ID2212 Network Programming with Java
Lecture 4

Networking with Sockets

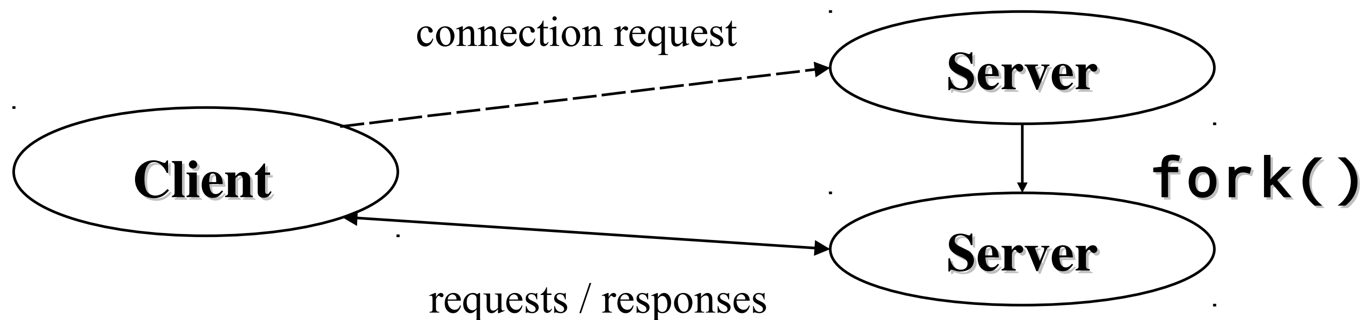
Leif Lindbäck, Vladimir Vlassov
KTH/ICT/SCS
HT 2015

Outline

- Review
  - Client-server architecture
  - Berkley Socket API in C
- Socket API in Java
  - Client Socket – connecting socket
  - Socket for Servers – listening socket
  - UDP sockets
  - Multicast

Review: Client-Server Architecture

- The most commonly used model for distributed applications
  - Can be applied for a particular request-response interaction
- The ***client*** is the entity (process) accessing the remote resource and the ***server*** provides access to the resource.
- Request / response protocols



