ENTERPRISE JAVA PROGRAMMING / PROGRAMMING II

MODULES CT545 / CT875, SEMESTER 2, ACADEMIC YEAR 2020-2021
NATIONAL UNIVERSITY OF IRELAND, GALWAY

Lecture 4a

Lecturer: Dr Patrick Mannion

TOPICS

- Networking and client/server programming with Java

NETWORKING AND CLIENT/SERVER

- Now for a different (and perhaps easier) topic...
- Networking and Client/Server programming with Java...
  - We briefly revise basics of networking in general
  - We look in detail at how clients and a server communicate with each other
  - Along the way: a practical application of *streams* (beyond `System.out` and file handling)
  - Also an important use case for multithreading!

NETWORKING AND CLIENT/SERVER

- Developing applications which are capable of interacting with other software over the Internet is a crucial task in modern software development.
- This lecture covers the basic of networking and client/server software.
- In contrast to the Enterprise lectures in the second half of the semester, we now create *socket-based* servers and clients almost from scratch, instead of using some Enterprise server such as JBoss

NETWORKING AND CLIENT/SERVER

- Basic concept:
  Two network *endpoints* (either software or hardware devices) communicate with each other by transferring data (e.g., web pages, or emails) over the network.
- The computer corresponding to an endpoint is called *host*

NETWORKING AND CLIENT/SERVER

- Most networked software applications are *client/server* applications:
    - One of the endpoints belongs to a *server* = a program or machine which offers a certain service
    - The other belongs to a *client*: a program or machine which uses that service
    - Typically, one server interacts with multiple clients
    - Typically, clients send *request* messages to the server, the server responds to these requests by sending messages to its clients.

      Example: web browsers (clients of web servers) request web pages from a web server. The web server responds by sending the requested web pages to the respective web browser (or it sends an error message in case the requested page isn't available)

NETWORKING AND CLIENT/SERVER

- A *network protocol* consists of *the* rules which specify how the network endpoints interact with each other correctly
- Network protocols specify the syntax, order and meaning of the messages which are exchanged between the server and its clients
- Also: what to do in case of an error in order to avoid loss of data, and how to start and end a networking session.

NETWORKING AND CLIENT/SERVER

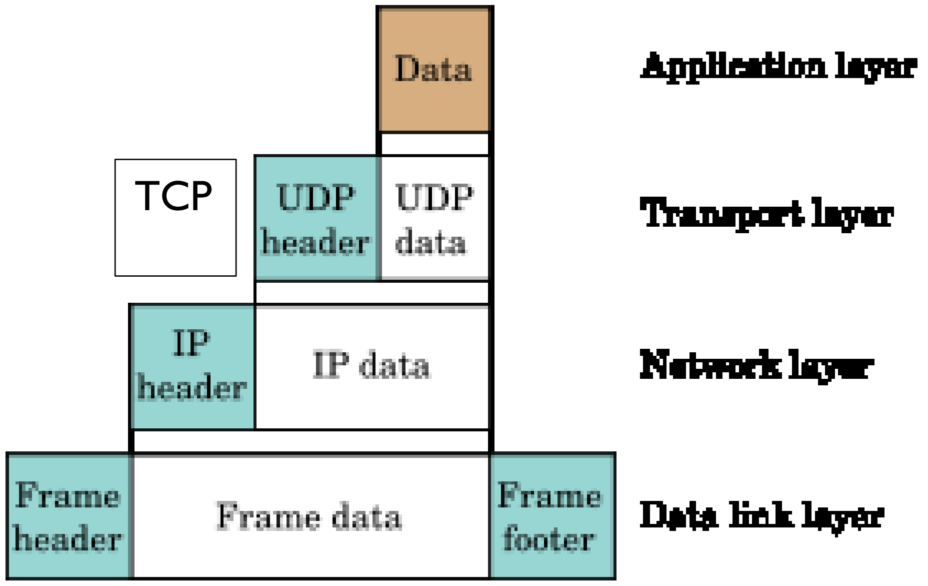
- Typically, a network connection is governed by *multiple*, aligned protocols at the same time.
- Each protocol is responsible for certain functionalities only and might rely on lower-level protocols for more basic tasks.
- The Internet is based on a standardized *hierarchy* of network protocols.
- Some of these protocols are used by all Internet applications, such as TCP/IP, whereas others are application-specific

NETWORKING AND CLIENT/SERVER

- Each network protocols resides on a certain (notional) *network layer*. Each layer stands for a certain level of network functionality and abstraction.
- The *Internet protocol suite (IP suite)* is the set of networking protocols which the Internet is based on. It is organised in several layers which form a hierarchy. Such a set of protocols is called a *protocol stack*.

