# Socket Programming in C

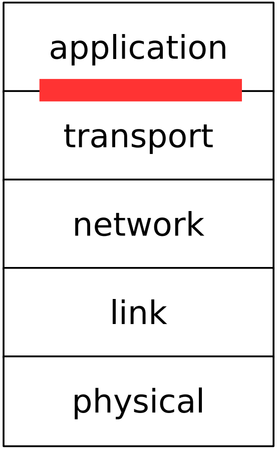
Introduction to Internet and Security

Claudio Marxer

<claudio.marxer@unibas.ch>

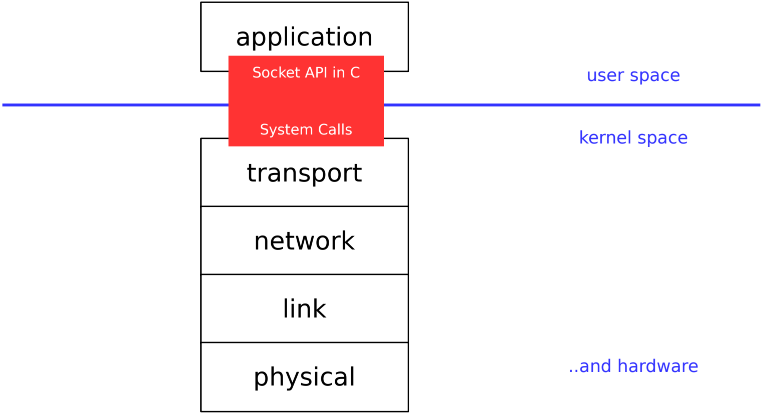
UNI
BASEL

# Big Picture: Network View

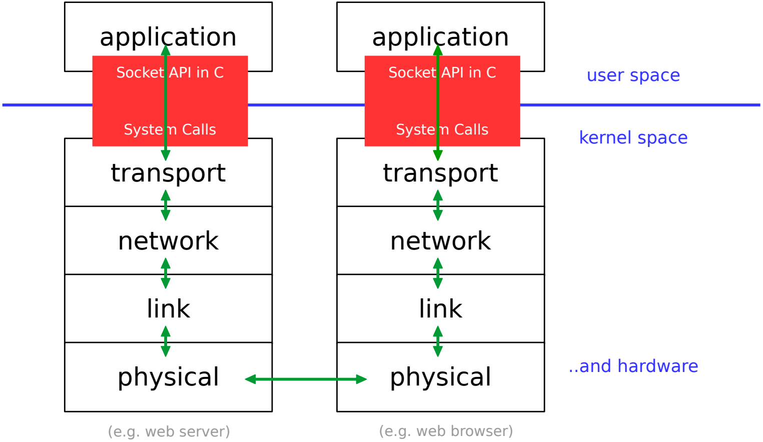


| | |
|---|---|
| application | Webbrowser und Server ... |
| transport | TCP, UDP |
| network | IP |
| link | Ethernet, IEEE 802.11, ... |
| physical | ... |

# Big Picture: OS View

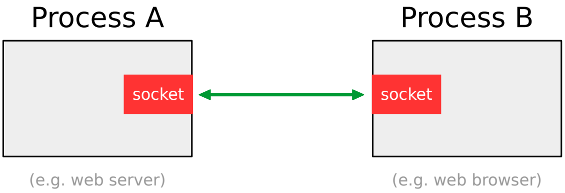


# Big Picture: OS View (2)



# What is a Socket?

- A socket is a bi-directional communication abstraction (called association) between two processes.
- A socket is a communication endpoint
- Sockets are an Application Programming Interface (API) for Inter-Process-Communication (IPC)



# Socket Types

- `SOCK_STREAM`: Connection Oriented, Guaranteed Delivery (e.g. TCP)
- `SOCK_DGRAM`: Datagram-Based Communication (e.g. UDP)
- `SOCK_RAW`: Direct access to the network layer

  E.g. build ICMP messages or custom IP packets
- `SOCK_PACKET`: Direct access to the link layer

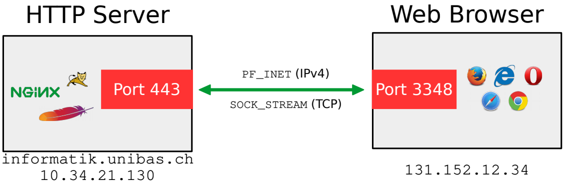
  ...

