Ad hoc network programming

Nov 1st, 2011
Netzprogrammierung
(Algorithmen und Programmierung V)
Our topics last week

Descriptive models for distributed system design

<table>
<thead>
<tr>
<th>Physical model</th>
<th>Architectural model</th>
<th>Interaction model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architectural elements</td>
<td>Communication paradigm</td>
<td>Interaction model</td>
</tr>
<tr>
<td>Communicating entities</td>
<td>Roles and responsibilities</td>
<td>Placement</td>
</tr>
<tr>
<td>Processes</td>
<td>Inter-process communication</td>
<td>Multiple server</td>
</tr>
<tr>
<td>Objects</td>
<td>UDP sockets</td>
<td>Proxy/Cache</td>
</tr>
<tr>
<td>Components</td>
<td>TCP sockets</td>
<td>Mobile code</td>
</tr>
<tr>
<td>Web Services</td>
<td>Multicast</td>
<td></td>
</tr>
<tr>
<td>Architectural patterns</td>
<td>Architectural styles</td>
<td></td>
</tr>
<tr>
<td>Vertical distribution</td>
<td>Client-server</td>
<td></td>
</tr>
<tr>
<td>Multi-tier</td>
<td>Peer-to-peer</td>
<td></td>
</tr>
<tr>
<td>Horizontal distribution</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Claudia Müller-Birn, Netzprogrammierung 2011/12
Our topics today

- Internet Protocols, esp. TCP/IP layer
- API for Internet protocols, esp. sockets vs. ports
- UDP datagram communication
- TCP stream communication
- External data representation
- Multicast communication
Ad hoc network programming

Internet protocols
Protocols

Protocol refers to a set of rules and formats to be used for communication between processes in order to perform a given task.

- Specification of the sequence of messages that must be exchanged.
- Specification of the format of the data in the messages.
Conceptual layering of protocol software

Sender

Communication medium

Recipient

Layer n

Layer 2

Layer 1

Message sent

Message received
Review: Protocol layers in the ISO Open Systems Interconnection (OSI) model

Application
Presentation
Session
Transport
Network
Data link
Physical

Message sent
Communication medium
Message received

Sender
Recipient
Encapsulation as it is applied in layered protocols
TCP/IP layer

Layers

Application

Messages (UDP) or Streams (TCP)

Transport

UDP or TCP packets

Internet

IP datagrams

Network interface

Network-specific frames

Underlying network
IPv4 addressing

Objective: schemes for naming and addressing hosts and for routing IP packets to their destinations.

Defined scheme assigns an IP address to each host in the Internet
- Network identifier – uniquely identifies the sub-network in the internet
- Host identifier - uniquely identifies the host’s connection

32-bit, written in a 4 Bytes in decimal notation, e.g. 130.149.27.12

<table>
<thead>
<tr>
<th>Class</th>
<th>Network ID</th>
<th>Host ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class A</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>Class B</td>
<td>1 0</td>
<td>14 16</td>
</tr>
<tr>
<td>Class C</td>
<td>1 1 0</td>
<td>21 8</td>
</tr>
<tr>
<td>Class D (multicast)</td>
<td>1 1 1 0</td>
<td>Multicast address</td>
</tr>
<tr>
<td>Class E (reserved)</td>
<td>1 1 1 1 0</td>
<td>unused</td>
</tr>
</tbody>
</table>
Java API: package java.net

Java provides class InetAddress that represents Internet addresses

- Method static InetAddress getByName(String host)
- Can throw an UnknownHostException

- Example

```java
w3c = InetAddress.getByName("www.w3c.org");
me = InetAddress.getByName("localhost");

System.out.println(InetAddress.getByName("localhost"));
// localhost/127.0.0.1
System.out.println(InetAddress.getLocalHost());
// lounge.mi.fu-berlin.de/160.45.42.83
```

http://download.oracle.com/javase/6/docs/api/java/net/InetAddress.html
API for Internet protocols
Middleware layers

Applications, services

Remote invocation, indirect communication

Underlying inter-process communication primitives: Sockets, message passing, multicast support

UDP and TCP
Characteristics of inter-process communication

**Synchronous communication:** sending and receiving processes synchronize at every message = send and receive are blocking operation

**Asynchronous communication:** send and receive operations are non-blocking
Sockets

Interprocess communication consists of transmitting a message between a message between a socket in one process and a socket in another process.
Socket address = IP address and port number

Sockets

- Sockets provide an interface for programming networks at the transport layer.
- Network communication using Sockets is very much similar to performing file I/O.
- Socket-based communication is programming language independent.