NETWORKING AND CLIENT/SERVER

- The "IP suite" uses *encapsulation* to provide abstraction of protocols and services (similarly to encapsulation in object oriented programming).
- A protocol at a higher layer relies on protocols on lower layers to help accomplish its aims:

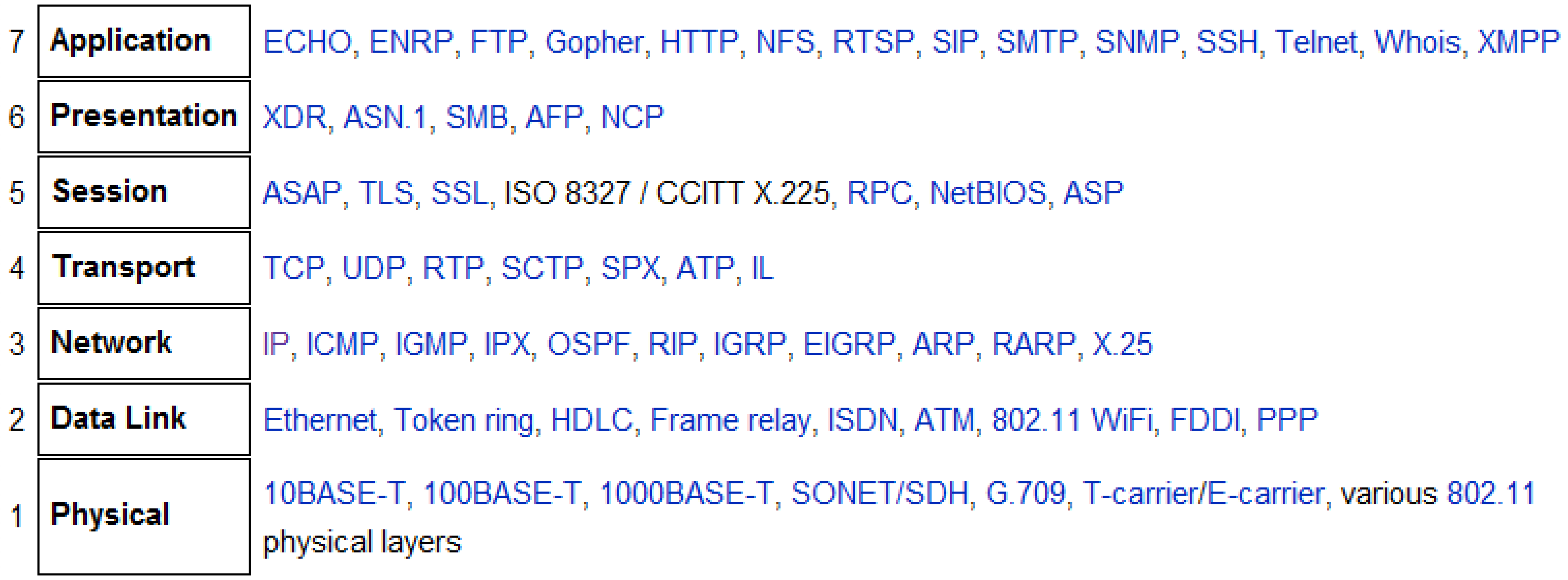


(top of the hierarchy)

(bottom of the hierarchy)

NETWORKING AND CLIENT/SERVER

- The Internet has lots of protocols. The following set is incomplete (and no need to memorise all of these protocols, they are just given as examples):



7	**Application**	ECHO, ENRP, FTP, Gopher, HTTP, NFS, RTSP, SIP, SMTP, SNMP, SSH, Telnet, Whois, XMPP
6	**Presentation**	XDR, ASN.1, SMB, AFP, NCP
5	**Session**	ASAP, TLS, SSL, ISO 8327 / CCITT X.225, RPC, NetBIOS, ASP
4	**Transport**	TCP, UDP, RTP, SCTP, SPX, ATP, IL
3	**Network**	IP, ICMP, IGMP, IPX, OSPF, RIP, IGRP, EIGRP, ARP, RARP, X.25
2	**Data Link**	Ethernet, Token ring, HDLC, Frame relay, ISDN, ATM, 802.11 WiFi, FDDI, PPP
1	**Physical**	10BASE-T, 100BASE-T, 1000BASE-T, SONET/SDH, G.709, T-carrier/E-carrier, various 802.11 physical layers

NETWORKING AND CLIENT/SERVER

- The following protocols provide the foundation of the World Wide Web (WWW):
  - *Application-layer protocols*. For example, the *Post Office Protocol* (POP3), the *Hypertext Transfer Protocol* (HTTP), and proprietary protocols for other kinds of software
  - The *Transmission Control Protocol* (TCP)
  - The *Internet Protocol* (IP)
- The application-layer protocols rely on TCP. TCP relies on IP.

NETWORKING AND CLIENT/SERVER

Internet Protocol (IP)

- The *Internet Protocol* (IP) was developed to enable different *local area networks* to communicate with each other. Therefore, the Internet is sometimes called a "network of networks".
- It has become the basis for connecting computers around the world together over the Internet.
- It is a low-level protocol on which all other Internet protocols rely.
- *IP addresses* identify the hosts (endpoints) of a certain Internet connection. With version 4 of the IP protocol, each IP address is denoted as a sequence of four numbers, each being one byte (example: 130.65.86.66).
- The current version of IP is 6, but IP 4 is still far more often used.

NETWORKING AND CLIENT/SERVER

Internet Protocol (IP)

- Some IP addresses have a special meaning. E.g., 127.0.0.1 identifies the *local host* = your own computer.

