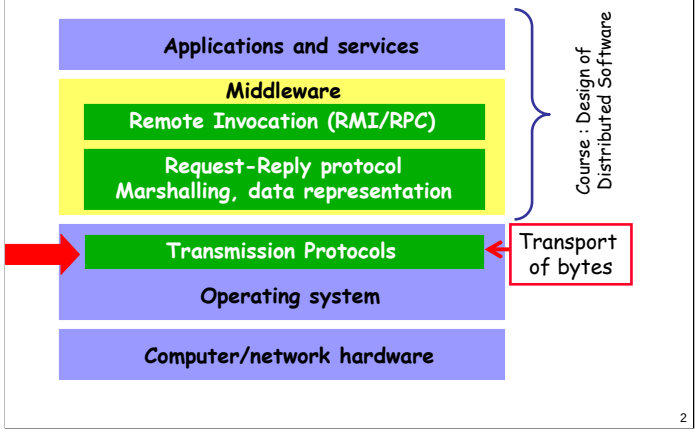
Appendix I

Socket Programming in Java

Layering



Communication between applications is structured in layers, and in fact one has the choice where to implement the communication. At the tramission protocol level, communication is focused on transporting bytes between two processes.

The semantics of these bytes (i.e. what data or method call they represent), is of no importance to the tranmission protocol layer. This layer is only responisble for the proper transport of the bytes between source and destination.

Higher layers will enhance the semantics of the communication. Of course, all communication is eventually handled by some transport protocol.

Interprocess communication

	process 1	process 2
Synchronous	blocks [until receive completed]	idem
Asynchronous	non-blocking	[1] blocking variant [2] non-blocking variant

Possible destinations

- process on local host or remote host
- unicast or multicast messages

3

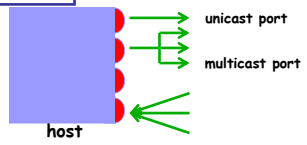
Interprocess communication (IPC) comes in two flavours: a synchronous variant, where the sender blocks until the receiver acknowledges receipt of the complete message (of course, in this case also the receiving end blocks until the complete message received).

In the second variant, the send operation is asynchronous (i.e. the sending process does not block, but simply continues with the task at hand, without waiting for acknowlegdment). In this case, the receiver can be either implemented to block (until the complete message is received) or to receive the message in parallel with current task execution.

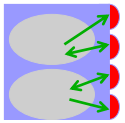
Destinations for IPC can either be local (i.e. another process on the local machine), or remote (i.e. some process on another computer). Most of the communications are unicast communications (one source sends a message to a single destination). Also support multicast communication is typically foreseen (one source sends the same message to a number of destinations).

Port and socket abstractions

Port =
- message destination within a computer
- specified as integer



- can have many senders
- receivers :
  - unicast port : only 1
  - multicast port : many
- 2^{16} possible ports/host
- port specified by [host IP-address, port number]



Process
- uses port(s) to send/receive data
- can NOT share ports with other processes on same host [exception : multicast ports]

4

To uniquely identify a process on a global scale we need to:

•identify the computer the process is running on (the host) : this is done through a globally unique IP-address

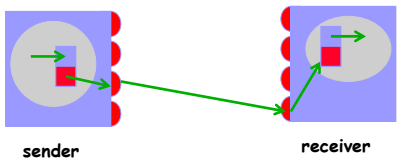
•identify the process on this host : this is done through s 16 bit port number.

A sending process uses a port to identify itself when transmitting a message, and similarly “listens” to a port for messages destinated for the process at hand.

In the case of unicast ports, port-to-process bindings are one-to-one: ports are NOT shared between processes (otherwise the process can not be uniquely identified). A process can however listen to multiple ports, or send messages to multiple ports. In case of a multicast communication, the [IP-address,port-number] combination specifies a group of processes for multicast interaction (and processes effectively share ports).

Port and socket abstractions

Socket =
communication endpoint
in memory space of process



- binds to port [IP-address, port number]
- runs one transport protocol (UDP or TCP)

5

A socket is the physical realization of a communication endpoint: the socket is the area in memory where data, received from the port is put (in the case of receiving a message), and where the port reads the data to be transmitted. A socket therefore binds to a [IP,port] combination (each port has some memory associated with it), and runs a transport protocol. The latter protocol determines how the message transmission is performed (e.g. witt/without retransmissions in case of packet loss, with/without error correction, etc.).