# Protocol Families

There are many domains and protocols families (PF) available:

- `PF_INET`: IPv4 (32-bit address length)
- `PF_INET6`: IPv6 (64-bit address length)
- `PF_UNIX`: Interprocess communication on local machine
- `PF_APPLETALK`: Appletalk Networks
- `PF_IPX`: Novell Netware Networks
- ...
- `PF_ALG`: Kernel Crypto API (linux only)

# Hands on..



## How to “see” sockets on your system:

```
$ netstat -ap -A inet
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
...
tcp        0      0 Notebook:3348           10.34.21.130:https      ESTABLISHED 9094/firefox
...
```

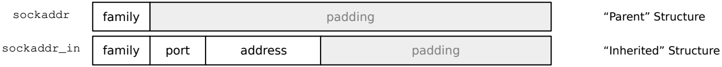
# API: Berkeley Sockets / POSIX Sockets

- Address Configuration
- Socket Creation, Binding, Listening
- Initiation and Acception of a Connection
- Sending and Receiving Data
- Socket Destruction
- Programming Techniques (monitoring set of sockets, error handling)

# Address Configuration

```
struct sockaddr_in {
    short            sin_family;    // address family
    unsigned short   sin_port;      // port number
    struct in_addr   sin_addr;      // IP address
    char             sin_zero[8];   // zero for padding
};
```

```
struct in_addr {
    unsigned long s_addr;
};
```



sockaddr family padding “Parent” Structure

sockaddr_in family port address padding “Inherited” Structure

# Watch the Conversions!

$$\boxed{3456789_{\text{dec}} \;\widehat{=}\; 00\ 34\ \mathrm{BF}\ 15_{\text{hex}}}$$

There are two ways to encode this number in 32 bits:

- Litte-Endian (Intel..): Least significant byte in lowest memory address.

|  |  |  |  |
|---|---|---|---|
| 15 | BF | 34 | 00 |

- Big-Endian (Motorola, IBM..): Most significant byte in lowest memory address.

|  |  |  |  |
|---|---|---|---|
| 00 | 34 | BF | 15 |

⚠ `sin_port` and `sin_addr` must always be converted between *host byte order* and *network byte order* (big-endian) after RX and before TX!

# Conversion Functions

- `htonl(..)` host to network long

  Before TX: Convert a long integer...

- `htons(..)` host to network short

  Before TX: Convert a short integer...

- `ntohl(..)` network to host long

  After RX: Convert ...

- `ntohs(..)` network to host short

  After RX: Convert ...

```c
struct sockaddr_in unibas_web;
unibas_web.sin_family = AF_INET;                             // address family IPv4
unibas_web.sin_port = htons(80);                             // convert port to network byte order
unibas_web.sin_addr.s_addr = inet_addr("131.152.228.33");    // convert address to network byte order
```

# Socket Creation

```
int socket (int domain, int type, int protocol);
```

domain Specifies the domain (protocol family) in which a socket is to be created.
PF_INET, PF_INET6, PF_UNIX,...

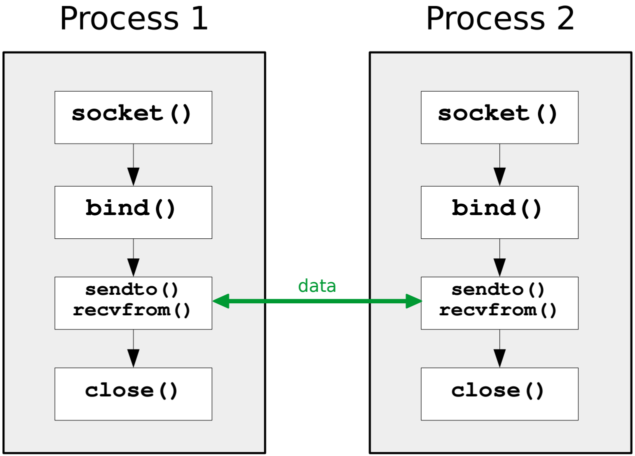
type Specifies the type of socket to be created.
SOCK_DGRAM, SOCK_STREAM

protocol Specifies a particular protocol to be used with the socket.
0 for default protocol or IPPROTO_UDP, IPPROTO_TCP,...