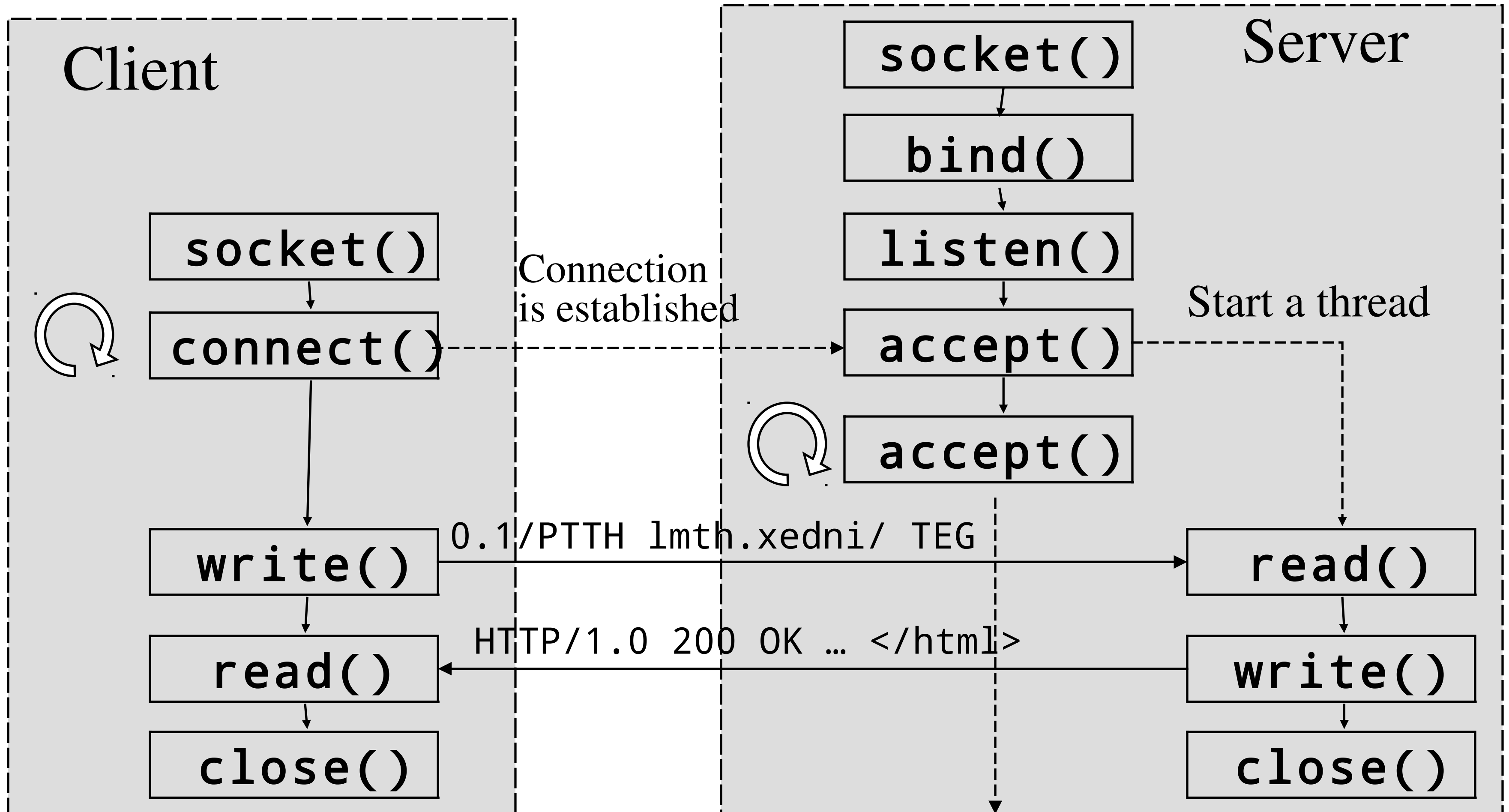
Review: Sockets

- ***Socket*** is an end-point of a virtual network connection between processes – much like a full-duplex channel
  - A socket address: IP address and a port number
  - A transport protocol used for communication over a socket
- ***TCP socket*** - stream-based, connection-oriented
- ***UDP socket*** - datagram-based, connectionless
- Sockets, a.k.a. ***Berkeley sockets***, were introduced in 1981 as the Unix BSD 4.2 generic API for inter-process communication
  - Earlier, a part of the kernel (BSD Unix)
  - Now, a library (Solaris, MS-DOS, Windows, OS/2, MacOS)

Ports

- ***Port*** is an entry point to a process that resides on a host.
- 65,535 logical ports with integer numbers 1 - 65,535
- A port can be allocated to a particular service:
    - A server listens the port for incoming requests
    - A client connects to the port and requests the service
    - The server replies via the port.
- Ports with numbers 1-1023 are reserved for well-known services.
  - A list of services and allocated ports is stored in
    - /etc/services (Unix)
    - C:\Windows\services (Windows95)
    - C:\WINNT\system32\drivers\etc\services (WindowsNT)
    - C:\WINDOWS\system32\drivers\etc (Windows XP)

The Berkeley Socket API for the Client-Server Architecture



Sockets in `java.net`

- Two classes of TCP sockets
- `Socket` – connecting socket, a.k.a. client socket
  - Used to connect to another (remove) TCP socket specified by an IP address and a port number.
  - A connected TCP socket provides two sequenced byte streams, input and output streams, used to communicate with the remote process by reads and writes.
- `ServerSocket` – listening socket, a.k.a. server socket
  - Used to listen for connection requests, to accept connection and create a `Socket` object connected to the requester.

Sockets in `java.net` (cont'd)

- Two classes of UDP sockets:
- `DatagramSocket`
  - used for sending and/or receiving datagrams represented by objects of the `DatagramPacket` class.
- `MulticastSocket`
  - is a subclass of `DatagramSocket` with capabilities for joining multicast groups on the Internet.
  - A multicast group is identified by an IP address of class D (multicast address (224-239.x.x.x)

java.net.InetAddress

- Represents an IP address of a node on the Internet.
- Does not have public constructors.
- Getting an IP address:

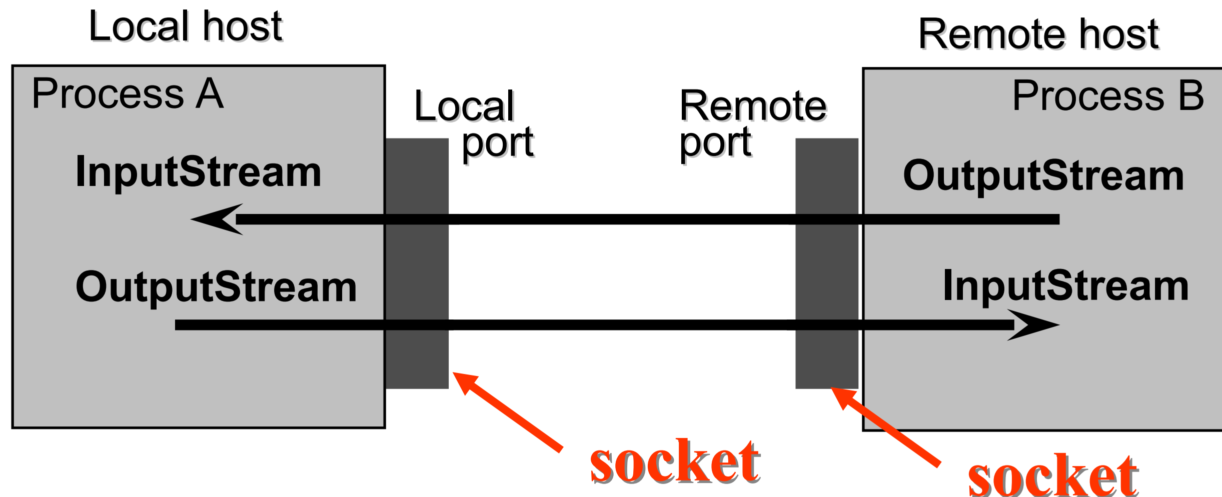
```
InetAddress ip1, ip2, localIP;
InetAddress[] ips;
try {
  // IP address of the local host
  localIP = InetAddress.getLocalHost();
  // IP address(es) of a remote host
  ip1 = InetAddress.getByName("it-gw.it.kth.se");
  ip2 = InetAddress.getByName("130.237.214.1");
  ips =
  InetAddress.getAllByName("130.237.214.1");
} catch ( UnknownHostException e ) {
  e.print.StackTrace();
};
```

Client Sockets – Connecting Sockets

`java.net.Socket`

java.net.Socket

- Implements a connecting TCP socket that provides connection to a specified host on a specified port.
  - When connected, provides input and output byte streams



Socket Constructors

Socket(…)

```
├─(String remoteHost, int remotePort)
├─(InetAddress remoteAddr, int remotePort)
├─(String remoteHost, int remotePort,
│   InetAddress localAddr, int localPort)
├─(InetAddress remoteAddr, int remotePort,
│   InetAddress localAddr, int localPort)
├─( )
└─(SocketImpl socketImplementation)
```

Socket's Attributes

- `setSoLinger(boolean, int)`
  - Enable/disable SO_LINGER with the specified linger time (linger on close if data are present).
    - Note: Use **netstat** utility to check open connection.
- `setSoTimeout(int)`
  - Enable/disable SO_TIMEOUT with the specified time-out, in milliseconds.
- `setTcpNoDelay(boolean)`
  - Enable/disable TCP_NODELAY (disable/enable Nagle's algorithm).

