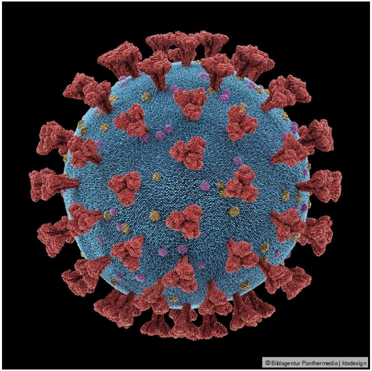
COVID-19 Measures

- Wear a mask (medical or FFP2) until you have taken a seat
- When seated you may take off the mask if you can maintain an interpersonal distance of 1,5 m
- Open the windows periodically whenever possible
- Behave reasonable and use common sense



What do we need to develop interoperable and portable software?

Distributed Systems
Sockets

Prof. Dr. Oliver Hahm
Frankfurt University of Applied Sciences
Faculty 2: Computer Science and Engineering
oliver.hahm@fb2.fra-uas.de
https://teaching.dahahm.de

Agenda

1. Motivation
2. UNIX Pipes and Sockets
3. Socket API

Agenda

Towards a Standard Network API

Since about 1980 most of the operating systems possess (still often proprietary) a interface for network access in order to allow communication with peer systems. Samples:

- Digital (DEC): VMS/OpenVMS $\leftrightarrow$ DECnet
- Novel: Netware $\leftrightarrow$ IPX/SPX
- IBM: MVS $\leftrightarrow$ VTAM/SNA, VM $\leftrightarrow$ IUCV
- Microsoft: Windows: $\leftrightarrow$ NetBIOS

Towards a Standard Network API

Since about 1980 most of the operating systems possess (still often proprietary) a interface for network access in order to allow communication with peer systems. Samples:

- Digital (DEC): VMS/OpenVMS $\leftrightarrow$ DECnet
- Novel: Netware $\leftrightarrow$ IPX/SPX
- IBM: MVS $\leftrightarrow$ VTAM/SNA, VM $\leftrightarrow$ IUCV
- Microsoft: Windows: $\leftrightarrow$ NetBIOS

This conflicts with goal of interoperability!

$\Rightarrow$ Nowadays, the **networking approach** possess the same level of usage w.r.t. the file storage concept. In fact, the boundaries between the access to local and remote resources get more and more blurry ($\rightarrow$ e.g., **cloud computing**).

Agenda

Unix Pipes

A **pipe** can be used to exchange information between processes by means of defined **file descriptors**:

```
# (1) Fitering of output by means of pipes
cat inputfile | grep -i 'error'

# (2) Redirection into file
cat inputfile | grep -i 'error' > errorfile

# (3) Reading from a file and use filterprog
filterprog < inputfile # or
cat inputfile | filterprog
```

- One process is writing into the pipe, while the other one is reading.
- Unix possesses usually three commonly used and distinct file descriptors:
  - **0** STDIN: Standard input
  - **1** STDOUT: Standard output
  - **2** STDERR: Standard error

- In Unix (and Windows as well) the sign '|' is a special command to couple the processes by means of the pipe; typically used to **filter** the output.
- The **output** can be **re-directed** to a file using the special command '>'.
- Processes can read (automatically) from a file (without open it explicitly) employing the command '<'.

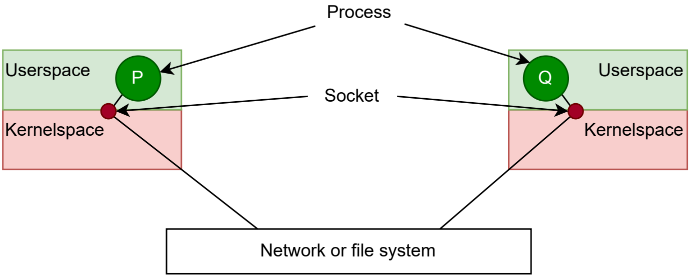
Co-Processes

A process invoked via the **pipe** is understood as **co-process**: The blocking of the **co-process** results in the blocking of the main process.

Unix Sockets

Sockets are part of the **TCP/IP** protocol family and have been introduced to Unix with BSD 4.2 in 1982 ($\rightarrow$ **Berkeley Sockets**).

- A **socket** is a **communication endpoint**
- It can be identified by the pair (IP address, Port number)
- In order two communicate, two sockets are required to be present on ...
  - different computing nodes $\rightarrow$ **Internet Sockets** or
  - the same node $\rightarrow$ **Domain Sockets** (realized as a **special** file)
- Sockets can be created and released from a process, and allow a bi-directional exchange of information among the peers.