Transmission protocols

UDP : datagram communication

- message size
  - receiving process specifies buffer size
  - maximum message size : 2^{16} bytes – header (UDP+IP)
  - usually : maximum message size = 8 KB
- blocking behavior
  - non-blocking send, blocking receive (timeout can be specified)
  - message discarded if no socket bound to port
- failure model
  - no protection against channel omission failures
    (no re-send !)
  - checksum to detect corrupt datagrams
  - out-of-order delivery possible
- usage
  - low latency applications (e.g. streaming video, VoIP, …)

6

The UDP protocol is a message oriented protocol: a single message is sent between source and destination, after which the communication ends. The main properties of the protocol are summarized above.

Transmission protocols

TCP : stream communication

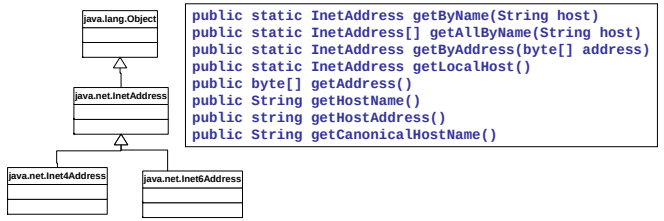
- message size
  - no maximum data size
- two streams per socket pair (one for each direction)
- blocking behavior
  - send : can be blocked by flow control, blocking receive
- failure model
  - resend lost messages (lost of timeout value exceeded)
  - in-order delivery
  - flow control
  - connection broken if no acknowledgement received within given time interval
- usage
  - http, ftp, telnet, SMTP, …

The TCP protocol is stream oriented: a channel is open between two processes (in two directions), and is only closed after issueing a specific closing command on the socket. This protocol is used in cases where data integrity is crucial (e.g. file transfer, web traffic).

Java API : InetAddress

class java.net.InetAddress

- utility class for IP-addresses
- performs DNS lookup
- application code easily upgradable to IPv6



The Java InetAddress class is a utility class able to model IPv4 and IPv6 network addresses. Methods contained in the class allow to perform DNS (i.e. find the IP address, given the logical name) and reverse DNS.

Java API : InetAddress

```
import java.net.*;
class Address {
   public static void main(String[] args) {
      try {
        InetAddress localHost=InetAddress.getLocalHost();
        System.out.println("Local host = "+localHost);
        System.out.println("Local host name = "+localHost.getHostName());
        System.out.println("Local host address = "+localHost.getHostAddress());
        InetAddress ugent=InetAddress.getByName("www.ugent.be");
        System.out.println("UGent host = "+ugent);
        InetAddress unknown=InetAddress.getByName("157.193.196.229");
        System.out.println("Unknown hostname = "+unknown.getHostName());
      } catch(UnknownHostException e) {System.err.println("Unknown host.");}
   }
}
```

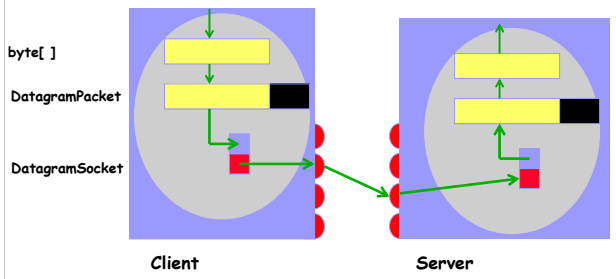
DNS lookup →
Reverse DNS lookup →

```
Local host = yourcenar/157.193.122.154
Local host name = yourcenar
Local host address = 157.193.122.154
UGent host = www.ugent.be/157.193.40.33
Unknown hostname = adhemar.ugent.be
```

9

Sample usage of the methods of the Java InetAddress class.

Java API : Datagram Communication



10

In case of datagram communication (UDP), a Javabyte array is put into an object of the class DatagramPacket, also containing header information (mainly containing source and destination for the datagram at hand). This object is handed to an object of the class DatagramSocket, which transmits the data over the network.