- In addition to the IP address, a host can have a human-readable *domain name* (e.g., "nuigalway.ie")
  The readable name for 127.0.0.1 is "localhost"

- Typically, client and server run on different machines, but using localhost, they can reside on the same machine for testing and debugging purposes (we will do this in the lab sessions).

NETWORKING AND CLIENT/SERVER

Transmission Control Protocol (TCP)

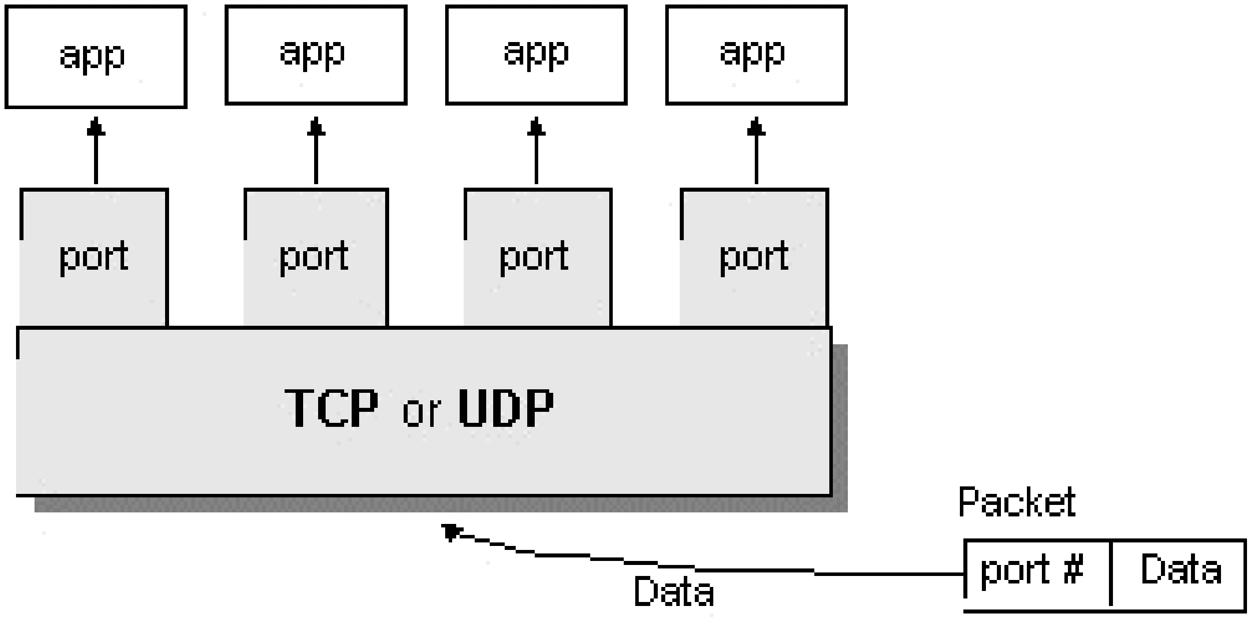
- The IP protocol does not notify the sender if data is lost or garbled.
- This is the job of a higher-level protocol which uses IP, namely the *Transmission Control Protocol (TCP)*.
- TCP is *reliable*:
  - It attempts to deliver the data and tries again if there are failures.
  - The sender is notified whether or not the attempt was eventually successful.
  - It rearranges out-of-order IP packets.

NETWORKING AND CLIENT/SERVER

Transmission Control Protocol (TCP)

- When data is sent to a certain host, it is necessary to indicate which application on this computer should receive the data.
- TCP uses *port numbers* (*or ports* for short) for this.
- A port number is an integer between 0 and 65535.
- The program on the sending host must know the port number of the program on the receiver host.
- Ports might be reserved for certain types of software applications (e.g., port 80 for *web servers*).
- Port numbers below 1024 are mostly reserved for standard purposes.

NETWORKING AND CLIENT/SERVER
Transmission Control Protocol (TCP)



NETWORKING AND CLIENT/SERVER

Application level protocols

- TCP/IP establishes an Internet connection between two ports on two hosts.
- Application-layer protocols are higher-level protocols for application-specific network interaction between programs, such as for sending emails or loading web pages.
- Application-layer protocols make use of underlying lower-level protocols (in particular TCP/IP).