Pipes and Sockets

Sockets and Unix **Pipes** are pretty much comparable from a usage point of view:

- **Pipes** use a **handle** on a file descriptor to exchange messages,
- **Sockets** use a **handle** for a network connection.

```
// Reading from a Unix Pipe
int read_fd (void) {
    unsigned long fd;
    char envbuf[8192];
    char *x;
    int j;

    while ((j = read(fd, envbuf, 8192)) > 0)
    { ... }
}
```

```
// Writing to an Internet Sockets
int tcp_write (void) {
    unsigned long intfd;
    int timeout = 20;
    int port;

    intfd = socket(AF_INET, SOCK_STREAM, 0);
    if (intfd == -1) temp_oserr();

    if (timeoutconn(intfd, &(ip.ix[i].ip), (unsigned int) port, timeout)
      == 0)
    { ... }
}
```

Domain Sockets

Unlike a **pipe** a **socket** provides a bi-directional connection between the communicating peers:

```
// Declaring a socket
#include <sys/types.h>
#include <sys/sockets.h>

int sockets[2];
int err_socket;

err_socket = socketpair(domain,type,protocol,sockets);

// Socket descriptors are stored in array sockets[2];
domain = AF_UNIX;      // AF_INET used for Internet
type = SOCK_STREAM;
protocol = 0;          // typical for TCP
```

Actually using a socket:

```
char buf[1024];

// Define DATA
read(sockets[1], buf, 1024);
write(sockets[2], DATA, sizeof(DATA));
```

$\Longrightarrow$ Socket files are typically created in `/tmp`

`srwxr-xr-x 1 hahm wheel 0 17 Jun 12:06 OSL_PIPE_1002_SingleOfficeIPC_d-....`

Agenda

1 Motivation

2 UNIX Pipes and Sockets

3 Socket API

Network Programming with Sockets

- Goal: message-oriented IPC between application parts on remote hosts
- Introduced in BSD UNIX 4.X in a C API
- Eventually became part of POSIX (Portable Operating System Interface)
- Today available for almost any OS (Windows, Linux, RIOT ...) in almost any programming language (Java, Python, C# ...)
- The most commonly used interface for programming network applications in TCP/IP environments
- Forms the foundation for all higher layer application layer protocols (like HTTP)
- Support client/server relationship between application components
- Java sockets represent BSD sockets as a set of classes

Types of Sockets

Stream Sockets: (`SOCK_STREAM`)

- Reliable communication (typically of a byte stream) between two endpoints
- Connection-oriented transport
- For Internet domain sockets TCP is the default protocol

Datagram Sockets: (`SOCK_DGRAM`)

- Unreliable communication of single messages (best-effort delivery)
- Connectionless datagram service
- For Internet domain sockets UDP is the default protocol

Raw Sockets: (`SOCK_RAW`)

- Allow access to underlying protocols like IP, ICMP ...
- Typically require superuser permissions

Streams and Datagram Sockets

Stream sockets realize a **rendezvous** between the client and the server by means of the following system **primitives**:

- Client: `connect();`
- Server: `accept();`

↪ Once `accept();` has been issued, the server is in **blocking** I/O mode.

Datagram socket **primitives**:

- Client: `sendto();`
- Server: `recfrom();`

↪ Messages are transmitted without the necessity of acknowledgments at the receiver side.

Socket Calls

The Berkeley Socket family provide the communication over IPv4 (`AF_INET`) and IPv6 (`AF_INET6`) networks using the following calls:

Primitive	Meaning
`socket`	Create a new communication endpoint
`bind`	Attach a local address to a socket
`listen`	Announce willingness to accept N connections
`accept`	Block until request to establish a connection
`connect`	Attempt to establish a connection
`send/sendto/write`	Send data over a connection
`receive/recvfrom/read`	Receive data over a connection
`select`	Wait on multiple I/O events
`shutdown`	Close a connection
`close`	Release the connection

Socket Datatypes

- Header files:

```c
#include <sys/types.h>
#include <sys/socket.h>
```

- IP address:

```c
struct in_addr { uint32_t s_addr; };
```

- Socket address (generic type, used in system calls):

```c
struct sockaddr {
   u_short sa_family;   // here AF_xxxx
   char    sa_data[14]; // up to 14~B type specific address
};
```