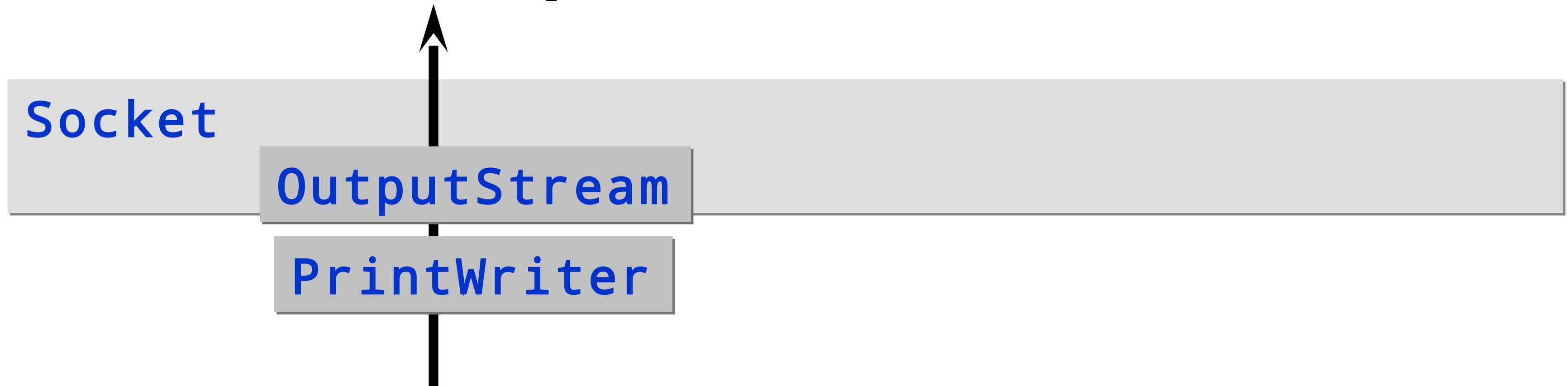
Communicating via a TCP Socket

- Steps:
  - Establish a socket connection to the specified host on the specified port by create a connected `Socket` object.
  - Set socket's attributes.
  - Get an input stream of the socket connection for reading data.
  - Get an output stream of the connection for writing data.
  - Communicate via the input and output streams by reads and writes according to an application specific communication protocol.
  - Close the socket connection.

Example: A Code Fragment from a HTTP Client

```
try {
    Socket socket = new Socket(host, port); // create a connected
 socket
    socket.setSoTimeout( 10000 ); // set 10 sec timeout
    PrintWriter wr = new PrintWriter(socket.getOutputStream()); //
 output stream
    wr.println("GET " + file + " HTTP/1.0"); // send GET request
    wr.println();
   wr.flush();
    BufferedReader rd = new BufferedReader( new
        InputStreamReader(socket.getInputStream())); // input stream
    String str;
    while ((str = rd.readLine()) != null) // receive and print response
      System.out.println(str);
    socket.close();  // close connection
  } catch (IOException e) {
    System.err.println(e);
  }
```

Communication in the `Httpc` Client (Example 3.6)




```
Socket socket = new Socket(host, port);
PrintWriter wr = new PrintWriter(socket.getOutputStrea
wr.println("GET " + file + " HTTP/1.0");
wr.println();
wr.flush();
```

The `Httpc` Client (cont)

Reply: "HTTP/1.1 200 OK
Date: Mon, 15 Nov 1999 11:23:29 GMT …"

Socket

InputStream

InputStreamReader

BufferedReader

…adLine

…ng str

System.out

…intln

```
BufferedReader rd = new BufferedReader( new
 InputStreamReader(socket.getInputStream()));
 String str;
 while ((str = rd.readLine()) != null)
    System.out.println(str);
socket.close();
```

Examples: Clients

- See examples at:

http://www.ict.kth.se/courses/ID2212/examples/html/

- Example 3.2: Looking at the assigned ports `LookAtPorts.java`
  - The application tries to connect to the specified host on the assigned ports (1, …, 1023) and informs if the connection to the port is established. Being connected, it closes the connection.
- Example 3.3: Check TCP connection `CheckConnection.java`
  - The application (applet) tries to connect to the specified host on the specified port and informs whether the connection is established. Being connected, the application closes the connection.
  - The example illustrates a security sandbox for applets in a browser: *An applet can connect only to the host from where it is originated.*

Examples: Clients

- See example at:

http://www.imit.kth.se/courses/ID2212/examples/html/

- Example 3.4: A day-time client `TimeClient.java`
  - The client connects to a specified host on port 13 (a day-time server), reads data from an input stream of the connection and prints it out.
- Example 3.5: A finger client `Finger.java`
  - The client connects to a finger server (on port 79), sends request, gets a response and prints it.
- Example 3.6: An HTTP Web Client `Httpc.java`
  - The client connects to an http server specified in an URL, sends GET request, receives an HTML document pointed to by the URL and prints it out.

Parsing Textual Data by Tokenizers

- `java.io.StreamTokenizer` supports splitting a character stream into "tokens".

```
StreamTokenizer rd = new StreamTokenizer (
                                  new BufferedReader(
                                  new InputStreamReader(

   socket.getInputStream())));
```

- `java.util.StringTokenizer` supports splitting a string into "tokens".
  - `StringTokenizer(String)`
  - Can be used to parse a text from the input stream accumulated into a `StringBuffer` object.

