Network programming(II)

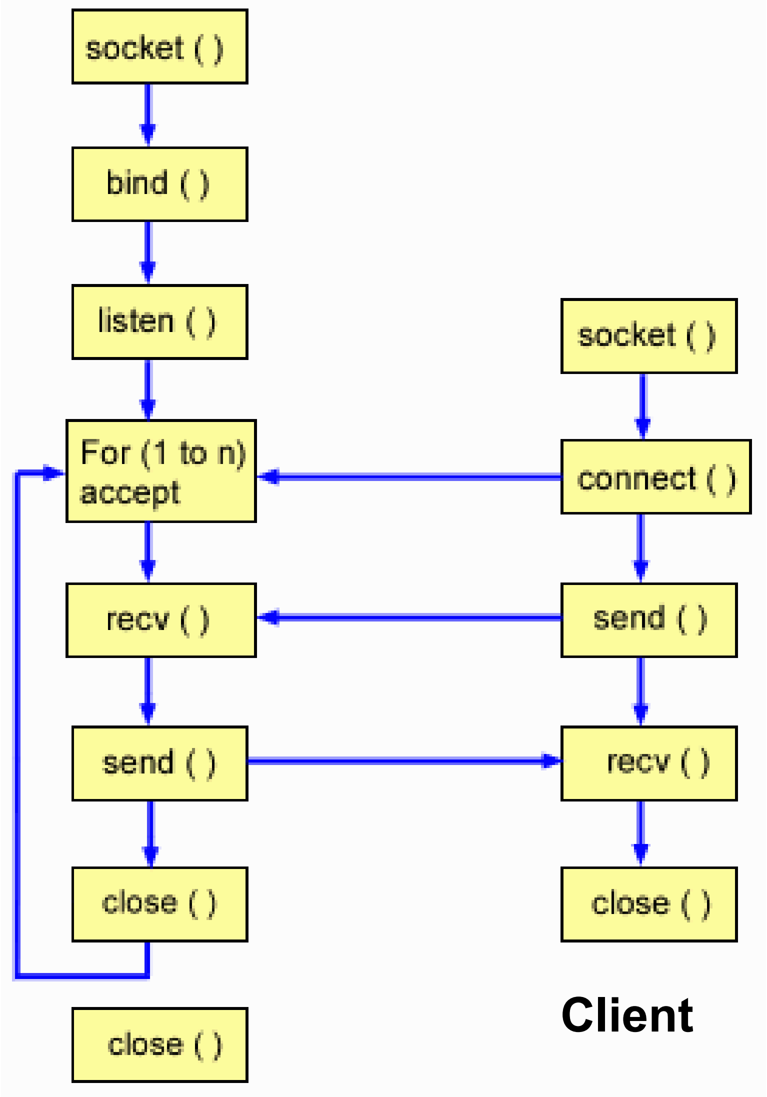
Lenuta Alboaie

adria@info.uaic.ro

Content

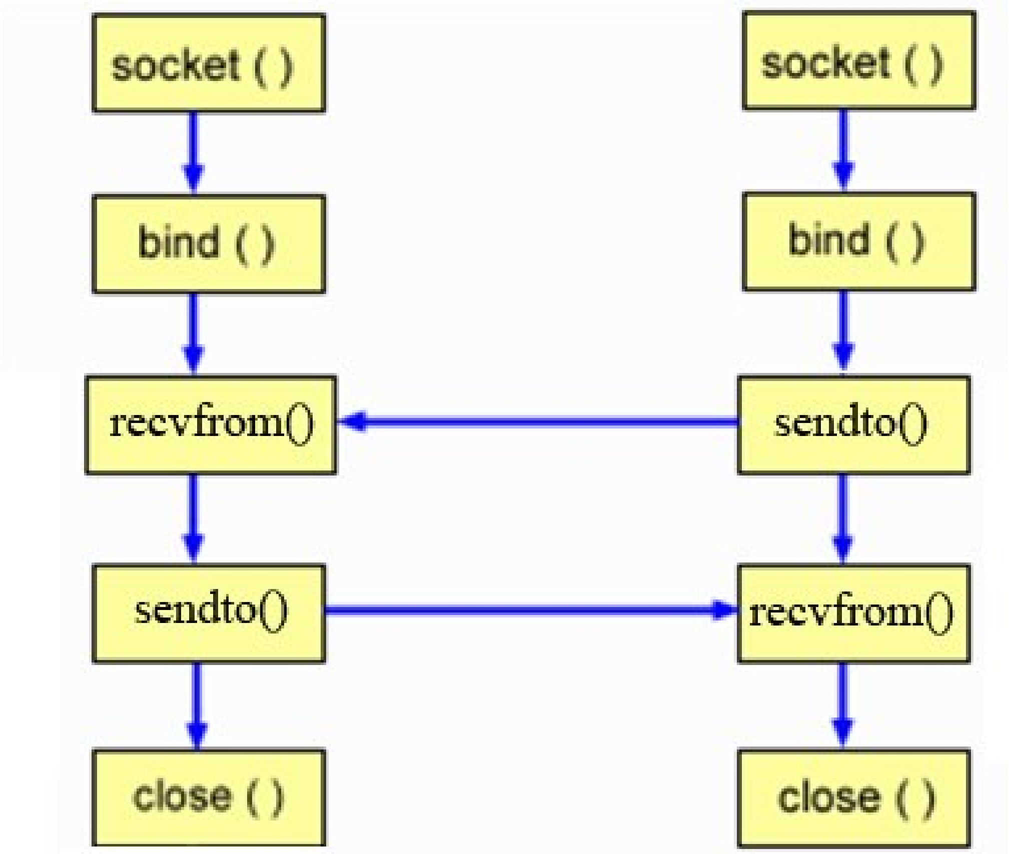
- … let’s remember: iterative TCP client/server
- UDP client/server model
- I/O primitives
- Advanced programming aspects in Internet
- *socket* API – discussions and critics

TCP server/client Model



Server

UDP Client/Server Model



Server UDP

Client UDP

UDP Client/Server model

- For **socket()** it is used **SOCK_DGRAM**
- listen(), accept(), connect() are not usually used
- For datagrams sending it can be used **sendto()** or **send()**
- For datagrams reading it can be used **recvfrom()** or **recv()**
- Nobody guarantees that the sent data have reached the addressee or is not duplicate

UDP Client/Server model

- UDP sockets can be “connected”: the client can use connect() to specify the server address (IP, port) – **pseudo-connections**:
  - Utility: sending several datagrams to the same server, without specifying server address for each datagram
  - For UDP, connect() will retain only the information about the endpoint without getting initiate any data exchange
  - Although connect() reports success does not mean that the address is a valid point or the terminal server is available

UDP Client/Server model

- UDP Pseudo-connections
  - shutdown() can be used to stop transmitting data in one direction, but no message will be sent to the conversation partner
  - close() can be called to remove a pseudo-connection

I/O primitives

```
#include <sys/types.h>
#include <sys/socket.h>
int send (int sockfd, char *buff, int nbytes, int  flags);
int recv (int sockfd, char *buff, int nbytes, int flags);
```

- They can be used in the connection-oriented communications or pseudo-connections
- send() and recv() assume that a previous connect() call was performed
- The first 3 arguments argumente sunt similare cu cele de la write(), respectiv read()
- The fourth argument is usually 0, but can have other values that specify conditions for the call
- Both calls return at normal execution, the transfer length (in bytes)

I/O primitives

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto ( int sockfd, char *buff, int nbytes, int flags,
                 struct sockaddr *to, int addrlen);
int recvfrom (int sockfd, char *buff, int nbytes, int flags,
                   struct sockaddr *from, int *addrlen);
```

- Used for connectionless communications
- At sendto() and recvfrom() the elemnts to identify the remote node is specified in the last two arguments
- Both calls return, in normal execution, the transfer length in bytes

I/O primitives