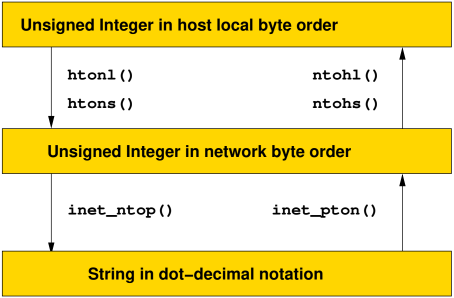
- Socket address (Internet type):

```c
struct sockaddr_in {
   u_short sin_family;      // here AF_INET, AF_INET6, or AF_UNIX
   u_short sin_port;        // Port Number (in network byte order)
   struct in_addr sin_addr; // IP-Adresse (in network byte order)
   char sin_zero[8];        // unused
};
```

- Cast:

```c
struct sockaddr_in my_addr;
...
(struct sockaddr*) &my_addr ...
```

Helper Functions: Address Conversion



Functions defined in
<sys/types.h>
<netinet/in.h>

Functions defined in
<sys/types.h>
<netinet/in.h>
<arpa/inet.h>

htonl()/htons():	host to network long/short
ntohl()/ntohs():	network to host long/short
inet_ntop():	network to presentation/printable
inet_pton():	presentation/printable to network

Helper Function: Address Translation

getaddrinfo()

```c
struct addrinfo {
  int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
  int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
  int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
  int              ai_protocol;  // use 0 for "any"
  size_t           ai_addrlen;   // size of ai_addr in bytes
  struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
  char            *ai_canonname;// full canonical hostname
  struct addrinfo *ai_next;      // linked list, next node
};
```

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
int getaddrinfo(
  const char *node,
  const char *service,
  const struct addrinfo *hints,
  struct addrinfo **res);
```

"'Given node and service, which identify an Internet host and a service, getaddrinfo() returns one or more addrinfo structures, each of which contains an Internet address that can be specified in a call to bind(2) or connect(2)."'

(⇒ replaces gethostbyname(), getservbyname())

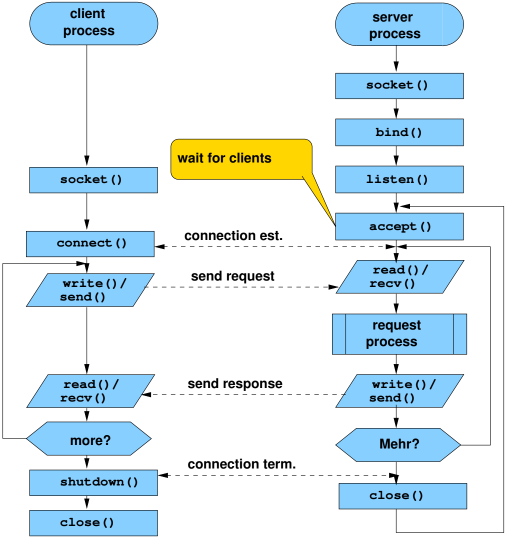
Example

```
int main(int argc, char *argv[])
{
    struct addrinfo hints;
    struct addrinfo *result;
    int s;
...
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_UNSPEC;    // Allow IPv4 or IPv6
    hints.ai_socktype = SOCK_DGRAM; // Datagram socket
    hints.ai_flags = AI_PASSIVE;    // For wildcard IP address
    hints.ai_protocol = 0;          // Any protocol
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
    s = getaddrinfo(NULL, argv[1], &hints, &result);
    if (s != 0) {
       fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
       exit(EXIT_FAILURE);
    }
```

More Helper Functions

`gethostname()`	Get the name of current host
`gethostid()`	Get the unique ID of current host
`getsockopt()`	Retrieve the current parameters of a socket
`setsockopt()`	Set the parameters of a socket

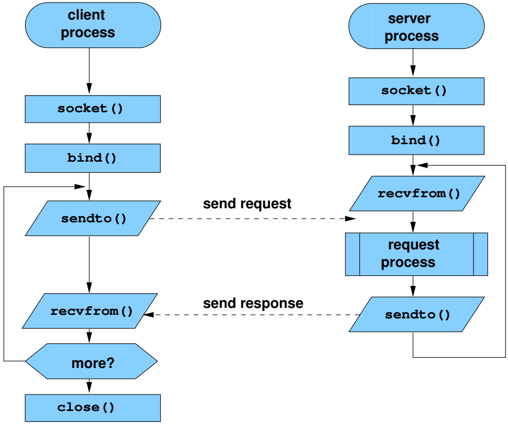
Simplified TCP Interaction



Generalization

- A server usually maintains multiple connections to clients.

Simplified UDP Interaction



socket()

Create a Socket

`int socket(int family, int type, int protocol)`

creates a socket for the Internet domain (`family=AF_INET`) or UNIX domain (`AF_UNIX`) of type stream socket (`type=SOCK_STREAM`), datagram socket (`SOCK_DGRAM`) or raw socket (`SOCK_RAW`) to be used with the protocol `protocol` and returns a descript for the created socket. For `protocol` typically the value 0 is passed. In this case the default protocol for the specified domain and socket type is selected. For the Internet domain TCP is the default for a stream socket and UPD for a datagram socket. No socket address is assigned yet → the socket is **unbound**.

Example:

```
sd1 = socket(AF_INET, SOCK_STREAM, 0)
sd2 = socket(AF_INET, SOCK_DGRAM, 0)
```

bind()

Binding of a Socket Address

`int bind(int sd, struct sockaddr *addr, int addrlen)`

binds the socket to the address that has been passed in `struct sockaddr`. The type of the address depends on the domain of the socket. For Internet domain sockets this structure is `struct sockaddr_in`, for Unix domain sockets a file name is passed. The socket is registered in the communication system. This clients of a connection-oriented communication this is not required.

Example:

```
rc = bind(sd, (struct sockaddr *) &my_addr, sizeof(my_addr))
```

`listen()`

Listen for Incoming Connection Requests

`int listen(int sd, int qlength)`

indicates that the socket `sd` is waiting for incoming connections. `qlength` is the maximum number of queued connection requests which have not yet been accepted (→ this is **not** the maximum number of possible clients.

Only required for the server site of connection-oriented communication.

Example:

```
rc = listen(sd, 5)
```

accept()

Accept Incoming Connection Requests

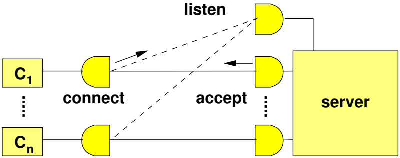
`int accept(int sd, struct sockaddr *claddr, int *addrlen)`

blocks until a new connection request of a client is received on socket `sd`. Then a new socket is created and its descriptor is returned. Hence, a a new, private connection between client and server is created. The socket `sd` is available for further connection requests again. The identity of the client (i.e, its remote socket address) is stored into the passed struct `claddr`. Its length is set accordingly in `addrlen`.

Only required for the server site of connection-oriented communication.

Example:

```
snew = accept(sd, &clientaddr, &clientaddrlen)
```



connect()

Connection Request

`int connect(int sd, struct sockaddr *saddr, int saddrlen)`

active connection request for a client using its socket `sd` to a server. The server's address is passed in `saddr` along with the address' length as `saddrlen`.

Only required for the client site of connection-oriented communication.

Example:

```
rc = connect(sd, &saddr, sizeof(saddr))
```

write()/send() und read()/recv()

Send

```
int write(int sd, char *buf, int len)
int send(int sd, char *buf, int len, int flag)
```

the call `write` is used in the same way as for file descriptors. The call `send` accepts an additional argument `flag` which can be used to set additional options.

Receive

```
int read(int sd, char *buf, int nbytes)
int recv(int sd, char *buf, int nbytes, int flag)
```

the call `read` is used in the same way as for file descriptors. The call `recv` accepts an additional argument `flag` which can be used to set additional options.

Example:

```
charcount = write(sd, buf, len)
charcount = send(sd, buf, len, sendflag)
charcount = read(sd, buf, len)
charcount = recv(sd, buf, len, recvflag)
```

shutdown()

Closing a Connection

`int shutdown(int sd, int how)`

Terminates a connection. The parameter `how` specify whether and how further transmission on this connection shall be handled.

The socket descriptor persists and has to be destroyed with a dedicated call to `close()`.

Example:

```
rc = shutdown(sd, 2)
```

select()

Wait for an I/O Event

```
#include <sys/time.h>
int select(int nfds, int *readmask, int *writemask,
int *exceptmask, struct timeval *timeout)
```

allows the monitoring of multiple socket or file descriptors in a single process. The calling process blocks until a particular event (e.g., the descriptor becomes *readable*) occurs for one of the specified descriptors – or the given timeout expires. The maximum waiting time (timeout) may be limited or unlimited.

The set of descriptors are passed via bitmasks. For this purpose some macro functions, e.g., `FD_SET`, exist.

When the function returns the value of `readmask` has changed and contains the bitmask of these descriptors where the event has occurred. The return value indicates the number of these descriptors.

Example:

```
int sd1, sd2;
fd_set fds;
sd1 = socket(AF_INET,...);
sd2 = socket(AF_INET,...);
...
```