The returned integer value is a *descriptor*: Distincts the socket from other objects (e.g. sockets, files, pipes, I/O ressources) at operating system level.

```
int tcp_socket = socket(AF_INET, SOCK_STREAM, 0);
```

# UDP: Datagram-Based Communication (`SOCK_DGRAM`)



# Binding to Interface and Port Number

**Binding:** Assign an IP address (interface) and a port number to a socket.

```c
int sockDesc = socket(PF_INET, SOCK_STREAM, 0);

struct sockaddr_in locAddr;
locAddr.sin_family = AF_INET;
locAddr.sin_port = htons(0);                  // let the system choose
locAddr.sin_addr.s_addr = htonl(INADDR_ANY);  // any addr of this sys

bind(sockDesc, (struct sockaddr*) &locAddr, sizeof(locAddr));
```

# Binding to Interface and Port Number (2)

**Don't forget to read auto-values back:** A port number automatically chosen by the operating system (sin_port=0) is not automatically saved in the structure.

```
// output argument, not input!
int length;

// update port number
getsockname(sockDesc, (struct sockaddr*) &locAddr, &length);

// print updated port number
printf("Port chosen by the OS: %d", ntohs(locAddr.sin_port));
```

# Sending and Receiving Packets

```c
#define BUFLEN  12
```

### Sending

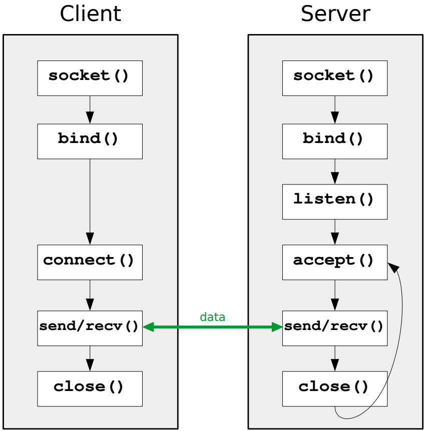
```c
char buf[BUFLEN] = "Hello World!";
sendto( sockDesc, buf, BUFLEN, 0, (struct sockaddr*) &peerAddr,
    peerAddrLen );
```

### Receiving

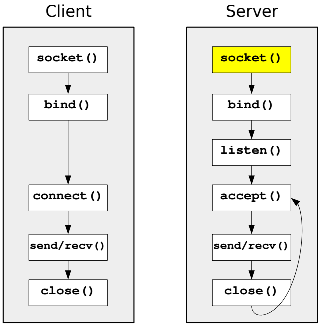
```c
char buf[BUFLEN];
recvfrom(sockDesc, buf, BUFLEN, 0, (struct sockaddr*) &peerAddr,
    &peerAddrLen);
```

→ **Code Demo**

# TCP: Stream-Based Communication (SOCK_STREAM)



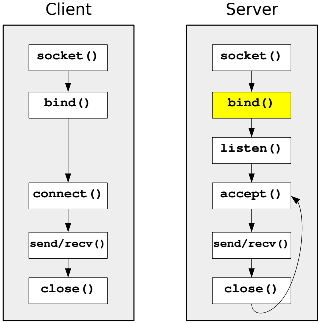
# Walk-Through Example: TCP Echo Server



## Server: Create Socket

```
int servSock = socket(PF_INET, SOCK_STREAM, 0);
```

# Walk-Through Example: TCP Echo Server

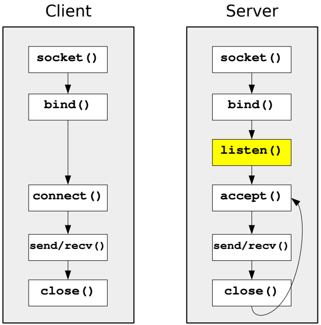


## Server: Binding to Local IP Address and Port

```
struct sockaddr_in servAddr;
servAddr.sin_family = AF_INET;
servAddr.sin_port = htons(8080);
servAddr.sin_addr.s_addr = htonl(INADDR_ANY);

bind(servSock, (struct sockaddr*) &servAddr,
     sizeof(servAddr));
```

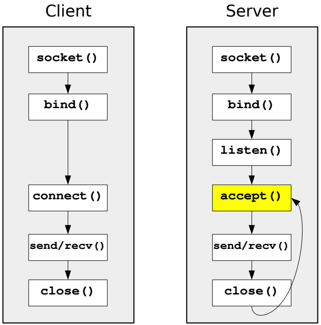
# Walk-Through Example: TCP Echo Server



**Server: Listening for Incomming Connections**

```
listen(servSock, MAXPENDING);
```

# Walk-Through Example: TCP Echo Server



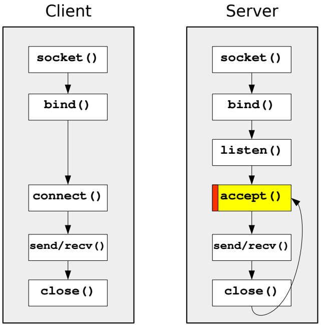
## Server: Accept Incomming Connections

```
for(;;) {
   struct sockaddr_in cltAddr;
   int cltAddrLen = sizeof(cltAddr);
   int cltSock;

   cltSock = accept(servSock,
                    (struct sockaddr*) &cltAddr,
                    &cltAddrLen);

   ... // later: RX, TX, close
}
```

