Networking Basics and Sockets

Communication Protocols

Communication in distributed systems takes place only via the exchange of messages

To enable communication, the communication partners have to agree on certain rules

Protocol = Definition of rules and algorithms that conduct the communication of two or more partners

Examples

- Plugs
- How is a sequence of bits is represented in form of voltage levels?
- Send high-level bit first or last?
- How to recognize and handle errors?

-> Rules belong to different levels

-> Structuring in form of layers

Layering: Terms and Concepts

Service

- functionality offered by one protocol layer to the layer above

Protocol

- rules governing the exchange of messages between components in the same protocol layer

Protocol stack

- layering of protocols over multiple levels

Properties

- logical communication on each layer but physical communication only at the bottom layer
- ideally each layer can be exchanged without affecting other layers (transparency)

ISO-OSI Reference-model (since 1977)

Open Systems Interconnection

Basic Layer

- 1: Physical Layer
  - standardization of cables and plugs
  - coding of sequences of bits by means of physical signals
- 2: Data Link Layer
  - correct transmission of data between two directly connected nodes
  - flow control (adjust speed of sender and receiver)
  - arbitration of access to physical media
- 3: Network Layer
  - transmission of data between arbitrary nodes
  - routing
  - multiplexing of connections
- 4: Transport Layer
  - logical connection between processes instead of nodes (sockets)
  - segmentation of messages

ISO-OSI (2)

Higher Layer

- 5: Session Layer
  - grouping of messages according to an application context (for example between login and logout)
  - grouping of multiple transport connections (e.g., video and audio)
  - no practical relevance
- 6: Presentation Layer
  - coding of complex data (type, domain, structure, ...)
  - today mainly handled by middleware
- 7: Application Layer
  - real applications (email, www, ...)

Connections

Connection-oriented communication

- before application data is transmitted, a connection has to be established
- when all data has been transmitted, the connection will be closed
- A connection involves additional overhead but allows powerful mechanisms for error detection and handling, flow control, negotiation of parameters ...

Connectionless communication

- Data can be sent immediately without prior negotiation with the receiver
- Less overhead but no guarantees about correct delivery or order

Internet-Technology

Basic ideas

- global addressing scheme (IP-addresses)
- TCP/IP-protocols + routing techniques + complementary services
- application protocols: email (SMTP, FTP, Telnet, HTTP, ...)

Addresses

- Each node on the internet has a unique IP-address
  - lenght: 32 bit (IPv6: 128 bit)
  - typically written in dotted decimal notation (e.g., 130.83.127.3)
- Addresses consists of a network and a host part
  - Class A: 1 bit Prefix (0); 7 bit netid, 24 bit hostid
  - Class B: 2 bit Prefix (10); 14 bit netid, 16 bit hostid
  - Class C: 3 bit Prefix (110); 21 bit netid, 8 bit hostid
  - today many intermediate sizes of networks exist (subnets)
  - special addresses for broadcast and multicast

IP (Layer 3)

- Connectionless transmission of packages (datagrams)
- end-to-end delivery
- best-effort (unreliable)
- fragmentation of datagrams depending on the underlying layer 2 network
- Problems: packet loss, duplicates, order can be wrong

UDP (Layer 4)

- Connectionless transmission of packages
  - user has to segment the message into packages
- unreliable
- Problems: packet loss, duplicates, order can be wrong
- allows multiplexing
- allows multicast and broadcast

TCP (Layer 4)

Connection oriented

offers a reliable channel for an unstructured stream of bytes

- not visible what data items are transported in which package
- no lost packages, no duplicates, preserves order

allows multiplexing

no broadcast or multicast

sliding window protocol used for error handling and flow control

The TCP/ UDP API: Sockets

Sockets form the Internet-API on top of the transport layer

A socket is the endpoint of a communication

There exist two types of sockets

- one socket that behaves like the end of a telephone link
- one socket that behaves like a mailbox

Sockets are available in almost every programming language. We will consider sockets in Java

Ports

A socket is identified by a port-number and an IP-address

A port identifies a communication endpoint

A port identifies a process on a certain machine

- a port encapsulates the internal process structure
- a process can have multiple ports