NETWORKING AND CLIENT/SERVER

Client/Server

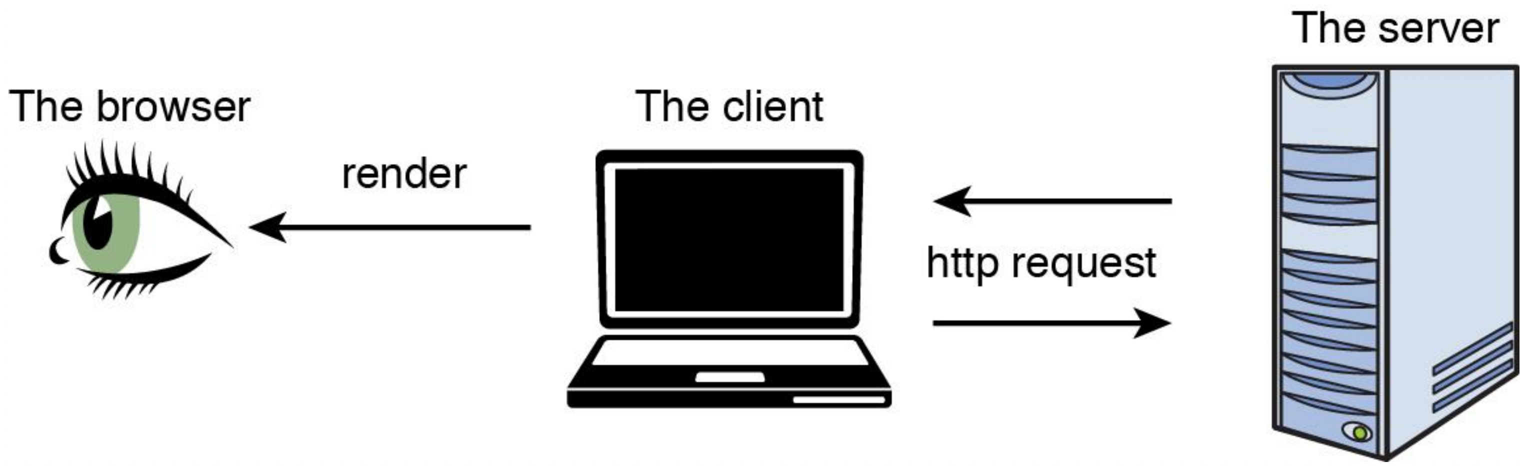
- In case of client/server applications, an appropriate application-layer protocol specifies how the messages exchanged between clients and server look like, what their meaning is, and in which order messages are allowed to be sent and received.
- Example protocols on the application layer: HTTP (if the server is a web server), IMAP/POP3/SMTP (email)
- For other client/server applications, we might need to define our own protocols...

NETWORKING AND CLIENT/SERVER
Hypertext Transfer Protocol (HTTP)

- *The Hypertext Transfer Protocol* (HTTP) is the best-known example for an application-layer protocol in a client/server setting:
  - HTTP is the application-layer protocol used for the World Wide Web (WWW).
  - HTTP servers are usually called *web servers*. Their clients are the *web browsers*.
  - By default, a web server listens on port 80 for client requests.
  - Main type of service: delivery of *web pages* to their clients.
  - Web pages can contain scripts (small programs, written e.g. in JavaScript) which are executed on client side (i.e., in the browser).

NETWORKING AND CLIENT/SERVER

Hypertext Transfer Protocol (HTTP)



NETWORKING AND CLIENT/SERVER
Hypertext Transfer Protocol (HTTP)

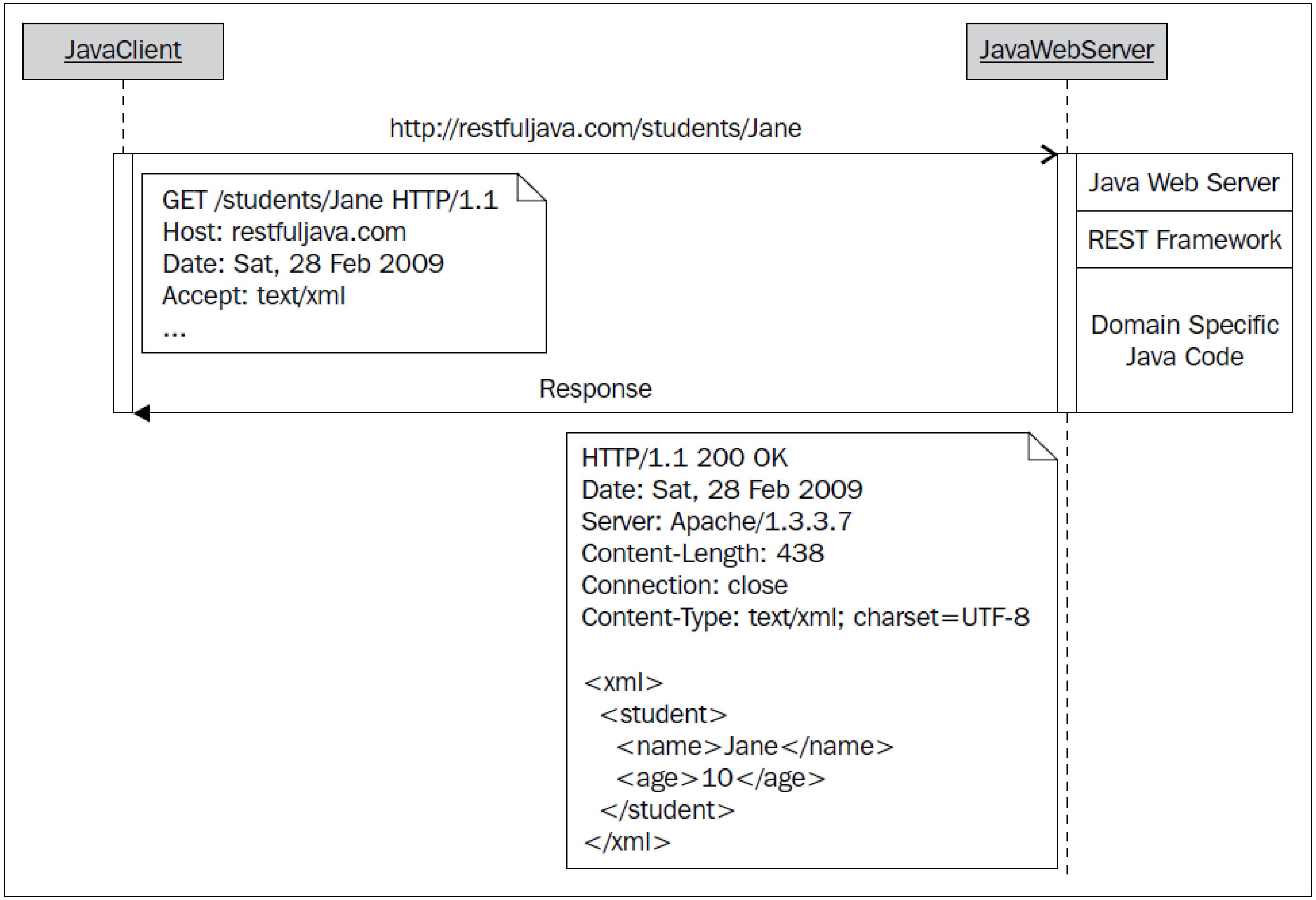
- A web resource is identified using a *Uniform Resource Identifier* (URI)
- URIs identify web resources. Optionally (and typically), a URI can also identify the *location* of this source. Such a URI is called *Uniform Resource Locator (URL*). Essentially, URLs achieve this by containing domain names, and optionally directory and file names. E.g.,

