Programming with network Sockets

Computer Science Department, University of Crete

Manolis Surligas surligas@csd.uoc.gr

October 22, 2015

Goal of this lab

- Learn to create programs that communicate over a network
- Create TCP and UDP sockets using the POSIX Socket API
- Handle properly data

The POSIX Socket API

What is POSIX?

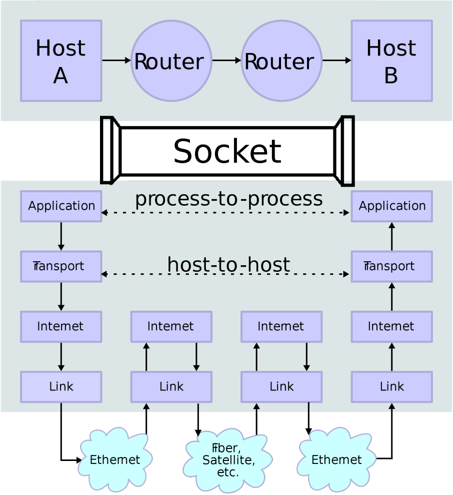
Portable **O**perating **S**ystem **I**nterface, is a family of standards specified by the IEEE for maintaining compatibility between operating systems.

- There are several Sockets implementations (e.g Berkeley, BSD)
- POSIX Socket API, provides a cross-platform and reliable way for network and inter-process communication

What is a Socket?

- Socket is an endpoint of communication between two processes
- Two basic types of sockets:
  - UNIX sockets
  - Network sockets
- Processes read and write data to the sockets in order to communicate

What is a Socket?



Transport Layer

- Transport layer is responsible for providing end-to-end data transfer between two hosts

- Two main protocols are used:
  - **TCP**
  - **UDP**

Transport Layer: TCP

- Connection-oriented communication
- Reliable, in-order and error free data delivery
- Flow-control, congestion avoidance

Transport Layer: UDP

- Connection-less communication
- Packets may be lost
- Packets may arrive in wrong order
- Packets may contain wrong data
- There is no guaranty that packets sent will reach their destination
- Used when low latency is critical (e.g VoIP, streaming, e.t.c.)

Creating a Socket

Prototype

```c
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- **socket()** creates a socket of a certain domain, type and protocol specified by the parameters
- Possible domains:
  - AF_INET for IPv4 internet protocols
  - AF_INET6 for IPv6 internet protocols

Creating a Socket

Prototype

```c
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- **socket()** creates a socket of a certain domain, type and protocol specified by the parameters
- Possible types:
  - SOCK_STREAM provides reliable two way connection-oriented byte streams (TCP)
  - SOCK_DGRAM provides connection-less, unreliable messages of fixed size (UDP)
- protocol depends on the domain and type parameters. In most cases 0 can be passed

Creating a Socket

SOCK_STREAM

Sockets of this type are full-dublex data streams that do not rely on a known data length. Before sending or receiving the socket must be in a connected state. To send and receive data, **send()** and **recv()** system calls may be used. By default, socket of this type are blocking, meaning that a call of **recv()** may block until data arrive from the other side. At the end, **close()** should be used to properly indicate the end of the communication session.

SOCK_DGRAM

This kind of sockets allowing to send messages of a specific size without the guarantee that they will be received from the other side. To send and receive messages **sendto()** and **recvfrom()** calls may be used.

TCP: Creating the socket

- Lets try to create our first TCP socket!

```
int sock;
if((sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == -1){
  perror("opening TCP listening socket");
  exit(EXIT_FAILURE);
}
```

- Always check for errors! Using **perror()** printing a useful and meaningful message is very easy!
- Opening a TCP socket is exactly the same for both server and client side

Bind a Socket

Prototype

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

int bind(int socket, const struct sockaddr *address,
         socklen_t address_len);
```

- **bind()** assigns an open socket to a specific network interface and port
- **bind()** is very common in TCP servers because they should waiting for client connections at specific ports

TCP: Bind the socket

