***PUT*** requests the supplied data to stored with the given URL.
  - ***DELETE*** requests the sever to delete the resource identified by the given URL.
  - ***OPTIONS*** requests the server to supply with a list of methods with the given URL
  - ***TRACE*** requests the server to send back the request message.

HTTP as a Request-Reply Protocol

- HTTP message format
  - Request message

method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/1.1		

  - Reply message

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

HTTP as a Request-Reply Protocol

- Example of HTTP request

```
GET /index.html HTTP/1.1
Host: www.example.com
[blank line]
[data]
```


- Example of HTTP response


```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.27 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8

<html>
  <body>
    <h1>Hello World!</h1>
          (more file contents)
  </body>
</html>
```

HTTP as a Request-Reply Protocol

- In HTTP 1.0 and before, TCP connections are closed after each request and response, so each resource to be retrieved requires its own connection.
- Opening and closing TCP connections takes a substantial amount of CPU time, bandwidth, and memory.
- In practice, most Web pages consist of several files on the same server, so much can be saved by allowing several requests and responses to be sent through a single persistent connection.

HTTP as a Request-Reply Protocol

- Persistent connections are the default in HTTP 1.1.
- Just open a connection and send several requests in series (called pipelining), and read the responses in the same order as the requests were sent.
- If a client includes the "Connection: close" header in the request, then the connection will be closed after the corresponding response.
- If a response contains this header, then the server will close the connection following that response, and the client shouldn't send any more requests through that connection.

Summary

- The fundamental communication elements are ***send*** and ***receive*** operations, which can be synchronous or asynchronous.
- In programming languages, sockets are used to encapsulate internet addresses, ports and methods for interprocess communication.
- UPD datagram communication is efficient but unreliable; TCP stream communication is reliable but has great overheads.
- The request-reply protocols are used for client/server communication.
- HTTP is a request-reply protocol built on TCP for reliable client/server communication over the Internet.