At the destination side, a second DatagramSocket restores a DatagramPacket from the bytes received. The header is stripped, and the original byte array is reconstructed.

Java API : UDP Client

```
import java.net.*;
import java.io.*;
class UDPClient {
   public static void main(String[] args) {
      byte[] message=args[0].getBytes();
      InetAddress host=null;
      int serverPort=1234;
      DatagramSocket socket=null;
      try {
         host =InetAddress.getByName(args[1]);
         socket=new DatagramSocket();
         DatagramPacket request=new DatagramPacket(message,message.length,host,serverPort);
         socket.send(request);
         byte[] buffer=new byte[50];
         DatagramPacket reply=new DatagramPacket(buffer,buffer.length);
         socket.receive(reply);
         System.out.println("Reply from server = "+(new String(reply.getData())));
      } catch(UnknownHostException e) {System.err.println(e);
      } catch(SocketException e) {System.err.println(e);
      } catch(IOException e) {System.err.println(e);
      } finally {if(socket!=null) socket.close();}
   }
}
```

Sending a message
1. Resolve destination
2. Create socket
3. Create datagram
4. Send datagram to socket
5. Close socket

Receiving a message
1. Create socket
2. Create receive buffer
3. Create empty datagram
4. Receive datagram from socket
5. Close socket

11

A sample application : sending a String object to a server. The latter transforms the strong to capitals, and sends this result back.

A UDP client is shown. Note that we send the message to a server, located at port 1234. After constructing the DatagramSocket object, a request object (of the class DatagramPacket) is built, containing the message payload (the String message) as well as the destination (host and serverPort).

After sending the message, a buffer is created (large enough to store the return message from the server, here we used 50 bytes). After constructing a dummy reply object, the socket waits for return messages. When a datagram is received, the contained data is printed to the console.

Java API : UDP Server

```
import java.net.*;
import java.io.*;

class UDPServer {
   public static void main(String[] args) {
      DatagramSocket socket=null;
      int serverPort=1234;
      byte[] buffer=new byte[50];
      try {
         socket=new DatagramSocket(1234);
         while(true) {
            DatagramPacket request=new DatagramPacket(buffer,buffer.length);
            socket.receive(request);
            String rMess=(new String(request.getData())).toUpperCase();
            DatagramPacket reply =new DatagramPacket(rMess.getBytes(),rMess.length(),
                                  request.getAddress(),request.getPort());
            socket.send(reply);
         }
      } catch(SocketException e) {System.err.println(e);
      } catch(IOException e) {System.err.println(e);
      } finally {if(socket!=null) socket.close();}
   }
}
```

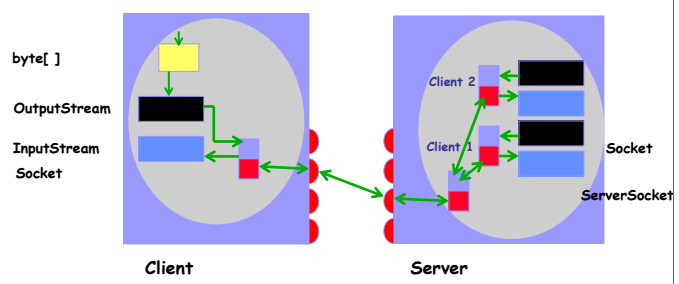
Processing a message
1. Create socket at fixed server port
2. Create receive buffer
3. Endless loop :
   1. Create empty datagram
   2. Read message from socket
   3. Process message
   4. Create reply datagram
   5. Write reply datagram to socket

12

The server side for the “convert-to-capitals” applications. A socket is created, listening at port 1234 (the agreed server port). A dummy request message is constructed, again using a large enough buffer. The socket blocks until a message is receivet. The data contained in the datagram is converted to upper case, after which a reply datagram is constructed. Note that the destination for this datagram is the source of the request datagram.

The endless-loop construct is typical for server side programs.

Java API : Stream Communication



In case of stream communication (using TCP) the situation is slightly more complex. Now, an object for input (receipt) and output (sending) of messages is created. Both objects are bound to the TCP-socket at the client side.