```c
struct sockaddr_in sin;
memset(&sin, 0, sizeof(struct sockaddr_in));
sin.sin_family = AF_INET;
sin.sin_port = htons(listening_port);
sin.sin_addr.s_addr = htonl (INADDR_ANY);

if(bind(sock, (struct sockaddr *)&sin,
sizeof(struct sockaddr_in)) == -1){
 perror("TCP bind");
 exit(EXIT_FAILURE);
}
```

- Always reset the struct **sockaddr_in** before use
- Addresses and ports must be assigned in **Network Byte Order**
- **INADDR_ANY** tells the OS to bind the socket at all the available network interfaces

Listening for incoming connections

Prototype

```
int listen(int socket, int backlog);
```

- After binding to a specific port a TCP server can listen at this port for incoming connections
- backlog parameter specifies the maximum possible outstanding connections
- Clients can connect using the **connect()** call

Hint! (Old Linux distributions)

For debugging you can use the **netstat** utility! Try:

```
bash$ netstat -ltpn
```

Listening for incoming connections

Prototype

```
int listen(int socket, int backlog);
```

- After binding to a specific port a TCP server can listen at this port for incoming connections
- backlog parameter specifies the maximum possible outstanding connections
- Clients can connect using the **connect()** call

Hint! (Recent Linux distributions)

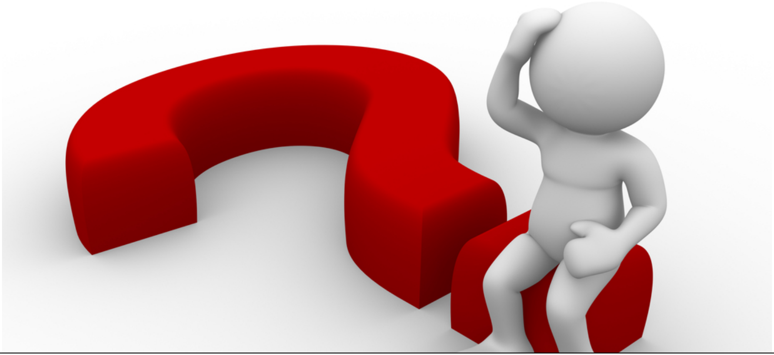
For debugging you can use the **ss** utility! Try:

```
bash$ ss -ltpn
```

Trivia

Think!

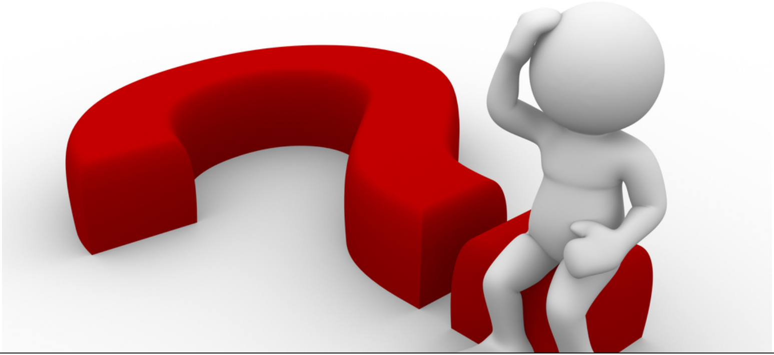
Which of the calls of the previous slides cause data to be transmitted or received over the network?



Trivia

Think!

Which of the calls of the previous slides cause data to be transmitted or received over the network? **NONE!**



TCP: Accepting connections

Prototype

```
#include <sys/socket.h>
int accept(int socket, struct sockaddr *restrict address,
           socklen_t *restrict address_len);
```

- **accept()** is by default a blocking call
- It blocks until a connection arrives to the listening socket
- On success a new socket descriptor is returned, allowing the listening socket to handle the next available incoming connection
- The returned socket is used for sending and receiving data
- If **address** is not NULL, several information about the remote client are returned
- **address_len** before the call should contain the size of the **address** struct. After the call should contain the size of the returned structure

TCP: Connecting

Prototype