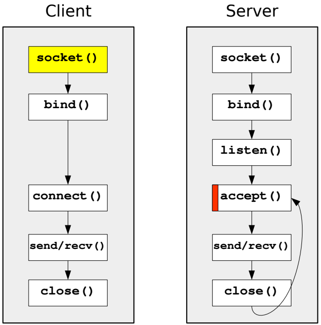
# Walk-Through Example: TCP Echo Server



## Server: Accept Incomming Connections (2)

- Server is now waiting for an incomming connection.
- So far, the server is blocking.

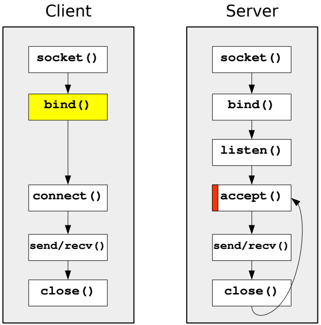
# Walk-Through Example: TCP Echo Server



## Client: Create Socket

```
int localSock = socket(PF_INET, SOCK_STREAM, 0);
```

# Walk-Through Example: TCP Echo Server

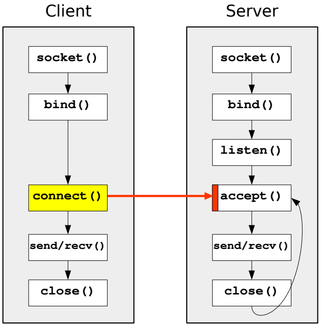


## Client: Bind to Local IP Address and Port

```
struct sockaddr_in localAddr;
localAddr.sin_family = AF_INET;
localAddr.sin_port = htons(0);
localAddr.sin_addr.s_addr = htonl(INADDR_ANY);

bind(localSock, (struct sockaddr*) &localAddr,
     sizeof(localAddr));
```

# Walk-Through Example: TCP Echo Server

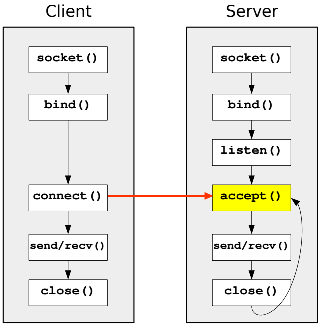


## Client: Connect to Server

```
struct sockaddr_in srvAddr;
srvAddr.sin_family = AF_INET;
srvAddr.sin_port = htons(8080);
srvAddr.sin_addr.s_addr = inet_addr("1.2.3.4");

connect(localSock, (struct sockaddr*) &srvAddr,
    sizeof(srvAddr));
```

# Walk-Through Example: TCP Echo Server



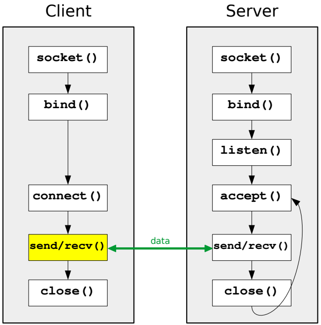
## Server: Accept Incomming Connections

Server process is blocked since..

```
cltSock = accept(servSock,
                 (struct sockaddr*) &cltAddr,
                 &cltAddrLen);
```

.. now `accept(..)` returns a new socket descriptor.

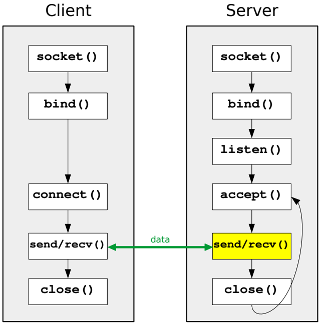
# Walk-Through Example: TCP Echo Server



## Client: Send a Message and Wait for Reply

```
char msg[] = "Ping-Pong Message!";
send(localSock, msg, strlen(msg), 0);
// recv(..) not shown here..
```

