Socket Programming

Robin Sharp

Informatics and Mathematical Modelling
Technical University of Denmark
Phone: (+45) 4525 3749
e-mail: robin@imm.dtu.dk

Application Layer protocols

- Are based on a (more or less reliable) Transport service — in the Internet, typically provided by TCP or UDP.
- May support various ways of organising an application. Common examples:
  - Peer-to-peer: Two or more participants with equal status.
  - Client/server: Two participants. One party (server) offers services to the other (client).
  - Agent-based: Several parties collaborate in an “intelligent” way.
  - Grid: (Very) large number of parties offer services, and system will find the most appropriate one.
- We focus here on Client/server systems.

Simple Socket Programming

- Internet Client/server applications are based on communication between suitable ports in the client and server systems.
- **Sockets** offer a programming abstraction of endpoints of communication channels, such as ports.
- Classic example: BSD TCP/IP Sockets, with operations:

Primitive	Semantics	C	S
socket	Create new communication endpoint	+	+
bind	Associate local IP addr. + port with socket	+	+
listen	Announce willingness to accept connections		+
accept	Block caller until conn. request arrives		+
connect	Initiate establishment of connection	+	
write/send	Send data over connection	+	+
read/recv	Receive data over connection	+	+
close	Release connection	+	+

C: in client S: in server

Java Client Sockets

- Objects of the class **`java.net.Socket`**
- Assume underlying communication protocol is TCP.
  - If *UDP* is required, use class **`DatagramSocket`**.
  - If *multicast* is required, use class **`MulticastSocket`**.
- Relation to BSD sockets:
  - socket: Incorporated in class constructor.
  - bind, close, connect: Methods of class **`Socket`**.
  - listen, accept: Not relevant for clients.
  - read, write: Methods of the stream objects associated with the socket.
- Other important methods of the class include:
  - getInputStream: Returns an input (byte)stream for socket.
  - getOutputStream: Returns an output (byte)stream for socket.

Simple Java client

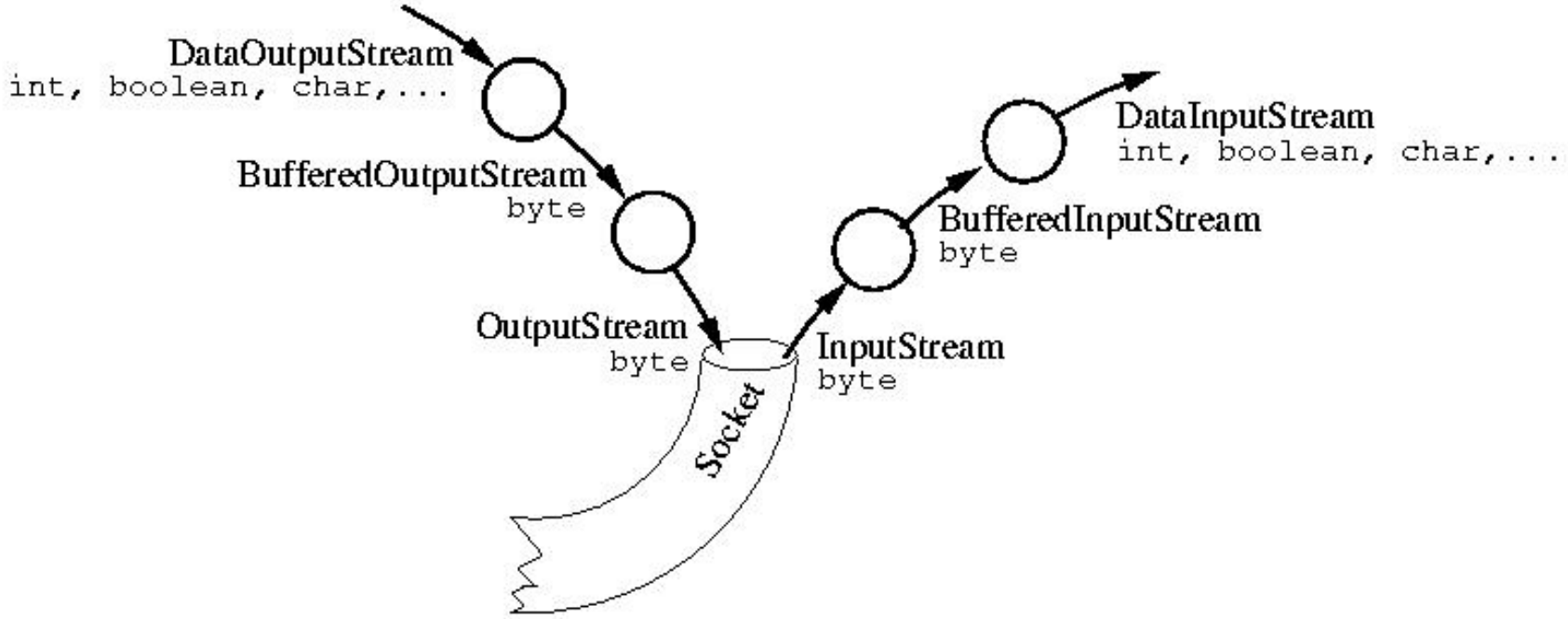
```
int    portno  = p;
String servname= s;
try
{ Socket serv = new Socket(servname,portno);
  System.out.println("Connected to server "+ ...);
  InputStream  si = serv.getInputStream();
  OutputStream so = serv.getOutputStream();
```