```
#include <sys/socket.h>
int connect(int socket, const struct sockaddr *address,
            socklen_t address_len);
```

- Connects a socket with a remote host
- Like **bind()**, zero the contains of **address** before use and assign remote address and port in Network Byte Order
- If **bind()** was not used, the OS assigns the socket to all the available interfaces and to a random available port

TCP: Sending Data

Prototype

```
#include <sys/socket.h>
ssize_t send(int socket,
             const void *buffer,
             size_t length, int flags);
```

- **send()** is used to send data using a connection oriented protocol like TCP
- Returns the actual number of bytes sent
- Always check the return value for possible errors or to handle situations where the requested buffer did not sent completely

Question!

Does this call block?

TCP: Sending Data

Prototype

```
#include <sys/socket.h>
ssize_t send(int socket,
             const void *buffer,
             size_t length, int flags);
```

- **send()** is used to send data using a connection oriented protocol like TCP
- Returns the actual number of bytes sent
- Always check the return value for possible errors or to handle situations where the requested buffer did not sent completely

Question!

Does this call block? **YES!**

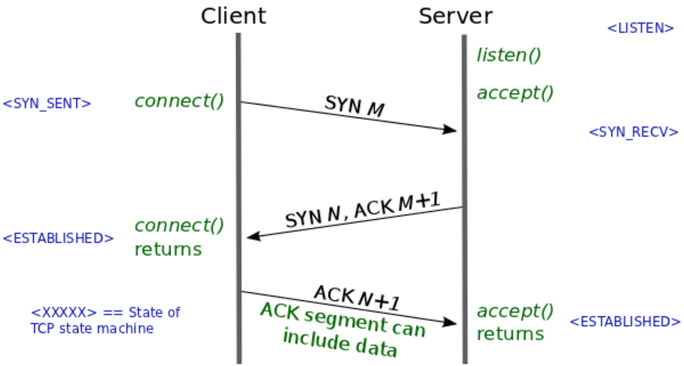
TCP: Receiving Data

Prototype

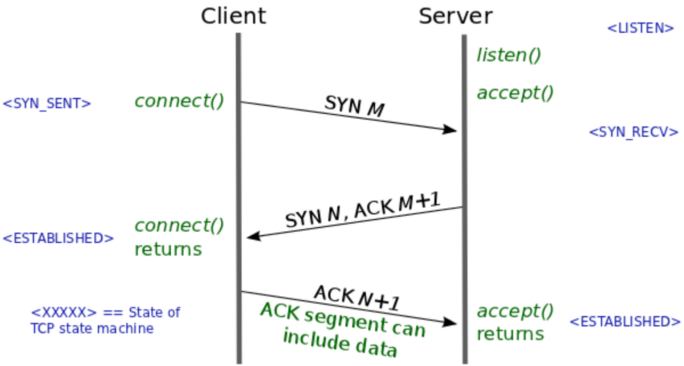
```
#include <sys/socket.h>
ssize_t recv(int socket, void *buffer,
             size_t length, int flags);
```

- **recv()** is by default a blocking call that receives data from a connection-oriented opened socket
- **length** specifies the size of the buffer and the maximum allowed received data chunk
- Returns the number of bytes received from the network
- **recv()** may read less bytes than **length** parameter specified, so use only the return value for your logic
- If you do not want to block if no data are available, use non-blocking sockets (hard!) or **poll()**

TCP Overview



TCP Overview



In high society, TCP is more welcome than UDP. At least it knows a proper handshake.

UDP: Creating the socket

- Creating a UDP socket is quite the same as with TCP

```
int sock;
if((sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1){
  perror("opening UDP socket");
  exit(EXIT_FAILURE);
}
```

- Only **type** and **protocol** parameters are different
- **bind()** is also exactly the same for UDP too

UDP: Connection-less

UDP is connection-less!!!
No need to call **accept()** or **connect()**!!!

UDP: Receiving data

Prototype

