Lesson 8

Christian Schwarz, Jakob Krebs

15.12.2019

Contents

Network Protocols

Socket Programming

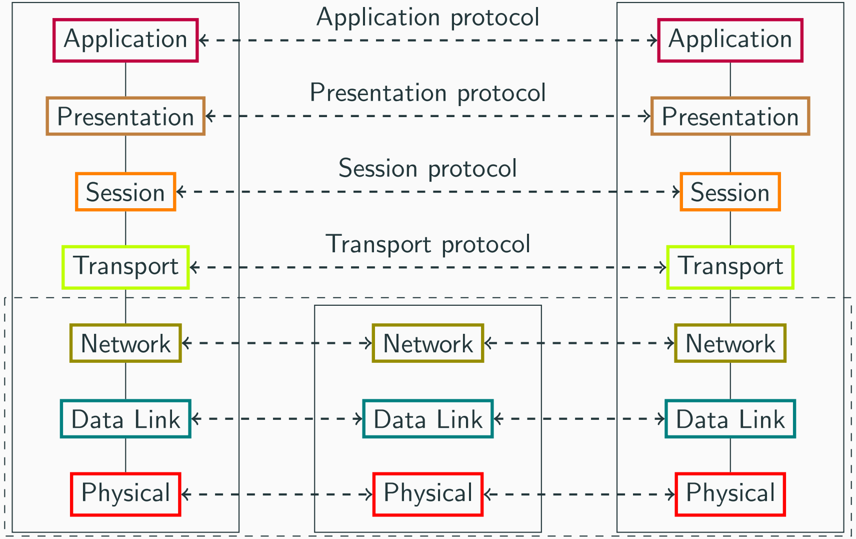
Task

Sources and Solutions

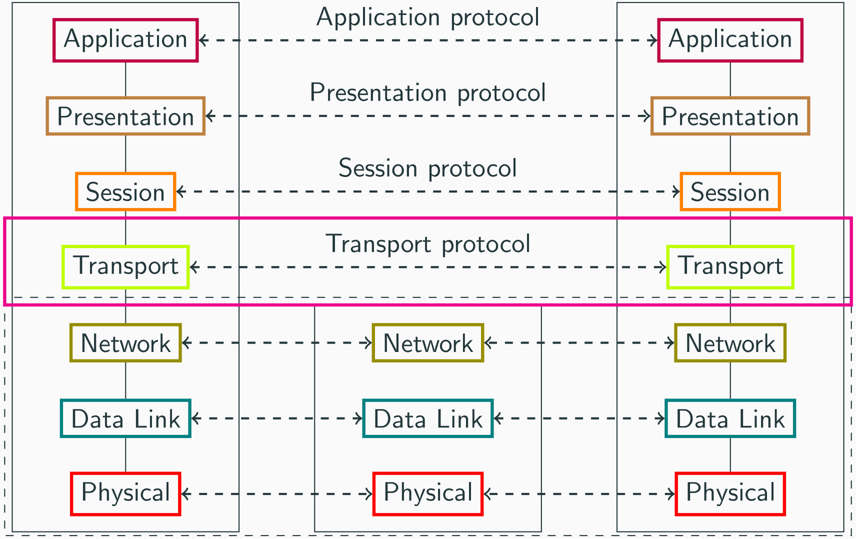
- we publish all code written in this course at `https://github.com/jkrbs/c_lessons`
- we will publish example solutions of the tasks on same site
- send us questions or your solutions to c-lessons@deutschland.gmbh

Network Protocols

Osi Reference Model



Osi Reference Model



Transport Protocols

TCP:

- connection-oriented
- three-way-handshake
- dialog between two sides
- guaranteed data delivery in the same order as sent

UDP:

- connectionless
- faster, since it is ”best effort” (no error recovery)
- no guarantee for sent packages to arrive

Socket Programming

Sockets

Sockets are abstractions for connection endpoints to be used by processes. Both the server and the client process have a socket which they use to send data to each other.

Sockets are platform-dependend, but the system call interface is similar:

Unix	file descriptors (int)
Windows	handles for kernel objects (**SOCKET**)

You will also have to include different headers:

```
// Unix
#include <sys/socket.h>
// Windows
#include <windows.h>
```

socket()

Create an endpoint for communication.

```
//Unix
int socket(int domain, int type, int protocol);
//windows
SOCKET socket(int domain, int type, int protocol);
```

domain	Communication domain for the socket `[AF_INET, AF_INET6, os-specific domains]`
type	Type of the socket `[SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, ...]`
protocol	The protocol to be used `[0, IPPROTO_TCP, IPPROTO_UDP, ...]`
return value	File descriptor / socket handle if successful, -1 otherwise

close[socket]()

Close an existing socket / file descriptor.

```
//Unix
int close(int fildes);
//windows
int closesocket(SOCKET socket);
```

fildes/socket File descriptor / handle of the socket to close

return value Exit status (0 = success, -1 = failure)

Do not leak file descriptors!

connect()

Connect a socket to another via the network.

```
// Unix
int connect(int socket, const struct sockaddr *address,
            socklen_t address_len);
// Windows
int connect(SOCKET socket, const struct sockaddr *address,
            int address_len);
```

socket	Socket to be connected
address	Structure containing target IP address and port
address_len	Size of `*address` in memory
return value	Exit status (0 = success, -1 = failure)

UDP sockets don't establish a connection → `connect()` is optional.

bind()

Bind an address to a socket.

```
// Unix
int bind(int socket, const struct sockaddr *address,
         socklen_t address_len);
// Windows
int bind(SOCKET socket, const struct sockaddr *address,
         int address_len);
```

socket	Socket to be bound
address	Structure containing IP address and port
address_len	Size of `*address` in memory
return value	Exit status (0 = success, -1 = failure)

Naming a socket is necessary for connections from the outside!

listen()

Enable listening for connections to a specific socket.

```
// Unix
int listen(int socket, int backlog);
// Windows
int listen(SOCKET socket, int backlog);
```

socket	Socket to put into listening mode
backlog	Hint for an upper bound of the number of outstanding connections in the listening queue of the socket
return value	Exit status (0 = success, -1 = failure)

Calling `listen()` on a socket is necessary to accept incoming TCP connections on a server.

accept()

Accept a new connection on a socket.

```
// Unix
int accept(int socket, struct sockaddr *restrict address,
           socklen_t *restrict address_len);
// Windows
SOCKET accept(SOCKET socket, struct sockaddr *address,
              int *address_len);
```

socket	Listening socket
address	Where to store the address of the connecting socket
address_len	Size of `*address` in memory
return value	Socket for the new connection on success, invalid descriptor otherwise

By default, `accept()` blocks if the socket's connection queue is empty!

send[to]()

Send a message on a socket.

```
// Unix
int send[to](int socket, const void *buffer, size_t length,
                                int flags[, const struct sockaddr *dest_addr,
                      socklen_t dest_len]);
// Windows
int send[to](SOCKET socket, const char *buffer, int lenght,
                int flags[, const struct sockaddr *dest_addr,
                int dest_len]);
```

socket	Socket to send from
buffer	Pointer to the message to be transmitted
length	Length of the message
flags	Type of transmission
dest_addr	Optional target socket
dest_len	Size of `*dest_addr` in memory

`recv[from]()`

Recieve a message on a socket.

```
// Unix
int recv[from](int socket, const void *buffer, size_t length,
               int flags[, struct sockaddr *restrict address,
                         socklen_t *restrict address_len]);
// Windows
int recv[from](SOCKET socket, const char *buffer, int length,
                      int flags[, struct sockaddr *address,
               int *address_len]);
```

socket	The connected socket
buffer	Pointer where to put the recieved message
length	Length of the message buffer
flags	Type of transmission
adress	Optional sending socket
adress_len	Size of `*adress` in memory
return value	#bytes recieved, 0 (connection closed), or -1 (failure)

Task

a simple server

let's write a program which listens on port `1337` and prints the send packet payload.

the output should be like this:

```
[sender address]: [message]
23.42.23.42: lame course :p
```

send us your name

send us your name on a tcp connection to `dvorak.krbs.me(IPv4 Address: 116.203.113.16)` on port `1337`

What you send us, will be printed on the the beamer.

have a try. Our program from the last task will run there.