Example: Usage of Stream Tokenizer

```
try {
        Socket socket = new Socket(host, port);
        socket.setSoTimeout( 10000 );
        PrintWriter wr = new PrintWriter( socket.getOutputStream() );
        wr.println( "GET " + file + " HTTP/1.0“ );
        wr.println();
        wr.flush();
        StreamTokenizer rd = new StreamTokenizer( new BufferedReader
    ( new
            InputStreamReader(socket.getInputStream())));
        int num = 0, word = 0, i = 0;
        while ( rd.nextToken() != StreamTokenizer.TT_EOF)
          switch (rd.ttype) {
               case StreamTokenizer.TT_NUMBER: num++; break;
               case StreamTokenizer.TT_WORD: word++;
               if (++i % 5 == 0) System.out.println(rd.sval);
               else System.out.print(rd.sval + " ");
          }
          System.out.println();
          System.out.println("NUM = " + num);
          System.out.println("WORD = " + word);
          socket.close();
      } catch (IOException e) {
          System.err.println(e);
      }
```

Examples

- Example 3.7: `Httpt.java` – A HTTP client with `StreamTokenizer`
  - The modified Web client uses `StreamTokenizer` to read and to count tokens from the input stream of the socket connection
- Example 3.8: `Https.java` – A HTTP client with `StringBuffer`
  - The modified Web client uses `StringBuffer` to accumulate strings read from a socket input stream in to one string

- See example at:

http://www.imit.kth.se/courses/ID2212/examples/html/

Sockets for Servers – listening sockets

`java.net.ServerSocket`

`java.net.ServerSocket`

- Implements a listening TCP socket, a.k.a. server socket
  - should be bound to some known local port (and known local IP address)
  - used to listen and accept connections from clients.

`ServerSocket(…)`

- `(int port)`
  A port of 0 creates a socket on any free port.
- `(int port, int backlog)`
  Here `backlog` is the maximum allowed length of queue of pending connection requests.
- `(int port, int backlog, InetAddress bindLocalAdd`

Accepting Connections

```
Socket clientSocket = serverSocket.accept();
```

- Blocks the current thread until a client connects
- Returns a connected `Socket` of the accepted connection.

- For example:

```
// create a server socket bound to the port 8080
ServerSocket serverSocket = new ServerSocket( 8080 );
while (true) {
    try {
        // wait for a client connection request
        Socket clientSocket = serverSocket.accept();
        // communicate with a client via clientSocket
        …
        // close the socket and wait for another connection
        clientSocket.close();
    } catch (SocketException e) { e.printStackTrace(); }
}
```

Handling Connections

- The server uses a `Socket` object to communicate with a connected client
  - The connection should be closed when service is done.
- The server may handle the connection in a separate thread.
  - When a client connects, constructs a handler thread with the `Socket` object as a parameter of the handler constructor.
  - Starts the handler thread.
  - The parent thread continues waiting for the next connection requests.

Fragment of a Multithreaded Server

```
    ServerSocket serversocket = new ServerSocket( 8080 );
    while (true) {
       try {
           Socket socket = serversocket.accept();
           Handler handler =  new Handler( socket );
           handler.setPriority( handler.getPriority() + 1 );
           handler.start();
       }
       catch (SocketException e) { e.printStackTrace(); }
    }
    . . .
}
class Handler extends Thread {
   private Socket socket;
   Handler(Socket socket) throws IOException { // thread
constructor
   this.socket = socket;
       ... }
   public void run() { // communicate with the client via the
socket
   ...
   }
}
```

Working with Files. Status of a File

- Very often a server accesses files and/or databases.
- `java.io.File` allows obtaining the status of a file or directory:

```
String basedir =
   “/afs/it.kth.se/misc/info/www/documents“;
File file = new File(basedir, “index.html”);
```

- Useful methods
  - Get the status of the file:

    `exists(), length(), lastModified(), canRead(), canWrite(), getPath()`
  - Get a list of files in the directory:

    `list(), list(FilenameFilter)`
  - Manipulate a file, directory:

    `mkdir(), delete(), renameTo(File)`

Two APIs to Access Files. See `java.io`

- Sequential-access file:
  - `FileInputStream` and `FileOutputStream`
    - represent a file as a byte streams that can be wrapped by any specialized stream:

```
DataInputStream in = new DataInputStream ( new
  FileInputStream("base.tsv"));
```

  - `FileReader` and `FileWriter` are used to access text files.