Ports

- Port is represented by a positive (16-bit) integer value.
- Some ports have been reserved to support common/well known services such as ftp (20 for data and 21 control).
- User level process/services generally use port number value >= 1024.
Realizing process-to-process communication

**UDP features**
- UDP datagram encapsulated inside an IP package
- Header includes source and destination port numbers
- No guarantee of delivery
- Message size is limited
- Restricted to applications and services that do not require reliable delivery of single or multiple messages

**TCP features**
- Provides reliable delivery of arbitrarily long sequences of bytes via stream-based programming abstraction
- Connection-oriented service
- Before data is transferred, a bidirectional communication channel is established
UDP datagram communication
UDP Sockets

1. Client creates socket bound to a local port

2. Server binds its socket to a server port

3. Client/Server send and receive datagrams

4. Ports and sockets are closed
Issues related to datagram communication

Message size
• Receiving process needs to specify an array of bytes of a particular size in which to receive a message
• If the received message is too big it is truncated

Datagram communication is carried out with a non-blocking `send` and a blocking `receive` operation

Timeouts can be set, in order to avoid that the receive operation waits indefinitely

Receive method does not specify an origin of the messages. But it is possible to connect a datagram socket to a particular remote port and Internet address.
Failure model of UDP datagrams

**Integrity**

- Messages should not be corrupted or duplicated
- Use of checksum reduces probability that received message is corrupted

**Failures**

- Omission failures: messages maybe dropped occasionally because of checksum error or no buffer space is available at source/destination
- Ordering: Messages can sometimes be delivered out of order
Using UDP for applications

Advantage of UDP datagrams is that they do not suffer from overheads associated with guaranteed message delivery

Example 1: Domain Name System
- DNS primarily uses UDP on port number 53 to serve requests
- DNS queries consist of a single UDP request from the client followed by a single UDP reply from the server

Example 2: VOIP
- No reason to re-transmit packets with bad speech data
- Speech data must be processed at the same rate as it is sent - there is no time to retransmit packets with errors
UDP datagram communication

Java API for UDP diagrams
Java API for UDP diagrams

Datagram communication is provided by two classes, `DatagramPacket` and `DatagramSocket`.

**DatagramPacket**
- Constructor that makes an instance out of an array of bytes comprising a message.
- Constructor for use when receiving a message, message can be retrieved by the method `getData`.

**DatagramSocket**
- Constructor that takes port number as argument for use by processes.
- No-argument constructor for choosing a free local port.
Example: Java client (UDP)

```java
import java.io.*;
import java.net.*;
class UDPClient {
    public static void main(String args []) throws Exception {
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));

        DatagramSocket clientSocket = new DatagramSocket();

        InetAddress IPAddress =
            InetAddress.getByName("hostname");

        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];
        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes();
    }
}
```

Claudia Müller-Birn, Netzprogrammierung 2011/12
Example: Java client (UDP) (cont.)

Create datagram with data-to-send, length, IP addr, port

```
DatagramPacket sendPacket =
new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
```

Send datagram to server

```
clientSocket.send(sendPacket);
```

Read datagram from server

```
DatagramPacket receivePacket =
new DatagramPacket(receiveData, receiveData.length);

clientSocket.receive(receivePacket);

String modifiedSentence = new String(receivePacket.getData());

System.out.println("FROM SERVER:" + modifiedSentence);

clientSocket.close();
```
Example: Java server (UDP)

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

class UDPServer {
    public static void main(String args []) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];
        while(true) {
            DatagramPacket receivePacket =
                new DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket);
        }
    }
}
```

Claudia Müller-Birn, Netzprogrammierung 2011/12
Example: Java server (UDP) (cont.)