As the stream remains open, unless explicitely closed, the server has to maintain a socket PER CLIENT. To this end, a special object of the class ServerSocket is created. This object listens to incoming requests from new clients, and creates a dedicated socket object PER CLIENT.

Java API : TCP Client

TCP client structure

1. Resolve destination
2. Create socket
3. Retrieve input and output byte stream
4. Convert byte streams to more suitable streams
5. Read and write to streams
6. Close socket

```
import java.net.*;
import java.io.*;

class TCPClient {
   public static void main(String[] args) {
      InetAddress host=null;
      int serverPort=1234;
      Socket socket=null;
      try {
         host =InetAddress.getByName(args[1]);
         socket=new Socket(host,serverPort);
         DataInputStream in=new DataInputStream(socket.getInputStream());
         DataOutputStream out=new DataOutputStream(socket.getOutputStream());
         out.writeUTF(args[0]);
         System.out.println("Reply from server = "+(in.readUTF()));
      } catch(UnknownHostException e) {System.err.println(e);
      } catch(IOException e) {System.err.println(e);
      } finally {if(socket!=null) try{socket.close();}catch(IOException e) {
                                                      System.err.println(e);}}
   }
}
```


The same application is shown, now using TCP as transport protocol. A socket is created (instead of a DatagramSocket) for the specific destination of the stream. References to the stream objects of the socket are identified (the variables in and out in the sample code shown above). Writing to this out-object, results in tranmitting the bytes to the server. Reading causes the socket to block until a message is received from the server.

Java API : TCP Server

TCP server structure

1. Create server socket at fixed server port
2. Endless loop
   1. Listen for request and create client socket
   2. Create suitable input/output streams for client socket
   3. Read input from input stream
   4. Process request
   5. Write reply to output stream
   6. Close client socket

```
import java.net.*;
import java.io.*;

class TCPServer {
  public static void main(String[] args) {
  Socket client=null;
  int serverPort=1234;
  try {
        ServerSocket listen=new ServerSocket(serverPort);
        while(true) {
              client=listen.accept();
              DataInputStream in=new DataInputStream(client.getInputStream());
              DataOutputStream out=new DataOutputStream(client.getOutputStream());
              String input=in.readUTF();
              out.writeUTF(input.toUpperCase());
        }
        } catch(IOException e) {System.err.println(e);
        } finally {if(client!=null) try{client.close();}catch(IOException e){System.err.println(e);}}
  }
}
```

Blocks until request received

Blocks other clients

The server side is slightly complicated, due to the stream-nature of the communication. First a ServerSocket is created, on which the accept()-method is invoked. This method blocks until a new client is announced. As soon as a new connection is discovered, the accept()-method returns a reference to the socket-object at server side, especially created for this new client. Reading and replying from/to the socket resembles the client side.

Note that handling the request blocks. So, while receiving one request, it is not possible to process another client's request. To solve this, multithreading is required.

Java API : TCP Multi-threaded Server

```
import java.net.*;
import java.io.*;

class TCPMultiServer {
        public static void main(String[] args) {
                Socket client=null;
                int serverPort=1234;
                try {
                        ServerSocket listen=new ServerSocket(serverPort);
                        while(true) {
                                client=listen.accept();
                                ClientConnection c=new ClientConnection(client);
                        }
                } catch(IOException e) {System.err.println(e);
                }
        }
}
// …
```

Thread object !!!

Java API : TCP Multi-threaded Server

```
// ...
class ClientConnection extends Thread {
        private DataInputStream in;
        private DataOutputStream out;
        private Socket client=null;
        public ClientConnection(Socket s) {
                client=s;
                try {
                        in=new DataInputStream(client.getInputStream());
                        out=new DataOutputStream(client.getOutputStream());
                        start();
                } catch(IOException e) {System.err.println(e);}
        }
        public void run() {
                try {
                        String input=in.readUTF();
                        out.writeUTF(input.toUpperCase());
                } catch(IOException e) {System.err.println(e);
                } finally {if(client!=null) try{client.close();}catch(IOException e){System.err.println(e);}}
        }
}
```