  http://www.java.sun.com/index.html
- Other Internet applications use URIs too. E.g.,

  mailto:admin@randomsite.com is a URI, but not a URL. The Internet resource named here is a mailbox. Such a URI is sometimes called a URN (*Uniform Resource Name*)
- Some technical documents use the term "URL" instead of "URI" even if the respective URI does not describe a location

NETWORKING AND CLIENT/SERVER

HTTP RESTful web service example


JavaClient
JavaWebServer
http://restfuljava.com/students/Jane
Java Web Server
REST Framework
Domain Specific Java Code
Response

```
GET /students/Jane HTTP/1.1
Host: restfuljava.com
Date: Sat, 28 Feb 2009
Accept: text/xml
...
```

```
HTTP/1.1 200 OK
Date: Sat, 28 Feb 2009
Server: Apache/1.3.3.7
Content-Length: 438
Connection: close
Content-Type: text/xml; charset=UTF-8

<xml>
  <student>
    <name>Jane</name>
    <age>10</age>
  </student>
</xml>
```

NETWORKING AND CLIENT/SERVER

- For many standard tasks of the Web, HTTP is used (not just for web servers)
- However, for your own client/server applications, you might need to develop your own application-layer protocol
- Sometimes, application-layer protocols are designed on top of other application-layer protocols (so it is possible to, e.g., create a protocol which makes use of HTTP for application-specific functionality which was originally not meant to be done using HTTP. E.g. RESTful web services)
- Later in this lecture we will see an example for a self-designed application-layer protocol

NETWORKING AND CLIENT/SERVER
Client/Server sockets

- How to use all this in a Java program...?
- Programming languages usually support networking using so-called (*Internet*) *sockets*.
- A socket represents an endpoint of a communication flow over the Internet
- Each socket is bound to a certain host (identified using an IP address and/or a host name) and a certain port number. The port number is application-specific.
  Host address and port number together form the *socket address*.
- Information flow is typically bi-directional, that is, data (messages) sent between two sockets can go from client to server as well as from server to client

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- The messages send between sockets are specified by the (Java) programmer (that is, you...), subject to an appropriate application-layer protocol
- As the lower-level protocols on which the application-layer protocol relies, sockets use TCP/IP (automatically)

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- Java socket objects provide the functionality of *client sockets* (class `java.net.Socket`) and *server sockets* (class `java.net.ServerSocket`)...
- A socket object is used to send messages to the respective other host over the Internet (or the special case of "localhost", where client and server are on the same machine)
- Server sockets are used on server-side only, to accept new client connections. <u>Client sockets are used on client-side (to connect to a server) ***and(!)*** on server-side (to deal with clients *after* they have connected to the server)</u>
- Sockets are not restricted to any concrete application (protocol). It is up to the programmer to send the correct messages in the correct order.

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- We first show how to use a client socket on client-side...
- Package: `java.net`
- Creating a client socket for communication with some server:

```
Socket socket = new Socket(hostname, portnumber);
```

- `hostname, portnumber` are those of the other host (i.e., the server)
- Example code for connecting to HTTP (web) server java.sun.com:

```
final int HTTP_PORT = 80;
Socket socket = new Socket("java.sun.com", HTTP_PORT);
```

- If it can't find the host, the Socket constructor throws an `UnknownHostException`

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- Use input/output *streams* attached to sockets to receive/send data from/to the respective other host via the Internet.
- A stream represents the sequence of data flowing between two sockets: the client-socket on client side and a client-socket on server-side.
- We need to obtain an *input* and an *output* stream for each client socket, in order to receive/send data via this socket

- Streams are not specific to networking, but can be used for other purposes two (e.g., writing/reading data into/from a file, or the well-known `System.out/System.in`) → future lecture.