- Random-access file (similar to file API in C):
  - An object of `RandomAccessFile` allows reading (or writing) data of various types from (or to) a file.

Access a Database

- To access a data source (database), a server can use JDBC
- A code fragment:

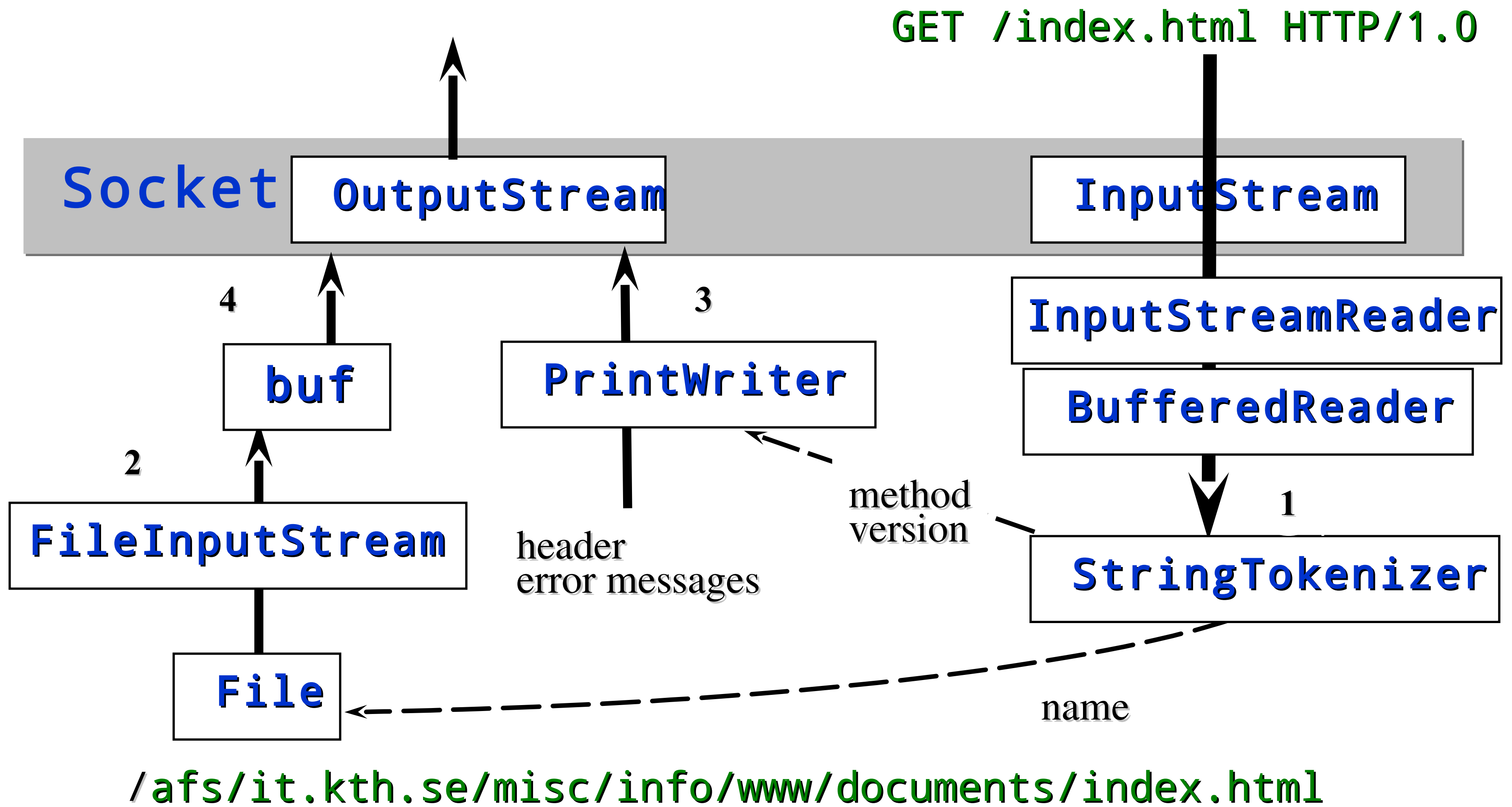
```
BufferedReader in = new BufferedReader( new
   InputStreamReader( socket.getInputStream()));
...
// load a JDBC driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
// open connection to the database NOG
Connection con =
   DriverManager.getConnection("jdbc:odbc:NOG","Admin", "");
// create an SQL statement
Statement stmt = con.createStatement();
// execute SQL query and get result set
ResultSet rs = stmt.executeQuery("SELECT * FROM
   participants " + in.readLine());
```

Examples: Servers

- Example 3.9: A Day-Time Server `DayTime.java`
  - The server waits for a client connection request.
  - When a connection is accepted, the server sends a response with the date and time at the server in a one-line ASCII string.
- Example 3.10: A Simple HTTP Server `Httpd.java`
  - Accepts GET requests and POST requests. Assumes that only a POST request can request the execution of a CGI program
  - The example illustrates how to start the execution of an external program (process) and communicate with the program via standard input and output streams. The example also illustrates how to access a file.
- See example at:

http://www.ict.kth.se/courses/ID2212/examples/html/

Data flow in Httpd : GET Request



Connection Failures

- Connection failures are signaled via exceptions thrown by methods of socket APIs (constructors, connect, accept, read, write, bind, etc.)
- Exceptions that indicate errors in the underlying protocol, such as a TCP error, are indicated by `SocketException` and its subclasses:
  - `BindException`
    - on failed attempt to bind a socket to a local address and port.
  - `ConnectException`, `NoRouteToHostException`
    - on failed attempt to connect a socket to a remote address and port.

(cont'd) Connection Failures

- The client can get exceptions
  - `UnknownHostException`
    - The IP address of a host could not be determined (getters of InetAddress)
  - `IOException` (subclass `ProtocolException`)
    - while creating the socket (Socket constructors) and communicating via the socket connection (getInputStream, getOutputStream, read/write, etc.)
- The server can get exceptions
  - `IOException` (subclass `ProtocolException`)
    - opening a server socket (ServerSocket constructors)
    - waiting for a connection and accepting the connection in accept()
    - closing the socket by close()

JSSE (Java Secure Socket Extension)

- A set of Java packages that enable secure Internet communications.
  - Implements a Java version of SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols
  - Includes functionality for data encryption, server authentication, message integrity, and optional client authentication.
  - `javax.net.ssl`
  - `javax.net`
  - `java.security.cert`
  - `com.sun.net.ssl`
- JSSE on the Web:

  http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html

  http://download.oracle.com/javase/7/docs/technotes/guides/security/

UDP Sockets. Multicast

```
java.net.DatagramSocket
java.net.DatagramPacket
java.net.MulticastSocket
```

UDP Sockets

- The `java.net.DatagramSocket` class
  - represents a UDP socket for sending and receiving datagrams – objects of the `java.net.DatagramPacket` class

- Sending datagrams:

```
DatagramSocket ds = new
    DatagramSocket();
byte buf[] = new byte[256];
// fill buf with data to be sent
…
// create a datagram
DatagramPacket dp = new DatagramPacket(
    buf, buf.length,
    InetAddress.getByName("dest.host.com
    "),
    4711);
// send the datagram via the UDP socket
ds.send(dp);
```

- Receiving datagrams:

```
byte b[] = new byte[256];
DatagramPacket dp = new DatagramPacket(b,
    b.length);
/* Set timeout – the amount of time (in
    milliseconds) that receive() waits for
    datagram before throwing an
    InterruptedIOException. With the time out
    of 0, receive() never times out.
*/
ds.setSoTimeout(timeout)
ds.receive(dp); // receive a datagram
byte[] data = dp.getData(); // get data
InetAddress source = dp.getAddress(); //
    source
int port = dp.getPort(); // source port
```

IP Multicast

- ***IP multicast*** is communication within a multicast group identified by a multicast IP address of the D class:

  `224-239.x.x.x`

- ***A multicast group*** is a set of computers sharing the same multicast address.
  - ~80 of multicast addresses are permanently assigned by the IANA (Internet Assigned Number Authority).
- To receive datagrams directed to a multicast group, a computer joins the group (gets its IP address)
  - Informs a default router about its interest in receiving UDP packets directed to the group's IP address

IP Multicast (cont'd)

- Multicast is based on sending UDP datagrams to a multicast group.
- An IP header of a UDP packet includes the field ***TTL (Time-To-Live)*** that specifies the number of routers that the packet can pass through (in the range 0-255).

MBONE. Videoconferencing on the Internet

- ***MBONE*** (Multicast Backbone on the Internet) is the range of Class D addresses beginning with 224.2.x.x
  - Mbone is a part of the Internet formed of routers supporting the IP multicast extension.
  - Mbone is used for audio and video broadcasts over the Internet.
- The MBONE programs should be announced on 224.2.127.254 (port 9875).
  - See Example 3.11.2: **sdr.java**

Multicast with Java

- `MulticastSocket` is a subclass of `DatagramSocket` that represents a UDP socket with capabilities for joining multicast groups on the Internet.
- Communicating with a multicast group
  - Construct a multicast socket
  - Join a multicast group (for receiving)
  - Send/receive data to/from the multicast group
  - Leave the group

Receiving from a Multicast Group

```
try {
  MulticastSocket ms = new MulticastSocket( 9875 );

  ms.joinGroup(InetAddress.getByName("224.2.127.254")
  );
   while (true) {
     ms.receive(dp);
     String s = new
  String(dp.getData(),0,0,dp.getLength());
     System.out.println(s);
  }
} catch (Exception se) {
  se.printStackTrace();
}
```

Sending to a Multicast Group

```
InetAddress iaddr = InetAddress.getByName( “224.17.17.17” );
DatagramPacket dp = new DatagramPacket(data, data.length,
      iaddr, port);
try {
     MulticastSocket ms = new MulticastSocket();
     ms.setTimeToLive( 16 ); // set TTL to 16
     ms.joinGroup(iaddr); // not necessary for sending
     ms.send(dp);
     ms.leaveGroup(iaddr); // not necessary for sending
     ms.close();
} catch (SocketException se) {
     se.printStackTrace();
} catch (IOException ioe) {
     ioe.printStackTrace();
}
```

Implication for A Course Project

- Client-Server interaction using TCP
  - Client side:
    - Host and port of a server should be command line arguments
    - Do not bind a client socket to a fixed local port number
    - Do not forget to flush the socket output stream when sending a request
    - Remember that read from a socket input stream is blocking
    - Should set timeout for a socket and properly handle exceptions
    - Take into account long communication latency: User interface should be responsive, use multithreading: a thread for the user interface, a thread for the network interface.
    - A Client may create and listen a server socket (e.g. for callbacks). It should do this in a separate "server" thread

Implication for the Course Project (cont'd)

- Client Server interaction using TCP
  - Server side:
    - A server port should be a command line argument
    - A server should be scalable, i.e. it should be able to handle multiple requests simultaneously by using multithreading
    - Two approaches to multithreading:
      - (1) create a new thread for each client connected;
      - (2) assign a thread to a client from a pool of threads
    - While communicating with a client, do not forget to set timeout to a TCP socket, and flush output stream when needed

Implication for the Course Project (cont'd)

- Communication using UDP
  - The same UDP socket can be used for both, sending and receiving
  - A TCP socket and a UDP socket may be bound to the same port number
  - Sending a datagram, set a proper TTL
  - Remember that receive is a blocking call
    - Set timeout and handle exceptions properly
  - You may get source address (IP address and port) from a datagram received

Implication for the Course Project (cont'd)

- Problem with request-response interaction using UDP:
  - if `receive` is interrupted because of timeout, you may treat this as there is no response to the sent request and send a new request. However, next `receive` may receive a response on the old request rather than the one sent recently.
  - Your application should be able to handle this situation.

Implication for the Course Project (cont'd)

- Communication with a multicast group
  - A multicast address(es) and port(s) should be command line arguments.
  - To send to a multicast group, it is not needed to join the group
    - You may use DatagramSocket for sending to the group
  - Responses to one member of a group should be sent directly to the member (to it's IP) rather than to the entire group
  - When your application waits for a response from the multicast group, it should be able to handle two extreme cases
    - No response
    - Too many responses
  - You should also solve the problem of receiving old responses (see previous slide).