# Walk-Through Example: TCP Echo Server

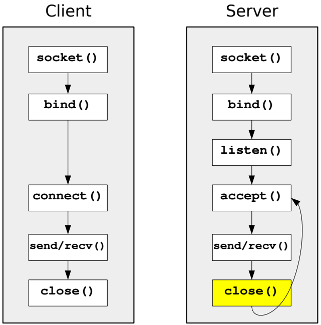


## Server: Receive Message and Reply

```
#define  BUFSIZE  32
char buf[BUFSIZE];
int bytesRecv;

bytesRecv = recv(cltSock, buf, BUFSIZE, 0);
while(bytesRecv > 0) {  // 0 means end of transm.
  send(cltSock, buf, bytesRecv, 0); // ECHO
  bytesRecv = recv(cltSock, buf, BUFSIZE, 0);
}
```

# Walk-Through Example: TCP Echo Server

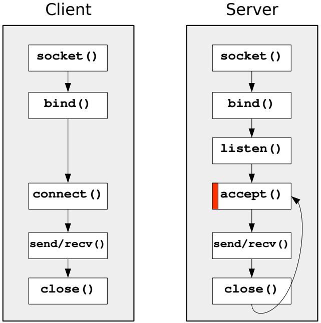


## Server: Teminate Connection

```
for(;;) {
    ...
    cltSock = accept(...);
    ... // ECHO

    close(cltSock);
}
```

# Walk-Through Example: TCP Echo Server



## Server: Accept Pending Client or Wait

```
for(;;) {
   ...
   cltSock = accept(...);
   ...
}
```

Server either..

..accepts the next pending connection.

..is again waiting/blocking.

# TCP Socket: Streaming and Segmentation

**What does “streaming” (SOCK_STREAM) mean?**

- The API offers the service to *stream* a byte buffer from one process to another.
- The underlying implementation (segmentation) is out of the process’ control.

```
// Process A
send("How are you?")
.
.
.
.
.
recv() -> "I’m fine!"
```

```
// Process B
.
recv() -> "Ho"
recv() -> "w ar"
recv() -> "e you?"
send("I’m ")
send("fine!")
.
```

# TCP Socket: Streaming and Segmentation (2)

**What to terminate a stream?**

- `EOF` (end-of-file) analogy: `close(..)`

```
// Process A
send(..)
.
close(..)
.
.
```

```
// Process B
.
while (recv(..) > 0)
  send(..)
.
close(..);
```

# Be Careful: Blocking Calls!

⚠ These are blocking: `recv(..)` `send(..)` `recvfrom(..)` `sendto(..)`

```
recvfrom(..);  // The process is blocked until it receives anything..
sendto(..);  // As long as recv(..) is blocking, process can not send!
```

**Solutions:**

- Multi-threading
- Use `select(..)` function

# Multi-Threaded Server

**New thread per accepted connection.**

```
for (;;) {
  cltSock = accept(srvSock, (struct sockaddr *) &cltAddr, ...);
  pid = fork();
  if (pid == 0) {  // child thread
    close(srvSock);
    dostuff(cltSock);
    exit(0);
  }
  else {  // parent thread
    close(cltSock);
    // continue loop:
    // accept or wait for next connection..
  }
}
```

Sketchy, for simplicity some parts are hidden.

# `select(..)` Function

Useful to monitor multiple sockets (or FDs in general) to see if they..

- .. have data waiting to be received.
- .. or if the program has data to send.
- .. or if an exception occurred.

→ `man 2 select`

# Last but not Least: Error Handling

Socket programming can easily lead to errors. Therefore it is mandatory to implement error checking:

```
if (bind(...) < 0) {
    perror("Bind: ");    // print explicit error message
    return -1;
}
```

Use `perror(..)` with all calls!

# Help Yourself!

**Linux Programmer's Manual** (recommended)

```
$ man 2 socket
$ man 2 bind
$ man 2 listen
...
```

**POSIX Programmer's Manual**

```
$ man 3 socket
$ man 3 bind
$ man 3 listen
...
```

*Find out header files names and further details..*

# Summary

- The Socket API provides a user-level abstraction for communication associations.
- A single API for different communication types..

  (streaming, datagram-oriented, connection-oriented, connection-less)

  ..and different protocols.

  (e.g. UDP, TCP, local inter-process communication)
- A lot of functions:

  `socket(..)`, `bind(..)`, `listen(..)`, `connect(..)`, `accept(..)`, `close(..)`
  `send(..)`, `recv(..)`, `sento(..)`, `recvfrom(..)`
- Programming techniques: Multi-threading, `select(..)`, error handling

```
if(!questions) {
  if(enough_time()) {
    project_introduction();
  }
  printf("Bye!\n");
  return 0;
}
```