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- Clients do not only receive data from the server, but they also need to send data to the server, in particular *requests*.
- Furthermore, well-programmed clients and servers acknowledge the correct reception or handling of important messages received from the other side by sending "acknowledgement messages".
- So, because the communication between a client and a server is typically bidirectional, each socket has actually two streams attached: an input and an output stream, which you obtain from that socket as follows:

```
InputStream instream = socket.getInputStream();
OutputStream outstream = socket.getOutputStream();
```

NETWORKING AND CLIENT/SERVER

Client/Server sockets

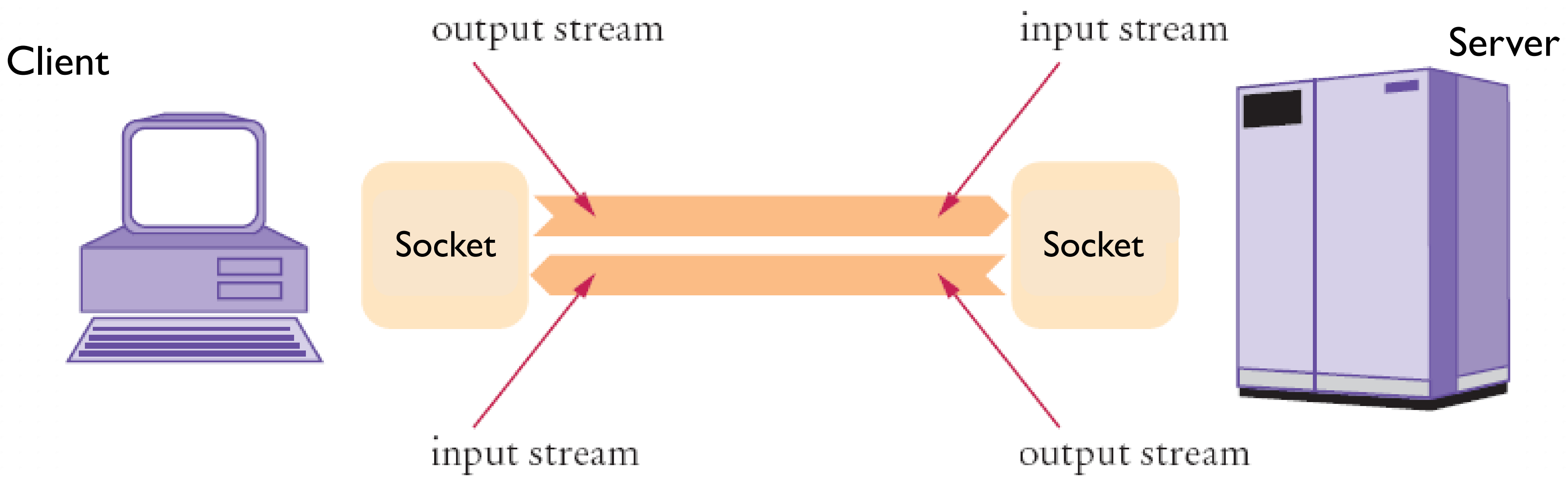
- When you write data into the client’s output stream, the socket forwards it to the server
- Conversely, when data coming from the server is received, it arrives in the client's input stream
- Vice-versa on server-side, where also a client socket (!) exists which also has an input and an output stream.
- **The client-side output stream is "connected" to the server-side input stream (and vice versa).**
- When you are finished with communication with the other host, "close" both streams and the socket:

```
instream.close();
outstream.close();

socket.close();
```

NETWORKING AND CLIENT/SERVER

Client/Server sockets



NETWORKING AND CLIENT/SERVER

Client/Server sockets

- The streams obtained using `socket.getInputStream()` and `socket.getOutputStream()` are low-level streams for <u>bytes</u>...
- To write strings like text messages to the output stream, you can use, e.g., `PrintWriter`:

```
PrintWriter out = new PrintWriter(outstream);
...
out.print("Hi!");
...
```

- To print a string followed by `\n` (i.e., an entire line):

```
out.println("Some paragraph with 4 words");
```

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- To receive tokens (e.g., words or numbers separated by whitespace or other delimiters), one possibility is to use class `Scanner`. It provides powerful methods similar to those seen with Iterators:

```
Scanner in = new Scanner(instream);

String request;
if(in.hasNext()) // check if input available
    request = in.next();  // read next word
```

- `Scanner` can read various data types. E.g., to read a float number, use

```
float f = in.nextFloat();
```

- `Scanner` can also read an entire line (ended by `\n`):

```
String str = in.nextLine();
```