```
#include <sys/uio.h>
ssize_t readv (int fd, const struct iovec *iov, int iovcnt);
ssize_t writev (int fd, const struct iovec *iov, int iovcnt);
```

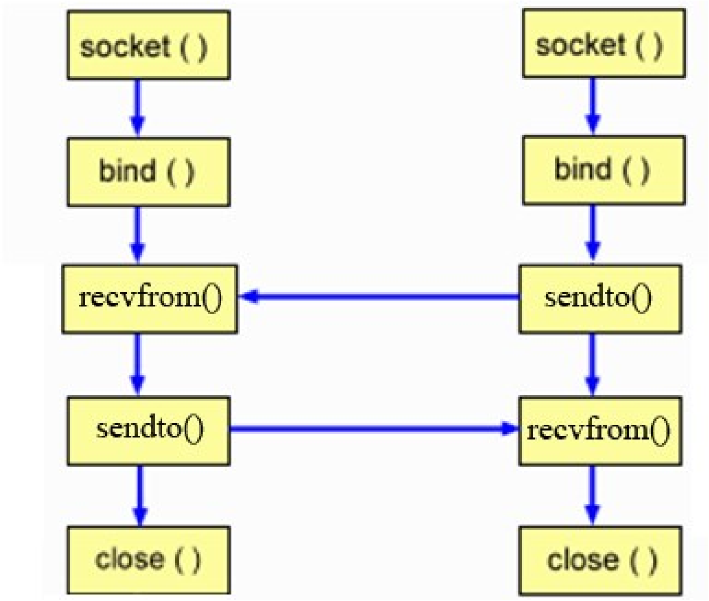
– Wider than read()/write(), provides the ability to work with data in non-contiguous memory areas

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t  recvmsg (int s, struct msghdr *msg, int flags);
ssize_t  sendmsg (int s, const struct msghdr *msg, int flags);
```

– Receives / transmits messages extract them from the *msghdr* structure

DEMO

UDP Client/Server model - Example



Server UDP

Client UDP

Primitives

- **getpeername()** – returns information about the other end of the connection

```
#include <sys/socket.h>
int getpeername (int sockfd, struct sockaddr *addr,
                            socklen_t *addrlen);
```

- **getsockname()** – return informations over the specified socket (local) –> (address to which is attached)

```
#include <sys/socket.h>
#include <sys/types.h>
int getsockname( int sockfd, struct sockaddr * addr,
                               socklen_t * addrlen);
```

Advanced network programming

- Options attached to sockets
  - **getsockopt()** and **setsockopt()**
- I/O Multiplexing

Primitives | Options

- Options attached to *sockets*
  - Attributes used for consulting or changing behavior, general or specific protocol for certain (types of) sockets
  - Type of values:
    - Boolean (*flags*)
    - Complex types:

      int, timeval, in_addr, sock_addr, etc

Primitives|Options

- **getsockopt()** – options consultations

```
#include <sys/types.h>
#include <sys/socket.h>
int getsockopt (int sockfd, int level, int optname, void *optval,
                socklen_t *optlen);
```

Name, value, option length

***Level* -** indicate if option is general or specific to a protocol

Example:

```
len = sizeof (optval);
getsockopt (sockfd, SOL_SOCKET, SO_REUSEADDR, &optval, &len);
```

Primitives|Options

- **setsockopt()** – setting options

```
#include <sys/types.h>
#include <sys/socket.h>
int setsockopt (int sockfd, int level, int optname, void *optval,
                    socklen_t *optlen);
```

Name, value, option length

Return:

- 0 = success
- -1 = error: EBADF, ENOTSOCK, ENOPROTOOPT,...

Primitives|Options

General options:

- Independent of protocol
- Some options are supported only by certain types of sockets (SOCK_DGRAM, SOCK_STREAM)
  - **SO_BROADCAST**
  - **SO_ERROR**
  - **SO_KEEPALIVE**
  - **SO_LINGER**
  - **SO_RCVBUF, SO_SNDBUF**
  - **SO_REUSEADDR**
  - **SO_OOBINLINE**
  - …

[http://www.beej.us/guide/bgnet/output/html/multipage/setsockoptman.html]

Primitives|Options

- **SO_BROADCAST** (boolean)
  - Enable / disable sending data in broadcast mode
  - Used only for SOCK_DGRAM
  - Prevents not to send improperly broadcast

- **SO_ERROR** (int)
  - Show error occurred (similar to errno)
  - Can be used with getsockopt()

- **SO_KEEPALIVE** (boolean)
  - Used for SOCK_STREAM
  - A probe data will be send to the other endoint if no data has been exchange for a long time
  - Used by TCP (e.g., telnet): allows processes to determine whether the corresponding process/host has failed

Primitives|Options

- **SO_LINGER** (struct linger)
  - Controls whether and how long a call after a close will wait for confirmations (ACKs) from the terminal point
  - Used only for connection-oriented sockets to ensure that a call close () will not return immediately
  - Values will be like:

```
struct linger {
   int l_onoff;      /* interpreted as  boolean */
   int l_linger;     /* time in seconds*/
}
```

  - **l_onoff = 0:** *close*() returns immediately, but unsent data is transmitted
  - **l_onoff !=0 and l_linger=0:** *close*() returns immediately and any unsent data are deleted
  - **l_onoff!=0** and **l_linger !=0:** *close*() does not return until the unsent data is transmitted (or the connection is closed by the remote system)

Primitives|Options

- **SO_LINGER – Example**

```
int result;
struct linger lin;
lin.l_onoff=1 ;
lin.l_linger=1;
result= setsockopt( sockfd,
                   SOL_SOCKET,
                   SO_LINGER,
                   &lin, sizeof(lin));
```

Primitives|Options

- **SO_RCVBUF/SO_SNDBUF** (int)
  - Change the size of buffers for receiving or sending data
  - Used for SOCK_DGRAM si SOCK_STREAM

- **Example**:

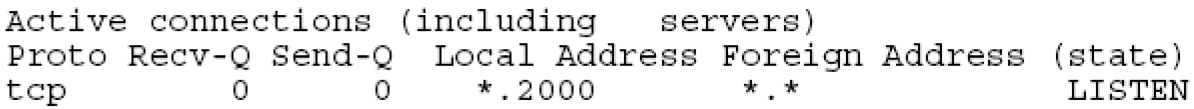
  int result; int buffsize = 10000;

  result= **setsockopt** (s, SOL_SOCKET, SO_SNDBUF, &buffsize, sizeof(buffsize));

Primitives|Options

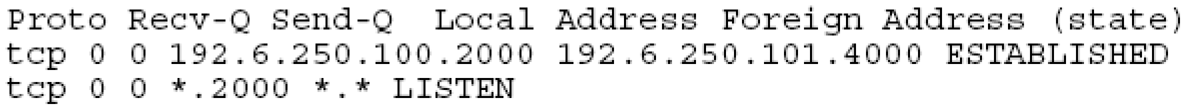
- **SO_REUSEADDR** – (boolean)
  - Allowing connection to an adress already in use
    - ➢ the unique binding rule is not violated
  - Used in a case in which a *passive socket* can use a port already in use

Stare 1



```
Active connections (including   servers)
Proto Recv-Q Send-Q  Local Address Foreign Address (state)
tcp          0       0     *.2000        *.*              LISTEN
```

Stare 2



```
Proto Recv-Q Send-Q  Local Address Foreign Address (state)
tcp 0 0 192.6.250.100.2000 192.6.250.101.4000 ESTABLISHED
tcp 0 0 *.2000 *.* LISTEN
```

- If the listening *daemon* at 2000 port is *killed*, restarting the demon will fail if SO_REUSEADDR is not set

Example

int optval = 1;

setsockopt (sockfd, SOL_SOCKET, **SO_REUSEADDR**, &optval, sizeof(optval));

bind (sockfd, &sin, sizeof(sin));

Primitives|Options

Specific options for IP protocol

- **IP_TOS** allows to set "Type Of Service" field (e.g., ICMP) from the IP header
- **IP_TTL** allows to set "Time To Live" field from the IP header

There are options for IPv6.(RFC 2460,2462)

-**IPV6_V6ONLY, …**

Primitives|Options

Specific options for TCP protocol

- **TCP_KEEPALIVE** set waiting time if SO_KEEPALIVE is activated
- **TCP_MAXSEG** sets the maximum length of a segment (not all implementations allow change this value by the application)
- **TCP_NODELAY** disabling the Nagle algorithm (reducing the number of small packets in a network WAN, TCP always sends packets of maximum size, if possible) - used to generate small packets (e.g., interactive clients such as *telnet*)

I/O Multiplexing

- The opportunity to monitor more I/O descriptors
  - A generic TCP client (e.g., telnet)
  - An interactive client (e.g., *ftp, scp, Web browser* ...)
  - A server that can handle multiple protocols (TCP and UDP) simultaneously
  - Solving unexpected situations (i.e. fall in the middle of communication)
- Example: data read from the standard input must be written to a socket, and the data received through the network should be displayed to *stdout*

I/O Multiplexing | Solutions

- Using non-blocking mechanism using primitives: **fnctl()** / **ioctl()**
- Using asynchronous mechanism
- Using **alarm()** to interrupt slow system calls
- Use of *processes/threads* (*multitasking*)
- Using primitives that allows checking from multiple inputs: **select()** and **poll()**

I/O Multiplexing | Solutions

- Using non-blocking mechanism using **fnctl()**
  - Set I/O calls as a no-blocking

```
int flags;
flags = fcntl ( sd, F_GETFL, 0 );
fcntl( sd, F_SETFL, flags | O_NONBLOCK);
```

  - If no data are available, read() will return -1 or if there is insufficient space in the buffer write() will return -1 (with the error EAGAIN)

I/O Multiplexing | Solutions

Using non-blocking mechanism using **ioctl()**

#include <sys/ioctl.h>

ioctl (sd, **FIOSNBIO**, &**arg**);

-arg is a pointer to an int
-If int is 0, the socket is set in blocking mode
-If int is 1, the socket is set to non-blocking mode

If the socket is in non-blocking mode, we have:

- accept() – if there is no request, *accept*() returns with the error EWOULDBLOCK
- connect() – if the connection can not be established immediately, *connect*() returns with the error EINPROGRESS
- recv() – if no data is received, *recv*() returns -1 with the error EWOULDBLOCK
- send() – if there is no buffer space for data to be transmitted, *send*() returns -1 with the error EWOULDBLOCK

I/O Multiplexing | Solutions

Sending and receiving data asynchronously

- **Problem**: Given that sockets are created by default in blocking mode (I/O), how a process can be notified when "something" happens to a socket?
- **asynchronous sockets** allows sending a signal (SIGIO) to the process
- SIGIO signal generation is dependent on protocol

I/O Multiplexing | Solutions

Sending and receiving data asynchronously

- For TCP SIGIO signal can occur when:
  - The connection has been fully established
  - A disconnect request was initiated
  - A disconnect request is completed
  - *shutdown()* is called for one communication sense
  - Data from the corresponding endpoint appear
  - Data were send
  - Error

I/O Multiplexing | Solutions

Sending and receiving data asynchronously

- For UDP SIGIO signal occurs when:
    - It receives a datagram
    - …
- We allow processes to carry out other activities and monitor UDP transfers

I/O Multiplexing | Solutions

Sending and receiving data asynchronously

- **Implementation**
  - *Socket* must be set as asynchronous

```
#include <sys/unistd.h>
#include <sys/fcntl.h>
int fcntl (int s, int cmd, long arg)
```

Example:

```
int sd = socket(PF_INET, SOCK_STREAM, 0);
fcntl (sd, F_SETFL, O_ASYNC);
```

I/O Multiplexing | Solutions

- Alarms use

```
while(…){
    signal (SIGALRM, alarmHandler);
    alarm (MAX_TIME);
    read (0,…);

    signal (SIGALRM, alarmHandler);
    alarm (MAX_TIME);
    read (tcpsock,…);...
}
```

Function written by a programmer

I/O Multiplexing and/or *Multitasking*

Concurrent Servers – per-client process

Pre-forked Concurrent Servers

- It creates a number of child processes immediately at initialization and every process freely interact with a specific client

Pre-threaded Concurrent Servers

- There are used *threads* instead of processes (see POSIX *threads* – pthread.h)
- Example: Apache server

Problems:

- The number of clients greater than the number of processes / threads
- Number of processes / threads too large in relation to clients number
- *OS overhead*

..... *(future course)*

I/O Multiplexing | solutions

Problems that arise:

- Using **non-blocking calls,** the processor is intensively used
- For **alarm(),** which is the optimal value **MAX_TIME?**

I/O Multiplexing |select()

- Allows use of blocked calls for descriptors (files, pipes, sockets,…)
- Suspend the program until descriptors from the manageed list are ready for I/O operations

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
  int select (int nfds,
              fd_set *readfds,
              fd_set *writefds,
              fd_set *exceptfds,
              struct timeval *timeout);
```

The maximum value of descript. plus 1

The set of descriptors for reading, writing, exception

Waiting time

I/O Multiplexing |select()

Handling the descriptors set (**fd_set** type) is performed using macros:

FD_ZERO (fd_set *set);	Delete the descritors set.
FD_SET (int fd, fd_set *set);	Add the fd descriptor in the set.
FD_CLR (int fd, fd_set *set);	Delete the fd descriptor in the set.
FD_ISSET(int fd, fd_set *set);	Test if the fd descriptor belongs to the set.

I/O Multiplexing |select()

For waiting time the structure defined in **sys/time.h** is used**:**

```
struct  timeval {
   long  tv_sec;/* secunde*/
   long  tv_usec;/* microsecunde*/
   }
```

- If timeout is NULL, select() will return immediately
- If timeout is !=0 specify the timeframe in which select() will wait

I/O Multiplexing |select()

A socket descriptor is ready for reading if :

- There are bytes received in the input *buffer* (read() will return >0)
- A TCP connection received a FIN bit(read() return 0)
- *The Socket* is a *listening socket* and there are some connection requests (accept() can be used)
- An error occurred on the *socket* (read() returns –1, with errno set) – errors can be filtered via getsockopt() using SO_ERROR

I/O Multiplexing |select()

A *socket* descriptor is ready for writing if:

- There are a number of bytes available in the writing buffer (write() will return a value > 0)
- The connection is closed in the sense of writing (attempt to write() will generate SIGPIPE)
- A writing error occurred (write() return –1, with errno set) – errors can be filtered via getsockopt() with the SO_ERROR option

I/O Multiplexing |select()

- A socket descriptor is in an exception state if:
  - There are out-of-band data or socket is marked as out-of-band (future course ☺)
  - If the *remote endpoint* has been closed while there were data on the channel, the read/write operation will return ECONNRESET

I/O Multiplexing |select()

select() may return

- The number of descriptors which are in read, write or exceptioon state
- 0 – the time has elapsed, no descriptor is ready
- –1 on error

The use of **select()** – general steps:

- **fd_set** declaration
- Initialization with **FD_ZERO()**
- Adding using **FD_SET()** of each descriptor intended to be monitored
- Calling **select()** primitive
- Upon returning successfully, **FD_ISSET()** is used for descriptors checking

Demo

select() use - Example

BSD Sockets |use

- Internet Services (services use sockets for communication among remote hosts)
  - Example of distributed applications
    - World Wide Web
    - Remote access to a database
    - Distribution of *tasks* on multiple *host*s
    - On-line games
    - …

BSD Sockets | Critics

The API based on BSD sockets has a number of limitations:

- It has a high complexity, because it was designed to support multiple protocols family (but rarely used in practice)
- No portability (some calls/types has different names/representations on other platforms; filenames - *antet.h* depend on system)
- Example: in WinSock the descriptors are pointers, in Unix we are using Int

Summary

- … let’s remember: iterative TCP client/server
- UDP client/server model
- I/O primitives
- Advanced programming aspects in Internet
- *socket* API – discussions and critics

Questions?