```java
String sentence = new String(receivePacket.getData());

InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();

DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, port);

serverSocket.send(sendPacket);
```

Get IP addr port #, of sender

Create datagram to send to client

Write out datagram to socket

End of while loop, loop back and wait for another datagram
TCP stream communication
Hiding network characteristics by TCP

Application can choose the **message size**, means how much data it writes to a stream or reads from it.

TCP protocol uses an **acknowledgement** scheme to avoid lost data.

TCP supports **flow control** that means if the writer is too fast for the reader, then the writer is blocked until the reader consumed sufficient data.

Message identifiers are used by each IP packet. The recipient can therefore detect and reject **duplicates** or can **reorder** message if needed.

Before a pair of communication processes communicate they **establish a connection**.
TCP Sockets

1. Server bind port

2. Server is ready and listening

3. Server is waiting for request, client sends request, server accepts

4. Client and server are connected - bidirectional!

5. Connection is closed
Failure model of TCP

In order to realize reliable communication, TCP streams use checksums to detect and reject corrupt packages and sequence numbers to detect and reject duplicate packets.

To deal with lost packages TCP streams use timeouts and retransmissions.

A broken connection has the following effects
• The processes using the connection cannot distinguish between network failure and failure of the process at the other end of the connection
• The communication processes cannot tell whether the messages they have sent recently have been received or not.
Use of TCP

Many frequently used services run over TCP connections with reserved port numbers

- **HTTP** [RFC 2068]: The Hypertext Transfer Protocol is used for communication between web browser and web server.
- **FTP** [RFC 959]: The File Transfer Protocol allows directories on a remote computer to be browsed and files to be transferred from one computer to another over a connection.
- **Telnet** [RFC 854]: Telnet provides access by means of a terminal session to a remote computer.
- **SMTP** [RFC 821]: The Simple Mail Transfer Protocol is used to send mail between computer.

The HTTP protocol

Suppose you enter the URL: http://www.inf.fu-berlin.de/groups/index.html

1a) http client initiates TCP connection to http server (process) at www.inf.fu-berlin.de; Port 80 is default for http server

1b) http server at host www.inf.fu-berlin.de waiting for TCP connection at port 80; “accepts” connection, notifying client

2) http client sends http request message (containing URL) into TCP connection socket

3) http server receives request message, forms response message containing requested object (someDepartment/home.index), sends message into socket
The HTTP protocol (cont.)

4) http server closes TCP connection

5) http client receives response message containing html file, displays html. Parsing html file, finds 2 referenced jpeg objects

6) steps 1-5 repeated for each jpeg objects
TCP stream communication

Java API for TCP
Java API for TCP streams

Java interface provides two classes ServerSocket and Socket

ServerSocket
- Class is intended to be used by server to create a socket at a server port for listening for connect requests from clients.

Socket
- Class is for use by a pair of processes with a connection
- The client uses a constructor to create a socket, specifying the DNS hostname and port of a server
Example: Java client (TCP)

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

class TCPClient {

    public static void main(String argv[]) throws Exception {

        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);
    }
}
```
Example: Java client (TCP) (cont.)

Create output stream attached to socket

```java
DataOutputStream outToServer = 
    new DataOutputStream(clientSocket.getOutputStream());
```

Create input stream attached to socket

```java
BufferedReader inFromServer = 
    new BufferedReader(new InputStreamReader(
        clientSocket.getInputStream()));
```

Send line to server

```java
sentence = inFromUser.readLine();
outToServer.writeBytes(sentence + '\n');
```

Read line from server

```java
modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);
```

```java
clientSocket.close();
```
Example: Java server (TCP)

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

class TCPServer {
    public static void main(String argv []) throws Exception {
        String clientSentence;
        String capitalizedSentence;