- See Java API documentation for further details and examples

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- What is being read must precisely match what was written into the output stream by the other host, in the same order!
  If, e.g., firstly a word and then a number have been written to the output stream, the input stream must also read the word first and afterwards the number!
- After using a stream, always close it:
  `stream.close();`
- Note: output streams often "buffer" the data put into them before actually passing it on. To write data immediately out to the receiver, use `stream.flush();`
- Stream operations throw various exceptions in case an error happens. Therefore, enclose them in `try/catch` accordingly

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- Example client code: requesting data from a *web server*, that is,
  in this example, we are using sockets with the HTTP protocol as application-layer protocol:

```
Socket s = new Socket("java.sun.com", 80);
InputStream instream = s.getInputStream();
OutputStream outstream = s.getOutputStream();
Scanner in = new Scanner(instream);
PrintWriter out = new PrintWriter(outstream);
String command = "GET / HTTP/1.0\n\n";  // request message
                                  // for the server (incomplete example)
out.print(command);
out.flush(); // immediately sends the message
while (in.hasNextLine()) {  // receive response from server
    String input = in.nextLine();
    System.out.println(input);
}
in.close(); out.close();
s.close();  // always close sockets and streams after use!
```

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- The output of this program might be something like this:

```
Getting / from java.sun.com
HTTP/1.1 200 OK
Server: Apache
Date: Wed, 21 Jul 2014 23:47:27 GMT
Content-type: text/html;charset=ISO-8859-1
Connection: close
```

HTTP meta-information about the completed transfer

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Java Technology</title>
...
</body>
</html>
```

HTML code of the web page received from the server

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- Now we look at the server-side...
- Implementing a server requires a *server socket* (class `java.net.ServerSocket`), in addition to class `Socket`
- The server socket waits for and accepts client connections and assigns each client a (server-side!) client socket.
- **This client socket on server-side communicates with the client socket on client-side**
  (using a stream of data "flowing" between the client socket on client-side and the one of server-side).
- We also need
  - again the application-layer protocol (standardized or self-defined)
  - decide for a unique port number at which our server should be reachable. Use a high, random port number for your own application, provided that any clients work on the same machine (i.e., the server is on host `localhost`)

NETWORKING AND CLIENT/SERVER
Client/Server sockets

- A server socket listens for incoming client connections and is used to obtain a client socket which later "talks" to the connected client:

```
int portNumber = 9999;
ServerSocket server = new ServerSocket(portNumber);
Socket s = server.accept(); // s is a client socket on server-side
... // use client socket s to send/retrieve data to/from the client,
  using streams attached to the client socket
```

- `server.accept()` *blocks* (waits) while it listens for new clients. Clients need to connect via the specified port (here: 9999).
- Server sockets are only for accepting connections. Don't attach streams to them!

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- As you can see on the previous slide, the server does not only need a server socket (class `ServerSocket`), but also a client socket (class `Socket`)
  ```
  Socket s = server.accept() // s is the client socket on server-side
  ```
- The client socket on server-side is used for the actual data transfer to/from the client (the client socket on client-side).
- In other words: the server socket works as a kind of "concierge" which accepts new clients and dispatches each new client to a client socket which is then responsible for the actual data exchange with the client's client socket.
- **Remember that you need to create a client socket object in your client program as well as in your server program.**

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- Note that no client IP address or client host name is specified. Any client host could connect to this server using just the server's IP address (or host name) and the port number (…provided that network access is granted by the firewall and the Java Security Manager)
- The server's IP number isn't assigned by the programmer of the server code (but by the network provider), just the clients need to know the server's IP address.
- In this lecture, clients use 127.0.0.1 a.k.a. localhost
  (this means the server runs on the same computer as the clients, which is useful for testing and security purposes)

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- Often, it is desired that a single server can communicate with multiple clients <u>simultaneously</u> (concurrently). This is achieved using multithreading:

```
while(true) {
    Accept a new client connection;
    Start a new thread which deals with that client (handling its requests, sending responses...);
}
```

- Each thread is responsible for serving exactly one client. All client requests and responses are handled concurrently.

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- In this regard, remember that objects or other resources which might be accessed concurrently by two or more client threads are <u>shared resources</u> (→multithreading).
- This means that code which accesses such needs to be properly synchronized in order to avoid race conditions

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- An example for a client/server application in Java:
  a simple online conversion application. Clients send a number to the server which represents kilograms, or vice versa. Server sends back the corresponding number of pounds or kilograms, respectively:

- Now we use sockets with a self-defined application-layer protocol:

Client request	Server response	Meaning
`CONVERT_TO_KGS` number	number/2.2	pound→kg
`CONVERT_TO_POUNDS` number	number*2.2	kg → pound
`QUIT` (not implemented in example code below!)	None	Close the connection to client

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- We look at a version of the server which doesn't support the concurrent handling of multiple clients...

- However, it supports the *serial* handling of multiple clients...
- This means that once the server has finished dealing with one client, it can accept a new client, and so on.
- Extending such a server to a server which handles multiple clients concurrently (i.e., at the same time) is straightforward
  Basic idea: once a new client has connected to the server, start a new thread which handles all the communication with that client.

NETWORKING AND CLIENT/SERVER

Client/Server sockets

```
public class ConversionServerNoConcurrency {

  private Socket s;
  private Scanner in;
  private PrintWriter out;

  public static void main(String[] args) throws IOException {

      ServerSocket server = new ServerSocket(8888);

