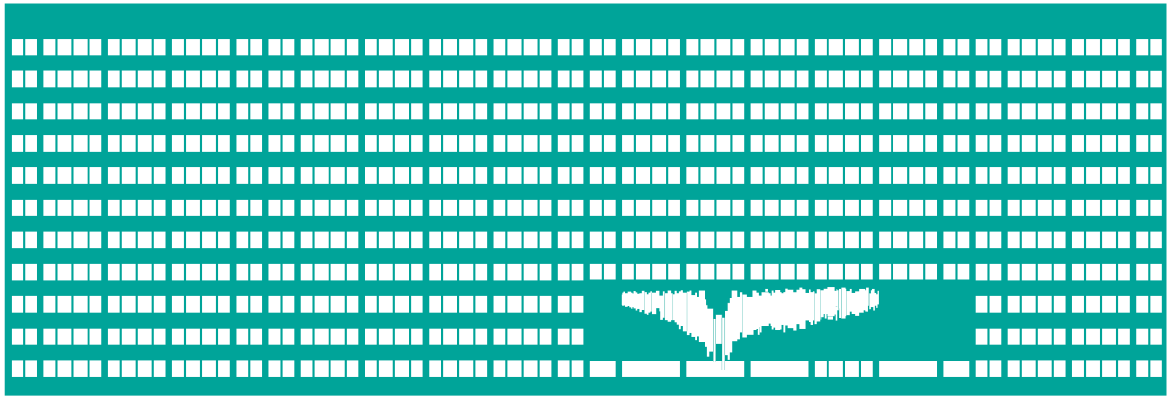
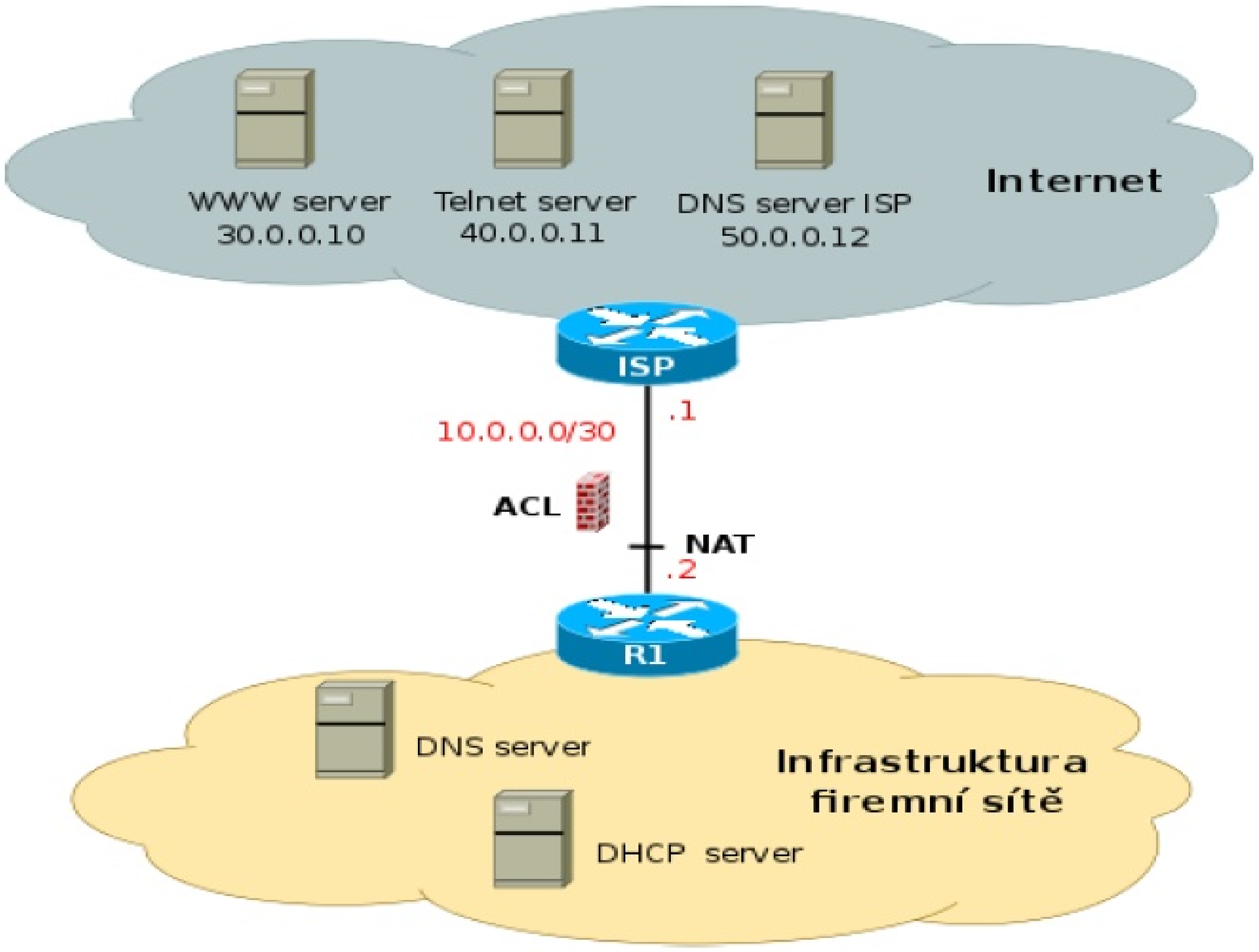
VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Sockets Interface
Java (C)



Computer networks
Seminar 3

Semestral project (1)



Semestral project (2)

Project parts:

- Address plan and VLAN configuration
- Routing and NAT
- DNS server
- DHCP server
- Securing the network – ACL

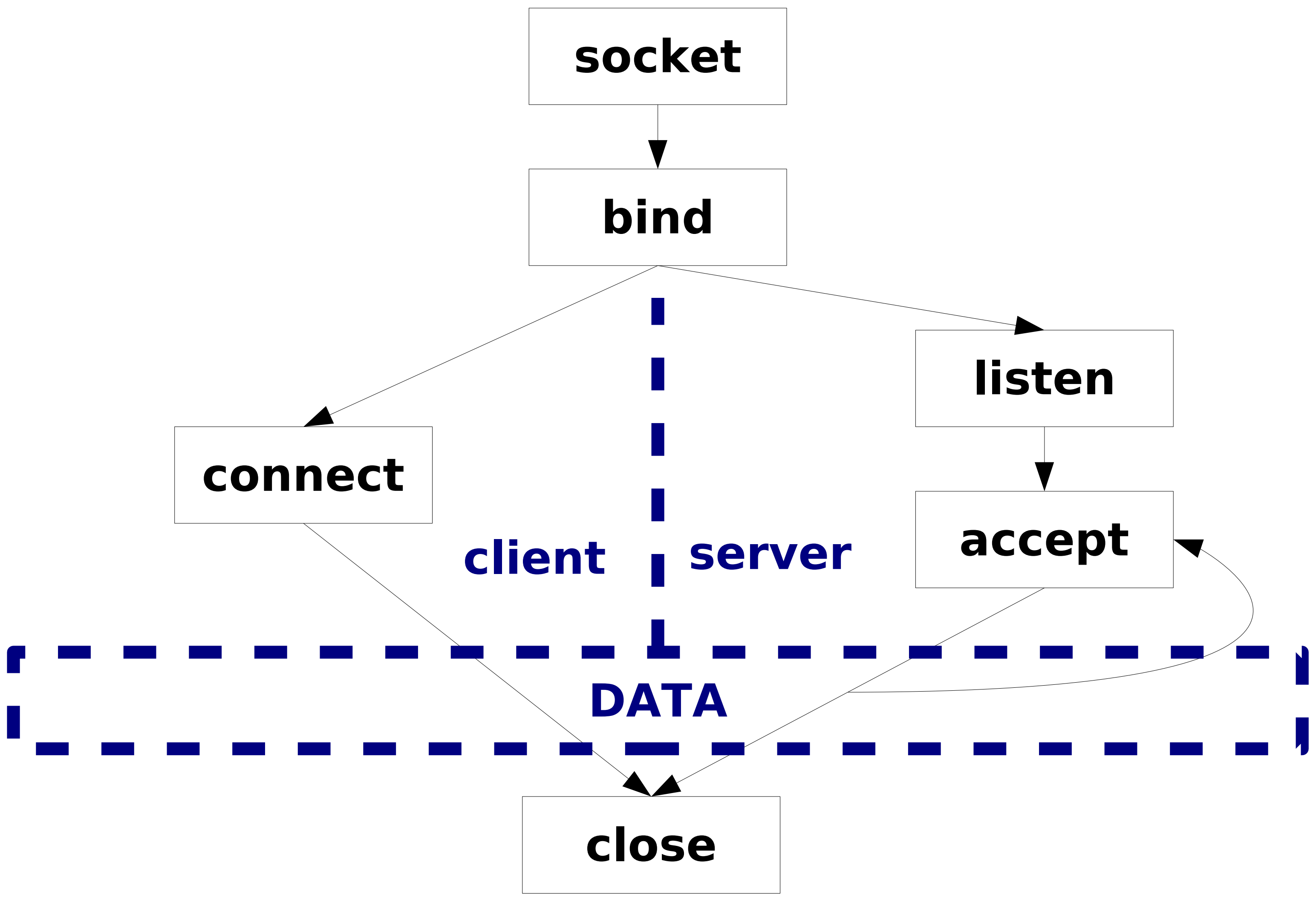
- The content of each project part is exactly specified

Introduction

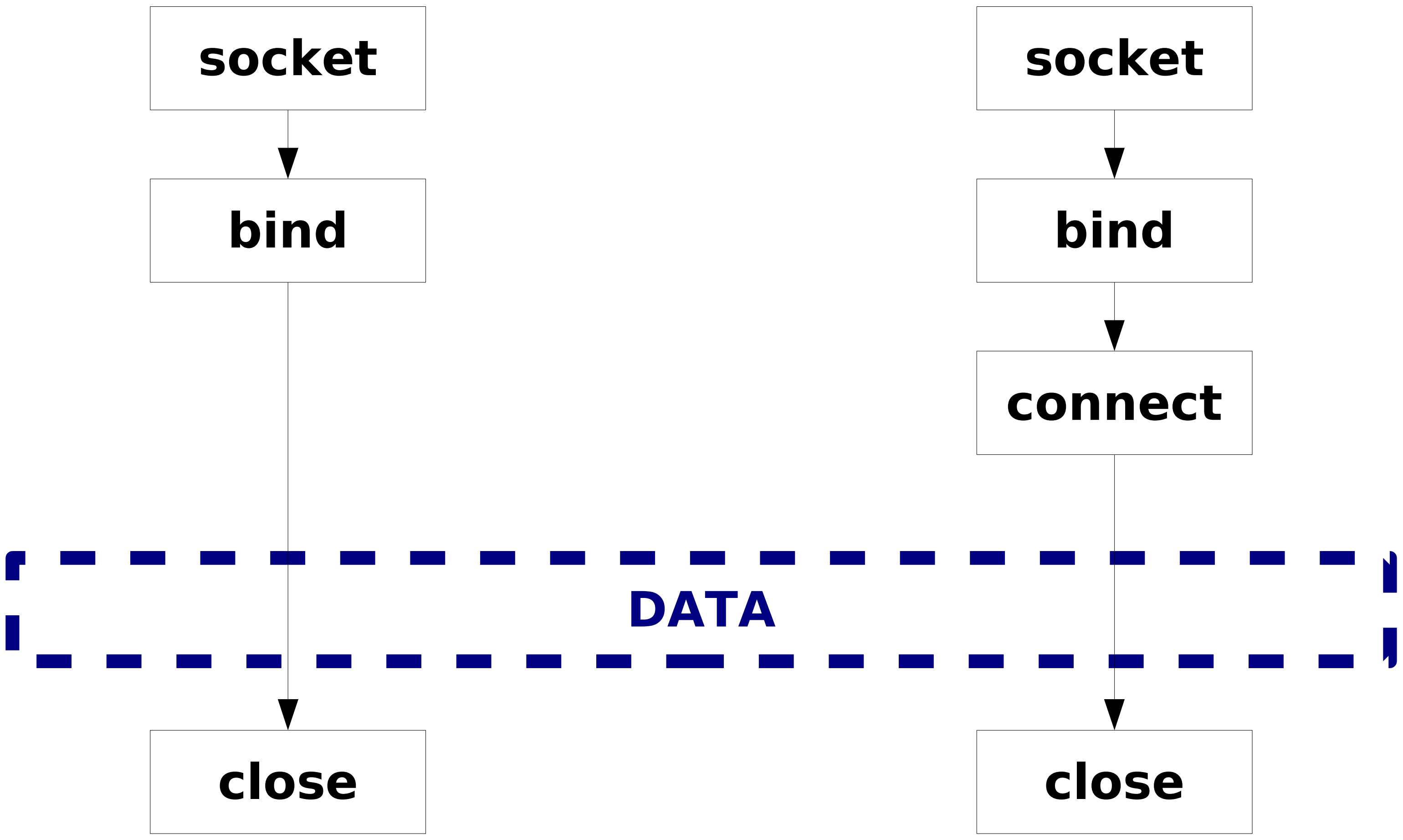
- **BSD Sockets** – working with network connection similar as with file (POSIX)
- **IP** protocol family – **TCP** and **UDP**, identification based on IP address and port number

    - **TCP** (Transmission Control Protocol) – reliable logical channel. The connection is established before communication, all received data are acknowledged, it is necessary to close the connection in the end
  - **UDP** (User Datagram Protocol) – no connection establishment, data are sent to the given IP address and UDP port and we don't know if data had been delivered, if they hadn't been duplicated... (in the case of checksum missing we don't know if data arrived allright).

Working with TCP sockets



Working with UDP sockets



Necessary libraries

- In Java programming language
  - java.net – Sockets
  - java.io – Exceptions, streams

Creating the socket

- `new Socket([adresa, c_port])`
- `new ServerSocket(z_port [, bl])`
- `new DatagramSocket([z_port])` – UDP
- `new MulticastSocket([z_port])` – UDP multicast

Binding

- Binding to the specified port
  - `public void` **`bind`**`(SocketAddress bindpoint)`

Client connection

- The IP address can be defined in Socket constructor (TCP), or
  - `void` **`connect`**`(SocketAddress endpoint)`
  - In the case of using UDP you can use the class **DatagramPacket** and methods **send** a **receive** of the class **DatagramSocket**

Running the server

- There are functions **bind** (matches **bind+listen**) and **accept** in class **ServerSocket**
  - We specify port number and maximum of requests waiting in the queue (blacklog) in constructor or `void` **`bind`**`(SocketAddress endpoint, int backlog)`
  - Function `void` **`accept`**`()` - opens incoming socket

Sending data

- We use reader/writer created over the streams obtained from **getInputStream()**, **getOutputStream()** or class **DatagramPacket**

Closing the connection

- To close the connection call the function
  - `void close()`
    - On socket or on Input/Output stream

Socket parameters

- To set the parameters use the functions **get/set** included in the classes *Socket (for example setSoTimeout)

Further details

- Java doesn't support “non-blocking” sockets, but it is possible to call the function
  - **`setSoTimeout`**`(milliseconds)`
    - If blocking operation (reading, writing) is not finished on time → exception **java.net.SocketTimeoutException**
      - Socket is not destroyed, new try of executing the operation can follow after threating the excpetion
- Domain name to InetAddress conversion
  - `InetAddress ia=InetAddress.`**`getByName`**`(name)`
    - `ia.getHostAddress()` – returns IP address as a string

Alternative progr. languages

- (BSD) Winsock tutorial for C/C++ in Windows

https://docs.microsoft.com/en-us/windows/win32/winsock/getting-started-with-winsock

- Python
  - Client sockets: https://docs.python.org/3/library/socket.html
  - Server: https://docs.python.org/3/library/socketserver.html

- C# (.NET)
  - General description:
    https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.socket
  - (Synchronnous) TCP client – 1st part of assignment:
    https://docs.microsoft.com/en-us/dotnet/framework/network-programming/synchronous-client-socket-example
  - UDP receiver – 2nd part of the assignment
    https://docs.microsoft.com/en-us/dotnet/framework/network-programming/using-udp-services

Examples

- Detailed description and examples
  - C/C++
    - http://www.cs.vsb.cz/grygarek/PS/sockets.html
  - Java
    - http://www.cs.vsb.cz/grygarek/PS/dosys/inprise/net/examples
      - Ex1 – TCP
      - Ex2 – UDP
      - Ex3 – multicast
      - Ex4 – getting data by URL(class URLConnection)

Java – TCP Socket connection

```
try {
  s=new Socket(host,port);
  BufferedReader sis = new BufferedReader(new InputStreamReader(System.in));
  BufferedWriter os = new BufferedWriter(
                                    new OutputStreamWriter(s.getOutputStream()));
  BufferedReader is = new BufferedReader(
                                    new InputStreamReader(s.getInputStream()));
  String l;
  do {
    System.out.println("Type a line to send to server.");
    l=sis.readLine();
    os.write(l);
    os.newLine();
    os.flush();
    System.out.println("Server: " + is.readLine());
  } while (!l.equals("exit") && !l.equals("down"));
  s.close();
} catch (IOException e) { System.out.println(e); }
```

Java – TCP Server

```
…
try {
 ServerSocket s=new ServerSocket(port);
 Socket cs;
 do {
   cs=s.accept();
   BufferedReader is = new BufferedReader
                        (new InputStreamReader(cs.getInputStream()));
   BufferedWriter os = new BufferedWriter
                        (new OutputStreamWriter(cs.getOutputStream()));
   do {
    msg=is.readLine();
    os.write(String.valueOf(msg.length()));
    os.newLine(); os.flush();
   }  while (!msg.equals("exit") && !msg.equals("down"));
   cs.close();
 } while (!msg.equals("down"));
 s.close();
} catch (IOException e) { System.out.println(e); }
```

Java – MultiThreading TCP Server

Imporving previous example using threads, it is possible to process more requests at the same time

```
public class MyApplication implements Runnable {
      protected Socket cs;   static ServerSocket s;
      static int port=8000;
      public static void main(String[] args) {
        Socket cs;
        try {
            s=new ServerSocket(port);
            do {cs=s.accept(); new Thread(new MyApplication(cs)).start();
            } while (true);
        } catch (IOException e) {if(!e instanceof SocketException){System.out.println(e);}}
      }
      public MyApplication(Socket cs) {this.cs=cs;}
   public void run() {
     /* code from previous slide */
      if (msg.equals("down")) s.close(); // Close the socket and terminate the app.
   }
}
```

Java – UDP Client

```
String data;  //we will get data later, for example from System.in
int port=8000;
String server="www.cs.vsb.cz";
…
try {
  DatagramSocket s=new DatagramSocket();
  DatagramPacket p = new DatagramPacket(data.getBytes(), data.length(),
                                        InetAddress.getByName(server), port);
   s.send(p);
   s.receive(p);
   reply=new String(p.getData(),0,p.getLength());
   System.out.println("Reply arrived from "+ p.getAddress() +" : "+ p.getPort()+
                                                              " > " + reply);
   s.close();
} catch (IOException e) { System.out.println(e); }
```

Java – UDP Server

There is no dedicated class for datagram servers, thee is no need for it

```
…
try {
DatagramSocket s=new DatagramSocket(port);
DatagramPacket p;
String msg;
do {
  p=new DatagramPacket(new byte[512], 512);
  // ^ regarding fixed buffer size we allocate anew
  s.receive(p);
  msg = new String(p.getData(),0,p.getLength());
  System.out.println("D. from " + p.getAddress() + " : " + p.getPort() + " > " + msg);
  p.setData(msg.toUpperCase().getBytes());
  p.setLength(msg.length());
  s.send(p);
} while (!msg.equals("down"));
s.close();
} catch (IOException e) { System.out.println(e); }
```

Java - DataInputStream

- ByteArrayInputStream baStream = new ByteArrayInputStream(b);
- DataInputStream dis = new DataInputStream(baStream);
    - DataInputStream: readByte()
    - DataInputStream: readInt()
    - DataInputStream: read(byte[] b, int off, int len)
- InetAddress: getByAddress(String host, byte[] addr)

Sockets in C/C++

- In C/C++ following headers are to be included in Unix-based systems:
  - netdb.h
  - arpa/inet.h
  - sys/time.h
  - sys/socket.h
  - netinet/in.h
- On Windows Winsock in C/C++ we do:
  - #include <winsock2.h>
  - #include <ws2tcpip.h>
  - #include <stdio.h>

  - #pragma comment(lib, "Ws2_32.lib")

Creating the sockets

- `int socket(int domain, int type, int protocol)`
  - **domain** – type of communication (**PF_INET**: IP)
  - **type** – type of socket
    - **SOCK_STREAM**: TCP
    - **SOCK_DGRAM**: UDP
  - **protocol** – not used in our case, so it is set to 0.

Binding

- `int` **`bind`**`(int sck, struct sockaddr* name, int namelen)`
  - **sck** – socket descriptor (from socket function)
  - **name** – sockaddr_in structure with socket IP address and port number
    - **sin_family** – **AF_INET** – protocol IPv4
    - **sin_addr.s_addr** – IP address (**INADDR_ANY**)
    - **sin_port** – local port
  - **namelen** – structure size: `sizeof(sockaddr_in))`

Client connection

- `int` **`connect`**`(int sck, struct sockaddr* name, int namelen)`
  - **name** – like bind(), but it is IP address of target.
  - In the case of working with UDP it is not necessary to use **connet** function if you use functions **sendto** and **recvfrom**.

Running the server

- `To listen on the socket`
  - `int` **`listen`**`(int sck, int backlog)`
    - **backlog** – max. number of requests waiting in queue
  - `int` **`accept`** `(int sck, struct sockaddr* addr, int* addrlen)`
    - Block itself until receiving the request, client IP address is saved into addr (Accept is usually followed by int **fork()**)

Sending data

- C/C++ offers several function for sending data
  - POSIX file functions **read/write**
    - `int` **`read`**`(int sck, char* buf, unsigned buflen)`
    - `int` **`write`**`(int sck, char* buf, unsigned buflen)`
  - Functions send/recv
    - `int` **`send`**`(int sck, char* buf,int buflen, int flags)`
    - `int` **`recv`**`(int sck, char* buf,int buflen, int flags)`
      - **flags** can further specify data (urgent...)
  - Functions **sendto/recvfrom** for datagrams without connect
    - `int` **`sendto`**`(<jako send>, struct sockaddr* to, int tolen)`
    - `int` **`recvfrom`**`(<jako recv>, struct sockaddr* from, int *fromlen)`

Closing connection

- To close the connection
  - `int` **`close`**`(int sck)`
    - typical one
  - `int` **`shutdown`**`(int sck, int how)`
    - how = 0 – to close receiving
    - how = 1 – to close transmitting
    - how = 2 – connection reset

Socket parameters

- Socket functions
  - `int` **`getsockopt`**`(int sck, int lvl, int optname, char* optval, int* optlen)`
  - `int` **`setsockopt`**`(int sck, int lvl, int optname, char* optval, int optlen)`
    - **lvl**: SOL_SOCKET, IPPROTO_TCP, ...
- Files functions (fcntl.h, unistd.h)
  - `int` **`fcntl`**`(int sck, int cmd, long arg)`
- I/O device functions (sys/ioctl.h)
  - `int` **`ioctl`**`(int d, int request, ...)`

Non-blocking sockets v C/C++

- If we can not wait for incoming data:

  -
    ```
    int flag=0;
    flag=fcntl(sck, F_GETFL, 0);
    if(flag == -1) { flag = 0; }
    fcntl(sck, F_SETFL, flag | O_NONBLOCK);
    ```
- Old systems don't have O_NONBLOCK:

  -
    ```
    int flag=1;
    ioctl(sck, FIOBIO, &flag);
    ```
- Blocking functions return -1 instead of being blocked
  - **errno** set to EWOULDBLOCK

Timeout solution in C

```
#include <unistd.h>
#include <signal.h>
#define MAX_REP 10                           //Max. number of retransmissions
int repeat=0;                                 //Actual number of retrans.
void alarmed(int signo) {
  if (++repeat > MAX_REP) {                   //MAX_REP exceeded
                          //inform about transmission failing
    signal(SIGALRM, NULL); exit(-1); //End
  } else {                                    //do retransmissions
    alarm(1);                                 //set timeout again
    }
  }
…                                                 //Send data (1. try)
  alarm(1); signal(SIGALRM,alarmed); //Set timeout
          // Receiving of acknowledge (if it is not received in
            1s, raise SIGALRM)
  repeat=0; alarm(0);                             //Cancel timeout
```

Next functions in C

- Format conversions
  - **`htonl`**, **`htons`** – endian PC→network (long, short)
  - **`ntohl`**, **`ntohs`** – endian network→PC (long, short)
  - `unsigned long` **`inet_addr`**`(char* cp)`
    - Conversion from dot notation
  - `char*` **`inet_ntoa`**`(struct in_addr in)`
    - Conversion to dot notation
- Working with domain names
  - `int` **`gethostname`**`(char* name, int namelen)`
    - Name of local station
  - `struct hostent*` **`gethostbyname`**`(char* name)`
    - Address of the station is in **`char * he->h_addr`**
  - `struct hostent*` **`gethostbyaddr`**`(char* addr, int len, int type)`
    - Domain name from IP address **`char* he->h_name`**

Assignment – collective chat

- Sending messages to “talk” server over **TCP/8000**
- Receiving messages on **UDP/8010** ~~from broadcast~~ unicast due to on-line lab version
- Message
  - max. 255 ASCII characters
  - ending with <LF> (i. e. \n)
  - Header (9 bytes) of the message which is sent back from server :
    - 4 byte – sender IP address (int, format lo-hi)
    - 4 byte - htonl(time_t)
    - 1 byte – length of the message (0-255).

      | 127 | 0 | 0 | 1 | 0x4A | 0xAE | 0x1A | 0x30 | 3 | SSS |