```
FD_ZERO(&fds);
FD_SET(sd1,&fds);
FD_SET(sd2,&fds);
rc=select(FD_SETSIZE,&fds,
NULL,NULL,timeout);
```

Java Sockets

Provides an interface for the underlying BSD sockets via multiple interfaces and classes of the package `java.net`.

Addressing

- `InetAddress` with subclasses `Inet4Address` and `Inet6Address`
- `SocketAddress` with subclass `InetSocketAddress`

TCP Connections

- `ServerSocket`
- `Socket`
- For established connections: `getInputStream()`/`getOutputStream()`

Datagram communication via UDP

- `DatagramPacket`
- `DatagramSocket` – `MulticastSocket`

Server Sockets for Streams

For each configured IP address (`IPv4/IPv6`) of the server, the available **ports** (up to 64K) may be bound to exactly <u>one</u> server process.

- Ports below 1024 are <u>privileged</u> ports and may only be used with particular permissions (Unix **root** user).
- The server processes **binds** to that port while providing a **passively** open communication socket.

Once the client is going to connect to $IP_{Server} : Port_{Server}$, the socket is **cloned** (while a new copy of the server process is **instantiated**) and becomes **active**.

Sockets in the Unix OS

The command 'netstat' gives an answer which IP and domain sockets are currently active:

```
% netstat
Active Internet connections
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  artemis.49497          freebsd.isc.org.ftp    TIME_WAIT
tcp4       0      0  artemis.58716          hamburg134.serve.ssh   ESTABLISHED
udp4       0      0  localhost.50167        localhost.domain
udp4       0      0  localhost.domain       *.*
Active UNIX domain sockets
Address  Type   Recv-Q Send-Q    Inode     Conn     Refs  Nextref Addr
c5bf9870 stream      0      0        0 c57bc360        0        0 /tmp/.X11-unix/X0
c57bc360 stream      0      0        0 c5bf9870        0        0
c5bf9750 stream      0      0        0 c57bc870        0        0 /tmp/.X11-unix/X0
c57bc870 stream      0      0        0 c5bf9750        0        0
c57bc510 stream      0      0        0 c60d21b0        0        0 /tmp/.X11-unix
c5bf9090 stream      0      0 c60cd440        0        0        0 /var/tmp/dbus-OoNlIvGBXW
c57bcbd0 stream     32      0        0 c57bcb40        0        0
c57bcc60 stream      0      0 c5c09330        0        0        0 /tmp/.X11-unix/X0
c57bbea0 stream      0      0 c5a09dd0        0        0        0 /tmp/GNUstepSecure0/NSMess ...
c57bb120 stream      0      0 c59c8330        0        0        0 /var/run/cups.sock
c57bb1b0 stream      0      0        0 c57bb240        0        0
c57bb090 dgram       0      0        0 c57bbd80        0        0
c57bb000 dgram       0      0        0 c57bbcf0        0        0
c57bbcf0 dgram       0      0 c57d2cc0        0 c57bccf0        0 /var/run/logpriv
c57bbd80 dgram       0      0 c57d2dd0        0 c57bb090        0 /var/run/log
```

Figure: Output of netstat on a not very busy *nix system

Internet versus Domain Sockets

Unix Domain Sockets

- Can only be used on the same node (requiring a context-switch only)
- Same **API** like the IP sockets, however do not require ...
  - any underlying communication protocol like **TCP/IP**
  - any calculation (and verification) of **checksums**
- The use the **file system** to maintain the **name space**
  - The effective Unix **permissions** (`rwx`) are usable, in particular while creating the socket
  - ⇒ Only those user (on the very same node) belonging to the respective user/group have permissions to use the sockets
  - Domain sockets inherit the permissions from the process owner

IP Sockets

- IP sockets realize **network transparency** (connectivity to a remote node).
- May operate using TCP streams of UDP datagrams as communication protocol
  - → perhaps requiring the **session overhead** of the TCP service
- IP sockets via **localhost**
  - Use the **loopback** interface of the operating system
  - Behave in the same way, as usual IP sockets
- Require **two** context switches (at the client and the server side) to exchange the data.

Alternatives to POSIX

- TLI and STREAMS
  - UNIX' **Transport Layer Interface (TLI)** was based on STREAMS, a framework for implementing, e.g., network protocols and IPC
  - TLI was developed mostly with OSI protocols in mind
  - Today only relevant for historical reasons
- sock on RIOT
  - Designed for constrained devices, i.e., for example, no need for dynamic memory allocation
  - Protocol specific interfaces for various network stacks
  - Wrapper for POSIX sockets is implemented on top
  - Currently provides interfaces for TCP, UDP, RAW IP, DNS, and DTLS

Important takeaway messages of this chapter

- In order to implement platform independent distributed applications a common communication API is required
- The BSD Socket API became the de facto standard for programming network applications
- This API consists of less than 20 functions to achieve a generic functionality