      ConversionServerNoConcurrency serverInstance
              = new ConversionServerNoConcurrency();

      System.out.println("Server running. Waiting for a client to
  connect...");
```

NETWORKING AND CLIENT/SERVER

Client/Server sockets

(method main continued)

```
        while(true) {

                serverInstance.s = server.accept();

                System.out.println("Client connected");

                serverInstance.run();

                System.out.println("Client disconnected. Waiting for a new
  client to connect...");

        }

  }
```

NETWORKING AND CLIENT/SERVER

Client/Server sockets

```
public void run() {

     try {
          try {
               in = new Scanner(s.getInputStream());

               out = new PrintWriter(s.getOutputStream());

               doService(); // the actual service

          } finally {
               s.close();
          }

     } catch (IOException e) {
          System.err.println(e);
     }
}
```

NETWORKING AND CLIENT/SERVER

Client/Server sockets

```
public void doService() throws IOException {

    while(true) {

            if(!in.hasNext())
                   return;

            String request = in.next();

            System.out.println("Request received: " + request);

            // (...) test for type of request here (not
implemented)

            handleConversionRequest(request);
    }
}
```

NETWORKING AND CLIENT/SERVER

Client/Server sockets

```
  public void handleConversionRequest(String request) {

        String amountStr = in.next();
        double amount = Double.valueOf(amountStr);
        System.out.println("Received from client: " + amount);

        if(request.equals("CONVERT_TO_POUNDS")) {
                out.println(amount * 2.2d); //server response
                System.out.println("Sending conversion result to client");
        } else if(request.equals("CONVERT_TO_KGS")) {
                out.println(amount / 2.2d); //server response
                System.out.println("Sending conversion result to client");
        } else
                System.err.println("Unknown request!");
        out.flush();    }

}
```

NETWORKING AND CLIENT/SERVER
Client/Server sockets

- On the next slide, find an example <u>client</u> program for this application (a different program)
- This client works both with a multithreaded and the single-threaded version of our server

- Since it connects with server via the Internet, the client code could be put in a different project on a different computer (however, to test the client/server software, we use `localhost` (means that the server is assumed to run on the same computer=host as the clients)

- Make sure the port number for the sockets are the same both in server and client (here: 8888)!

NETWORKING AND CLIENT/SERVER

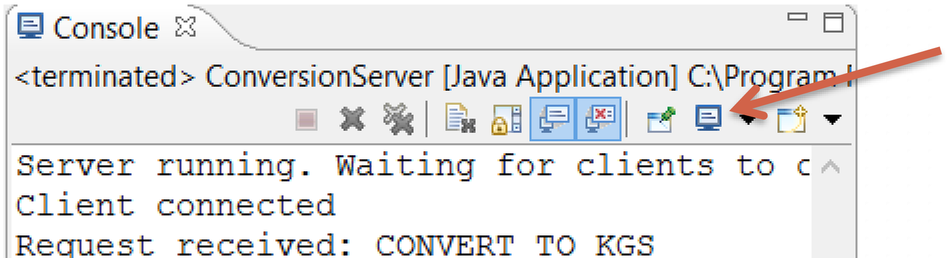
Client/Server sockets

```
public static void main(String[] args) throws IOException {
     Socket s = new Socket("localhost", 8888);
     InputStream instream = s.getInputStream();
     OutputStream outstream = s.getOutputStream();
     Scanner in = new Scanner(instream);
     PrintWriter out = new PrintWriter(outstream);
     String request = "CONVERT_TO_KGS 123.45\n";
    out.print(request);
     out.flush();
     String response = in.nextLine();
     System.out.println("Receiving: " + response);
    s.close();
}
```

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- To test the previous client/server program, the server needs to be started first, then the client(s)
- If a multithreaded server is running, multiple clients can connect to it even *simultaneously* (otherwise, clients can connect to that server only sequentially)
- The clients can be in different Eclipse projects or in the same project - doesn't matter
- Note that the server and each client write their console output (using `System.out.println()`) into different consoles (you need to switch between them in Eclipse using:



<terminated> ConversionServer [Java Application] C:\Program

```
Server running. Waiting for clients to c
Client connected
Request received: CONVERT TO KGS
```

NETWORKING AND CLIENT/SERVER
Client/Server sockets

- There is a potential problem with the socket code presented so far:
  Reading from a stream obtained from a socket <u>blocks</u> until data is available. So, if the host is unreachable, the program potentially blocks forever…
- Solution: *socket timeouts*

```
socket.setSoTimeout(10000); // 10 secs
```

- With this, a `java.net.SocketTimeoutException` is thrown if the timeout is reached before the operation is completed

NETWORKING AND CLIENT/SERVER

Client/Server sockets

- Example:

```
Socket socket = new Socket();
socket.setSoTimeout(10000); // 10 secs
...
try {
    Scanner in = new Scanner(socket.getInputStream());
    String line = in.nextLine(); // blocks
    ...
} catch(java.io.InterruptedIOException e) {
    //SocketTimeoutException is derived
    //  from InterruptedIOException
    ...
}
```