... communicate with server via streams si and so.

```
  serv.close();
}
catch(UnknownHostException e)
{System.out.println("Cannot find server "
                    + servname);          }
catch(IOException e)
{System.out.println("Error in connecting to port "
                    + Integer.toString(portno,10)
                    + " on " + servname); }
```

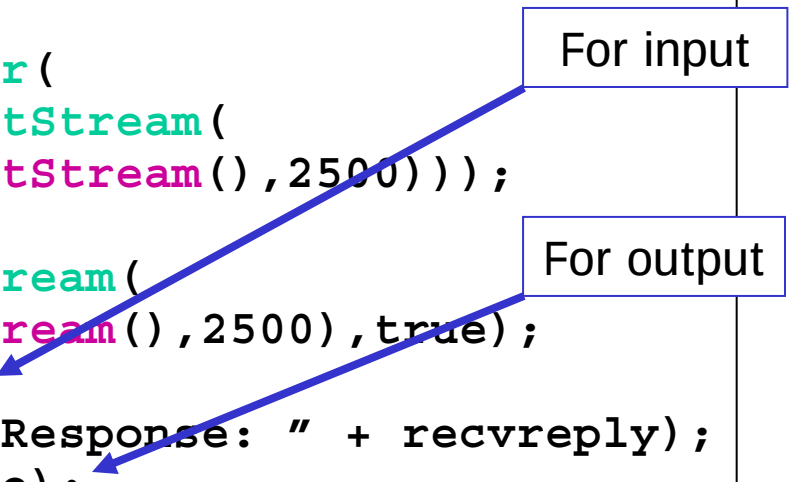
Stream filters

- `InputStream` and `OutputStream` are both unbuffered byte streams -- not usually convenient for use directly in applications.
- More usual to apply one or more stream filters, e.g.:



Client with filtered streams

```
Socket           s1;
PrintStream      p1;
BufferedReader   d1;
String           recvreply;
try
{ s1=new Socket(servname,portno);
  System.out.println("Connected to server " + ... );
  // Set up input and output streams
  d1=new BufferedReader(
          new InputStreamReader(
              new BufferedInputStream(
                    s1.getInputStream(),2500)));
  p1=new PrintStream(
          new BufferedOutputStream(
                s1.getOutputStream(),2500),true);
  recvreply = d1.readLine();
  System.out.println("Server Response: " + recvreply);
  p1.println("HELO " + cliname);
     . . .
```



Java Server Sockets

- Objects of the class `java.net.ServerSocket`
- Assume underlying communication protocol is TCP.
- Relation to BSD sockets:
  - socket: Incorporated in class constructor.
  - bind, close, accept: Methods of class `ServerSocket`. Note that accept creates a `Socket` with associated input and output streams when it accepts an incoming call.
  - connect: Not relevant for servers.
  - read, write: Methods of the stream objects associated with the `Socket` created by accept.
- Other important methods of the class include:
  - setSoTimeout: Set timeout for accept to get incoming call.

Simple Java server

```
int portno  = p;
int timeout = t;
try
{ ServerSocket serv = new ServerSocket( portno );
  serv.setSoTimeout( timeout );
  Socket client = serv.accept();
  System.out.println("Connected to client " + … );
  InputStream  si = client.getInputStream();
  OutputStream so = client.getOutputStream();

    … communicate with client via streams si and so

  client.close();
}
catch (SocketTimeoutException e)
{ System.out.println("Server socket timeout on port "
                     + Integer.toString(portno,10)); }
catch (IOException e)
{ System.out.println("Cannot set up server on port "
                     + Integer.toString(portno,10)); }
```

More useful server

Simple server deals with one call from one client.

For more useful setup, we may need to consider:

- Repetitive execution to deal with consecutive calls?
- Multiple threads to deal with simultaneous handling of clients?
  - Create new thread to deal with each incoming call, kill thread when connection is broken (by client? on inactivity?)
  - Release thread from pool of waiting threads on incoming call, set to wait again when connection is broken.
- Security issues?
  - Who is allowed to do what on which ports?

Java 2 security

Java 2 VM checks programs under execution against a list of **permissions** to access particular resources.

Permissions are typically specified in a **policy file**, and refer to a class of **permission objects** associated with the resource. For example:

```
grant signedBy "grass"{
  permission java.net.SocketPermission "*.dtu.dk:1099",
                                       "connect,accept" };
```

Causes permission object:

```
p1 = new SocketPermission("*.dtu.dk:1099",
                          "connect,accept");
```

allowing connections to port 1099 on any host in domain dtu.dk to be granted to code signed by "grass".

Socket permissions

- Specify:
  - one or more hosts

    `goofy.dtu.dk` `*.dtu.dk`
  - port number interval

    `7777` `1024-49151` `1024-`
  - permitted methods

    `accept`, `connect`, `listen`

    (all these implicitly include `resolve`).
- May be restricted to code originating from a particular source (`signedBy "xxx"`).

Policy files

- **Global policy file**: java.home/lib/security/java.policy
  where java.home is directory containing jre.
  - Allows anyone to listen on non-privileged ports.
  - Allows any code to read standard (non-sensitive) properties, such as os.home, file.separator,...
- **User policy file**: user.home/java.policy
  where user.home is user's home directory.
  - Can be set by user.
  - Default value same as global policy file.
- **Runtime policy file**: Specified by runtime parameter
  `-Djava.security.policy=filename`

  May also need `-Djava.security.manager` to turn on the default security manager!

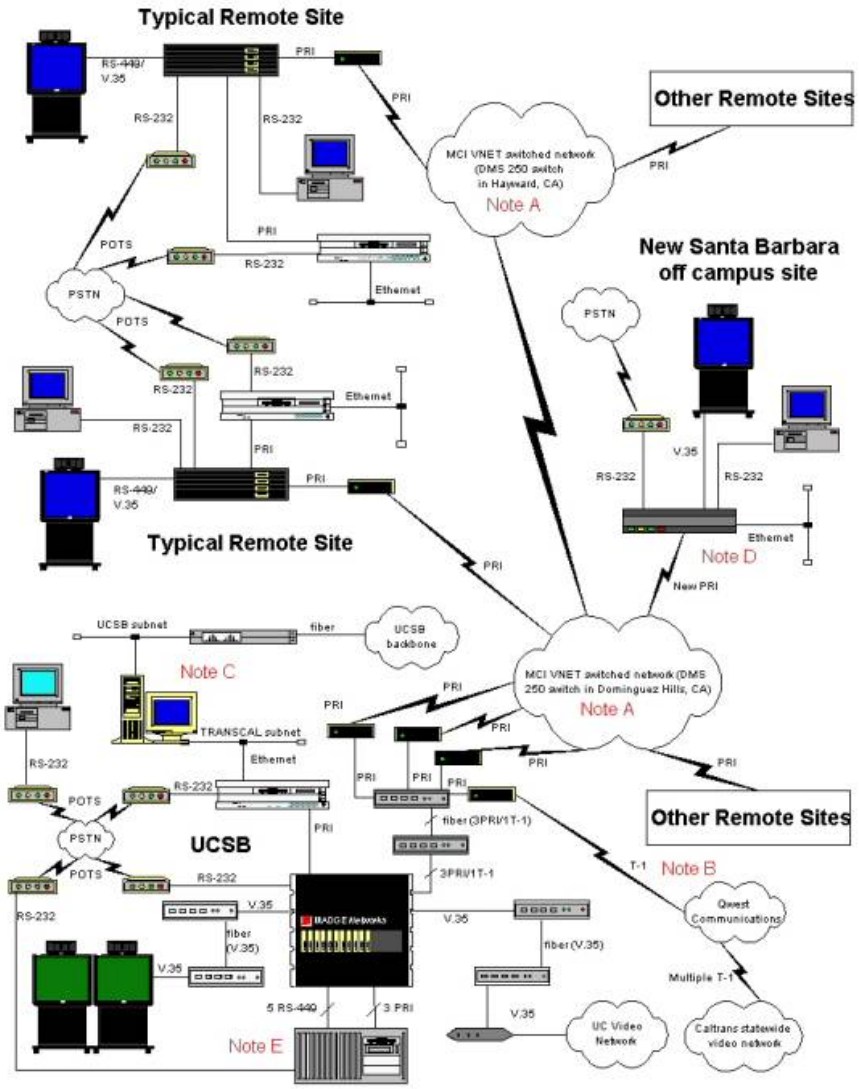
More complex C/S Systems

- Internet mail and browser applications are simple examples. In more complex cases:
  - The client may need to search for a server which offers the required functionality.

    Need a lookup service in the network.
  - Client and server may need to exchange arbitrary data in a more efficient way than by using messages in ASCII text.

    Need an efficient platform-independent manner of representing arbitrary data.
  - The client may need to authenticate itself to the server.

    Need suitable authentication and key distribution mechanisms.
- Such common facilities are often introduced by using a **middleware** layer between T- and A-layers....



Thank you for your attention