A port is a 16-bit number

- numbers 0 –1023 are reserved for standard services

UDP / Datagram Sockets

Is based on the UDP protocol

UDP sockets allow to send an array of bytes with a packet

A message sent via a UDP socket will not be acknowledged

Errors can result in packet loss

- application has to be aware of this fact

Java API for UDP-Sockets

Two important classes:

- `DatagramPacket`
- `DatagramSocket`

`DatagramPacket` represents the packet and contains the information to be sent/received as well as address information

`DatagramSocket` handles the data transmission with the following methods

- `send(DatagramPacket)`
- `receive(DatagramPacket)`

UDP Client

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

public class UDPClient{
  public static void main( String args[]){

    try {

      System.out.println(" Message:  " + args[0]);

      DatagramSocket aSocket = new DatagramSocket();
      byte [] m = args[0].getBytes();
      InetAddress aHost = InetAddress.getByName(args[1]);
      int serverPort = 6789;
      DatagramPacket request = new DatagramPacket (m, m.length, aHost, serverPort);
      aSocket.send (request);

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


UDP Server

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

public class UDPServer{
  public static void main( String args[]){
    System.out.println("The server is up");
    try{
      DatagramSocket aSocket = new DatagramSocket (6789);
      byte[] buffer = new byte[1000];
      while(true){
        DatagramPacket request = new DatagramPacket (buffer, buffer.length);
        aSocket.receive (request);
       System.out.println(" Request: " + new String(request.getData(), 0,
                                                    request.getLength()));

        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());}
  }
}
```

TCP / Stream Sockets

Based on TCP protocol

TCP sockets allow a stream-oriented communication

Packet errors are transparently handled by TCP

Data is delivered

- exactly once
- in the right order

Before application data can be exchanged, a connection has to be established

Java API for TCP-Sockets

Two important classes:

- `ServerSocket`
- `Socket`

A `ServerSocket` is passively waiting for a client requesting to connect

- a connection results in the creation of `Socket`-object

The `Socket`-class is used by clients as well as servers

- a client uses a constructor of `Socket` to establish a connection with the server

A `Socket`-object maintains an input- and an output-stream

- can be used for sending and receiving data
- fits to the stream model of Java

TCP Client

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

public class TCPClient {
  public static void main (String args[]) {
  // args[0]: Message
  // args[1]: Server

    try{
                int serverPort = 7896;
                Socket s = new Socket (args[1], serverPort);
                DataOutputStream out = new DataOutputStream ( s.getOutputStream());
                DataInputStream in = new DataInputStream ( s.getInputStream());
                out.writeUTF (args[0]);
                String data = in.readUTF ();
                System. out.println("Reply: "+ data) ;
                s.close();
    }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());}
  }
}
```


TCP Server (1)

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

public class TCPServer {
  public static void main (String args[]) {
    try{
      System.out.println("The server is up");
                int serverPort = 7896;
                ServerSocket listenSocket = new ServerSocket (serverPort);
                while(true) {
                  Socket clientSocket = listenSocket.accept();
                  System. out.println("Neue Verbindung");
                  Connection c = new Connection(clientSocket);
                }
    } catch( IOException e) {System.out.println(" Listen :"+ e.getMessage());}
  }
}
```

TCP Server (2)

```
class Connection extends Thread {
  DataInputStream in;
  DataOutputStream out;
  Socket clientSocket;

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

  public void run(){
    try {
      String data = in.readUTF ();
                out.writeUTF(data);

                System.out.println("Echo: " + data);
} catch( EOFException e) {System.out.println(" EOF:"+ e.getMessage());
    } catch( IOException e) {System.out.println(" IO:"+ e.getMessage());}
  }
}
```


Assessment of Sockets

Sockets provide no transparency for distribution

Sockets provide only basic mechanisms

Powerful communication models like remote procedure calls (RPC) require a homogeneous data representation in heterogeneous environments

- standardized message format
- standardized coding of parameters: marshalling/serialization
  - Java Object Serialization
  - CDR (CORBA common data representation)
  - ASN.1 (ISO OSI)

In case of distributed object systems mechanisms to globally identify objects are also required

This is provided by more complex middleware like

- Java RMI
- CORBA