```c
#include <sys/socket.h>
ssize_t recvfrom(int socket, void *restrict buffer,
                 size_t length, int flags,
                 struct sockaddr *restrict address,
                 socklen_t *restrict address_len);
```

- **length** specifies the length of the buffer in bytes
- **address** if not NULL, after the call should contain information about the remote host
- **address_len** is the size of the struct **address**
- Returns the number of bytes actually read. May be less that **length**

UDP: Problems at receiving

- Have in mind that **recvfrom()** is a blocking call
- How you can probe if data are available for receiving?

UDP: Problems at receiving

- Have in mind that **recvfrom()** is a blocking call
- How you can probe if data are available for receiving?
  - Use **poll()**

UDP: Problems at receiving

- Have in mind that **recvfrom()** is a blocking call
- How you can probe if data are available for receiving?
  - Use **poll()**
- What if the message sent is greater that your buffer?

UDP: Problems at receiving

- Have in mind that **recvfrom()** is a blocking call
- How you can probe if data are available for receiving?
  - Use **poll()**
- What if the message sent is greater that your buffer?
  - Use **recvfrom()** in a loop with **poll()**

UDP: Sending data

Prototype

```c
#include <sys/socket.h>
ssize_t sendto(int socket, const void *message,
               size_t length, int flags,
               const struct sockaddr *dest_addr,
               socklen_t dest_len);
```

- **length** is the number of the bytes that are going to be sent from buffer **message**
- **dest_addr** contains the address and port of the remote host
- Returns the number of bytes sent. May be less that **length** so the programmer should take care of it

UDP: Sending data

Prototype

```c
#include <sys/socket.h>
ssize_t sendto(int socket, const void *message,
               size_t length, int flags,
               const struct sockaddr *dest_addr,
               socklen_t dest_len);
```

- **length** is the number of the bytes that are going to be sent from buffer **message**
- **dest_addr** contains the address and port of the remote host
- Returns the number of bytes sent. May be less that **length** so the programmer should take care of it

Trivia!

Does **sendto()** block?

UDP: Sending data

Prototype

```
#include <sys/socket.h>
ssize_t sendto(int socket, const void *message,
               size_t length, int flags,
               const struct sockaddr *dest_addr,
               socklen_t dest_len);
```

- **length** is the number of the bytes that are going to be sent from buffer **message**
- **dest_addr** contains the address and port of the remote host
- Returns the number of bytes sent. May be less that **length** so the programmer should take care of it

Trivia!

Does **sendto()** block? **NO!**

Endianness

- Networks are heterogeneous with many different OS's, architectures, etc
- Endianess is a serious problem when sending data to other hosts
- When sending entities that are greater that a byte, **always** convert them in **Network Byte Order**
- By default Network Byte Order is Big-Endian
- Use **htons()**, **ntohs()**, **htonl()**, **ntohl()**

Endianness

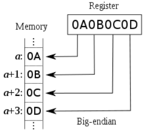
- Networks are heterogeneous with many different OS's, architectures, etc
- Endianess is a serious problem when sending data to other hosts
- When sending entities that are greater that a byte, **always** convert them in **Network Byte Order**
- By default Network Byte Order is Big-Endian
- Use **htons()**, **ntohs()**, **htonl()**, **ntohl()**

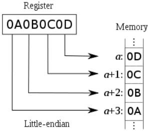
Trivia!

When sending large strings do we have to convert in Network Byte Order?

Endianness

- Networks are heterogeneous with many different OS's, architectures, etc
- Endianess is a serious problem when sending data to other hosts
- When sending entities that are greater that a byte, **always** convert them in **Network Byte Order**
- By default Network Byte Order is Big-Endian
- Use **htons()**, **ntohs()**, **htonl()**, **ntohl()**





Trivia!

When sending large strings do we have to convert in Network Byte Order? **NO!**

Useful man pages

- socket(7)
- ip(7)
- setsockopt(3p)
- tcp(7)
- udp(7)

Questions??