        ServerSocket welcomeSocket = new ServerSocket(6789);
        while(true) {
            Socket connectionSocket = welcomeSocket.accept();
            BufferedReader inFromClient = new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));
        }
    }
}
```

Create welcoming Socket at port 6789
Wait, on welcoming Socket for contact by client
Create input stream, attached to socket
DataOutputStream outToClient = new DataOutputStream (connectionSocket.getOutputStream());

clientSentence = inFromClient.readLine();
capitalizedSentence = clientSentence.toUpperCase() + '\n';

outToClient.writeBytes (capitalizedSentence);

End of while loop, loop back and wait for another client connection
External data representation and marshalling
What is the challenge?

Messages consist of sequences of bytes.

Interoperability Problems

• Big-endian, little-endian byte ordering
• Floating point representation
• Character encodings (ASCII, UTF-8, Unicode, EBCDIC)

So, we must either:

• Have both sides agree on an external representation or
• transmit in the sender’s format along with an indication of the format used. The receiver converts to its form.
External Data Representation and Marshalling

*External data representation*
An agreed standard for the representation of data structures and primitive values

*Marshalling*
The process of taking a collection of data items and assembling them into a form suitable for transmission in a message

*Unmarshalling*
Is the process of disassembling them on arrival into an equivalent representation at the destination

Claudia Müller-Birn, Netzprogrammierung 2011/12
Approaches for external data representation

CORBA’s common data representation
• Concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote invocation in CORBA.

Java’s object serialization
• Refers to the activity of flattening an object or even a connected set of objects that need to be transmitted or stored on a disk

XML
• Defines a textual format for representing structured data
Google Protocol Buffer

Protocol Buffer (PB) is a common serialization format for Google

Google adopts a minimal and efficient remote invocation service

The goal of Protocol Buffer is to provide a language- and platform-neutral way to specify and serialize data such that:

• Serialization process is efficient, extensible and simple to use
• Serialized data can be stored or transmitted over the network

More information here:
http://code.google.com/apis/protocolbuffers/docs/overview.html
Comparison of Protocol Buffer Language

Advantages of Protocol Buffer (PB)
- PB is 3-10 times smaller than an XML
- PB is 10-100 times faster than an XML

Can we compare PB with XML?
- PB works only on Google infrastructure, which is relatively closed system and does not address inter-operability
- XML is richer (it specifies self-describing data and meta-data). PB is not so rich. There are accessory programs that can create a full description. However, they are hardly used.
Multicast communication
Possibilities to communicate

Connection-oriented 1:1
TCP

Connectionless 1:1
UDP

Connectionless 1:n
Multicast
Multicast messages

Multicast messages provide a useful infrastructure for constructing distributed systems with the following characteristics:

- Fault tolerance based on replicated services
- Discovering services in spontaneous networking
- Better performance through replicated data
- Propagation of event notifications
Multicast Sockets

1. Participants bind socket

2. Participants join group

3. Participants receive messages from sender

4. Participants leave group and release socket
IP Multicast

Is built on top of the Internet Protocol (IP) and allow the sender to transmit a single IP packet to a set of computers that form a multicast group.

Multicast group is specified by a Class D Internet Address. Every IP datagram whose destination address starts with "1110" is an IP Multicast datagram.

IP packets can be multicast on a local and wider network. In order to limit the distance of operation, the sender can specify the number of routers that can be passed (i.e. time to live, or TTL)

Multicast addresses can be permanent (e.g. 224.0.1.1 is reserved for the Network Time Protocol (NTP))
Java API: java.net.MulticastSocket

```java
public class MulticastSocket extends DatagramSocket {
    public MulticastSocket()...
    public MulticastSocket(int port)...
        // create socket and select port number explicitely or implicitely
    public void setTimeToLive(int ttl) ...
        // define Time to Live – default is 1 !
    public void joinGroup(InetAddress mcastaddr) throws ...
        // join group under the address mcastaddr
    public void leaveGroup(InetAddress mcastaddr) throws ...
        // leave group
}
```

Please note: send, receive, ... are inherited from class DatagramSocket
Our topics last week

Descriptive models for distributed system design

Physical model

Architectural model

Architectural elements

Communicating entities
- Processes
- Objects
- Components
- Web Services

Communication paradigm
- Inter-process communication
- Indirect communication
- Remote invocation

Roles and responsibilities
- Architectural styles
  - Client-server
  - Peer-to-peer
- Placement
  - Multiple server
  - Proxy/Cache
  - Mobile code

Architectural patterns
- Vertical distribution
  - Multi-tier
  - Thin/Fat Client
- Horizontal distribution

Interaction model

Interaction model

Failure model

Security model
Summary

• TCP/IP layer
• Characteristics of inter-process communication
• Sockets vs. ports
• UDP datagram communication
  • Characteristics, failure model, usage
  • Java API for UDP diagrams
• TCP stream communication
  • Characteristics, failure model, usage
  • Java API for TCP streams
• Approaches for external data representation (marshalling)
• Multicast communication
Next class

Structured communication (RCP)
References

Main resource for this lecture: