Network programming(I)

Lenuta Alboaie
adria@info.uaic.ro

Content

- Client/server paradigm
- API for network programming
- BSD *Socket*
  - Characteristics
  - Primitives
- TCP client/server model

Network Communication paradigms

- Client/Server model
- Remote Procedure Call (RPC)
- Peer-to-Peer (P2P) mechanism – point-to-point communication

Client/Server model

[The first Web Server]

- **Server** Process
  - Provides network services
  - Accept requests from a client process
  - Performs a specific service and returns the result
- **Client** Process
  - Initializes communication with the server
  - Requests a service and expects the server`s response

Client/Server model

- Interaction alternatives:
  - **Connection-oriented** – based on TCP
  - **Connectionless** – based on UDP

Client/Server model

- Implementation:
  - **iterative** – each client is treated at a time, sequentially

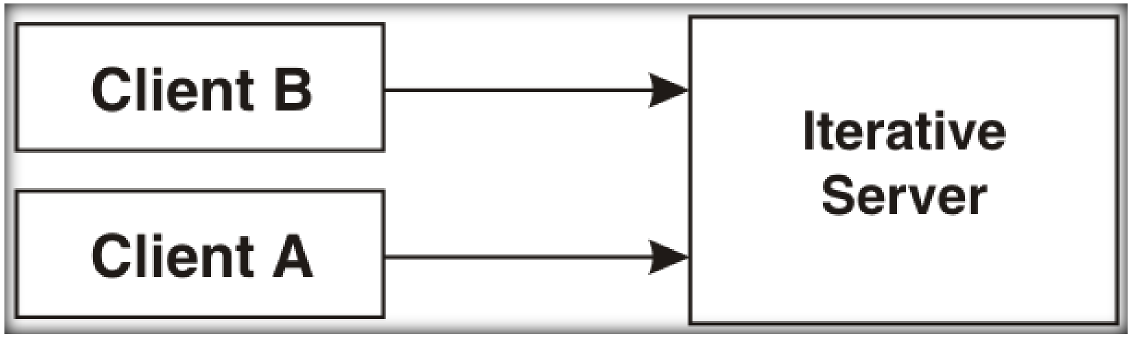


Figure: Example of iterative server

[http://publib.boulder.ibm.com]

Client/Server model

- Implementation:
  - **Concurrency** – the requests are processed concurrently
    - A child process for each request
    - Multiplexing
    - Combination techniques

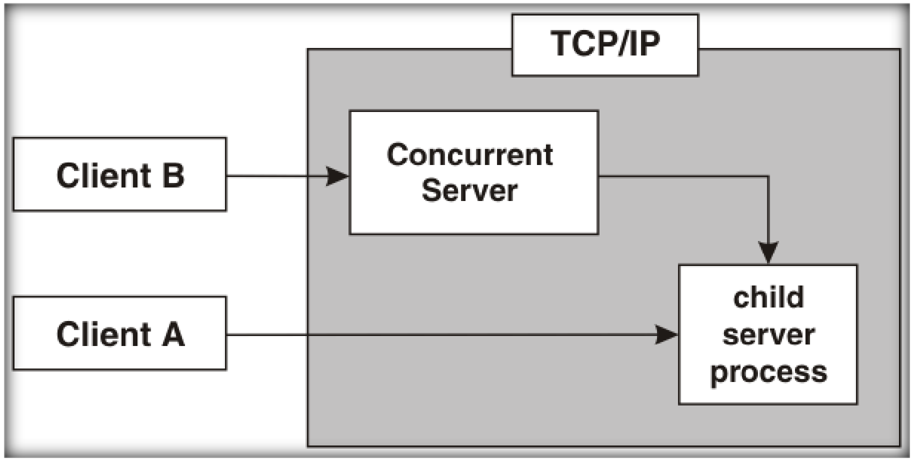


Figure: Concurrent server example

[http://publib.boulder.ibm.com]

API for network programming

- Necessity:
  - Generic interface for programming
  - Hardware and operating system independence
  - Support for different communication protocols
  - Support for connection-oriented communications or for connectionless communications
  - Independence in address representation
  - Compatibility with the common I/O services

API for network programming

- For programming Internet application multiple APIs can be use:
  - ***BSD Sockets***(*Berkeley System Distribution*)
  - TLI (*Transport Layer Interface*) – AT&T, XTI (Solaris)
  - Winsock
  - MacTCP
- Functions offered:

  specifying local and remote endpoints, initiating and accepting connections, sending and receiving data, end connection, error treatments
- TCP/IP does not include an API definition

…

Application programming interface based on

BSD sockets

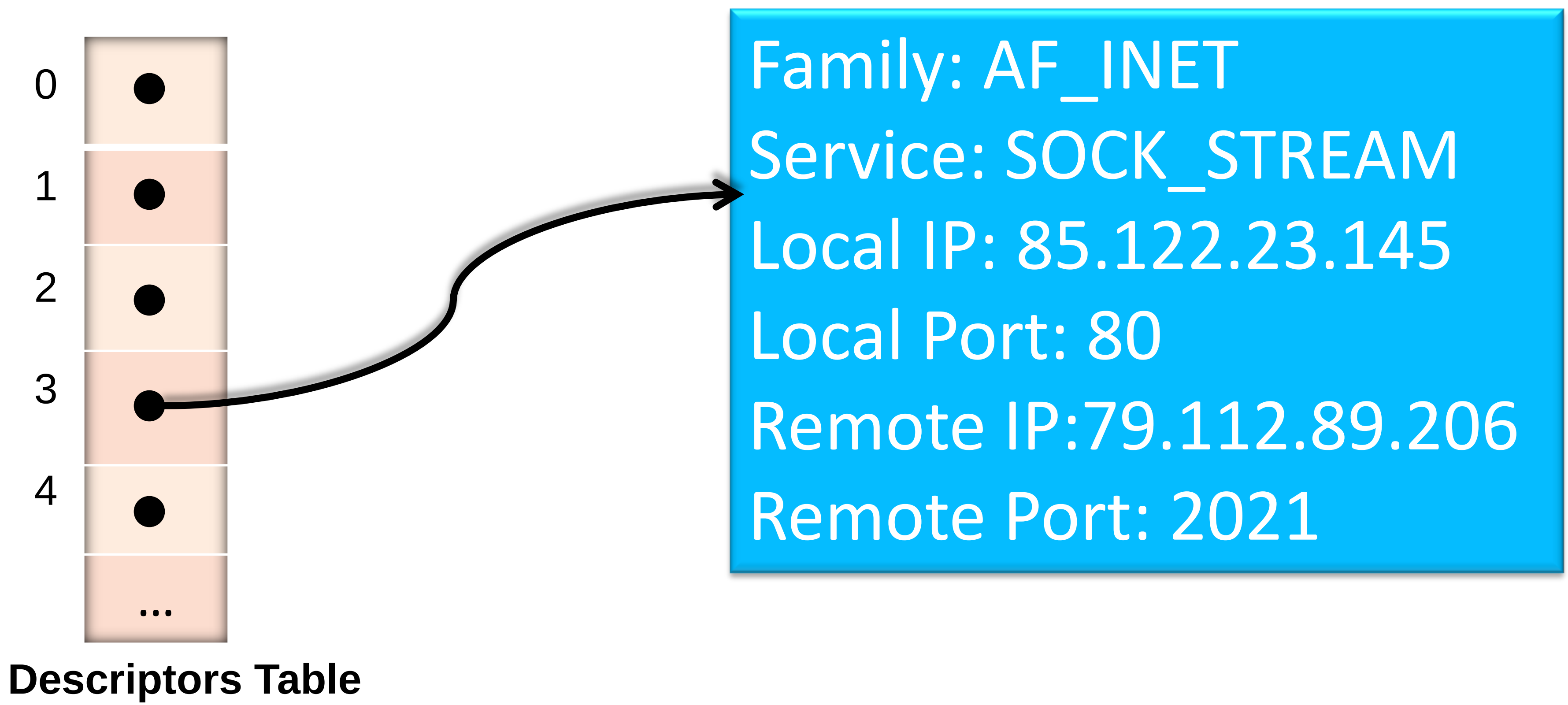
Socket

- General facility, independent of hardware architecture, protocol and type of data transmission for communication among processes on different network machines
- Offers support to multiple family protocols:
  – UNIX domain protocol – used for local communications
  – Internet domain protocol using TCP / IP
  – Other: XNS Xerox,…
- Abstraction of an end-point at the transport level

Socket

- Uses the existing I/O programming interface (similar to files, *pipes*, *FIFOs* etc.)
- May be associated with one/multiple processes from a communication domain
- It provides an API for network programming, having multiple implementations
- From the point of view of the programmer, a socket is similar to a file descriptor; the differences occur when sockets are created or when you set control options for a socket

Socket


0
1
2
3
4
...
Descriptors Table
Family: AF_INET
Service: SOCK_STREAM
Local IP: 85.122.23.145
Local Port: 80
Remote IP:79.112.89.206
Remote Port: 2021

Application programming interface based on BSD sockets

Basic primitives:

- **socket()** – creates a new connection end-point
- **bind() -** attaches a local address to a *socket*
- **listen() –** allows a *socket* to accept connections
- **accept() –** blocks the caller until a connection request appears (used by TCP server)
- **connect()** attempt (active) to establish the connection (used by TCP client)
- **send()** sending data via *socket*
- **recv()** receiving data via *socket*
- **close()** releases the connection(close *socket*)
- **shutdown()** closes a *socket* in one direction

Application programming interface based on BSD sockets

Other primitives:

- Data read
  - read() / readv() / recvfrom() / recvmsg()
- Data sent
  - write() / writev() / sendto() / sendmsg()
- I/O Multiplexing
  - select()
- Managing connection
  - fcntl() / ioctl() / setsockopt() / getsockopt() / getsockname() / getpeername()

Socket|Creation

socket() call system

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

Communication domain: **AF_UNIX, AF_INET, AF_INET6, …**

The protocol used for transmission (Usually: **0** for the transport)

Socket type (ways to accomplish the communication): **SOCK_STREAM, SOCK_DGRAM, SOCK_RAW**

Socket|Creation

socket() system call

Return value

– Success: the *socket* descriptor

– Error: -1

- Error reporting is done via **errno** variable
  - **EACCES**
  - **EAFNOSUPPORT**
  - **ENFILE**
  - **ENOBUFS sau ENOMEM**
  - **EPROTONOSUPPORT**
  - **…**

Constants defined in **errno.h**

Socket-uri

Example of possible combinations for the socket()'s arguments:

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

Domeniu	Tip	Protocol
AF_INET	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IP
AF_INET6	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IPv6
AF_LOCAL	SOCK_STREAM	Internal communication mechanism
	SOCK_DGRAM	

Note: AF_LOCAL=AF_UNIX (for historical reasons)

Sockets

Observations

- *socket()* allocates resources for a communication end-point, but it doesn't state which is the addressing mechanism
- *Sockets* provide a generic addressing mechanism; for TCP/IP , it must be specified (*IP address, port*)
- Other protocols suite may use other addressing mechanisms

POSIX types:

int8_t, uint8_t, int16_t, uint16_t, int32_t, int32_t, u_char, u_short, u_int, u_long

Sockets

- POSIX types used by sockets:
  - sa_family_t –address family
  - socklen_t – structure lenght
  - in_addr_t –IP adress (v4)
  - in_port_t – port number
- Specifying generic addresses

```
struct sockaddr {
      sa_family_t  sa_family;
      char sa_data[14]
  }
```

Address family: AF_INET, AF_ISO,…

14 bytes – used address

Sockets

- For IPv4 AF_INET a special structure is used: sockaddr_in

```
struct sockaddr_in {
      short int sin_family;
      unisgned short int sin_por
      struct in_addr sin_addr;
      unsigned char sin_zero[8];
}

struct in_addr{
      in_addr_t s_addr;
}
```

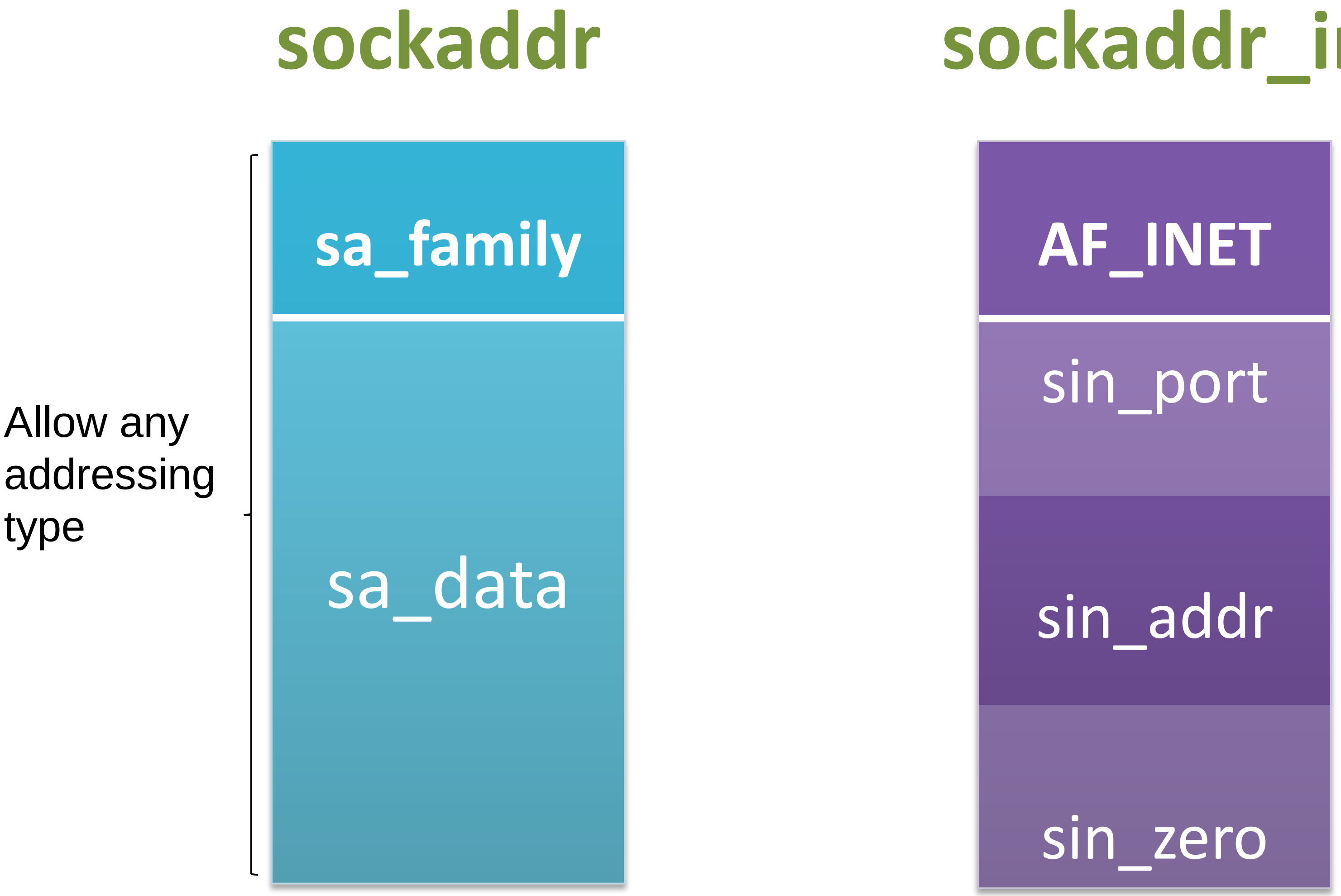
Address family: AF_INET

Port (2 octets)

Unused Bytes

4 bytes for IP address

Sockets


sockaddr
sockaddr_in
sa_family
sa_data
Allow any addressing type
AF_INET
sin_port
sin_addr
sin_zero

Sockets

- The values from sokaddr_in are stored in respect to *network byte order*
- Conversion functions (netinet/in.h)
  - **uint16_t htons (uint16_t)** – converting a short integer (2 bytes) from host to network;
  - **uint16_t ntohs (uint16_t)**;
  - **uint32_t ntohl (uint32_t)** – converting a long integer (4 bytes) from network to host;
  - **uint32_t htonl (unit32_t)**;

Discussion| Octets order

Byte order of a word (*word* - two bytes) can be achieved in two ways:

- **Big-Endian** – The most significant byte is first
- **Little-Endian** – The most significant byte is the second

Example:

BigEndian machine sends

(e.g. Motorola processor)

00000000	**00000010**	=2

LittleEndian machine will perform:

(e.g. Intel processor)

00000010	**00000000**	=512

As convention,

network byte order - BigEndian

Sockets

- For IPv6 AF_INET6 sockaddr_in6 structure it is used:

```
struct sockaddr_in6 {
  u_int16_t sin6_family;   /* AF_INET6*/
  u_int16_t sin6_port;
  u_int32_t sin6_flowinfo;
  struct in6_addr sin6_addr;
  u_int32_t  sin6_scope_id;
}

struct in6_addr{
  unsigned char s6_addr[16];
}
```

Sockets

Example:

```
// IPv4:
struct sockaddr_in ip4addr;  int s;
ip4addr.sin_family = AF_INET;
ip4addr.sin_port = htons(2510);
inet_pton(AF_INET, "10.0.0.1", &ip4addr.sin_addr);
s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip4addr, sizeof (ip4addr));

// IPv6:
struct sockaddr_in6 ip6addr; int s;
ip6addr.sin6_family = AF_INET6;
ip6addr.sin6_port = htons(2610);
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &ip6addr.sin6_addr);
s = socket(PF_INET6, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip6addr, sizeof (ip6addr));
```

Convert IPv4 and IPv6 addresses from string (x.x.x.x) in network byte order (#include <arpa/inet.h>)

? (next slide)

Sockets (slide 19)

Observations

- *socket()* allocates resources for a communication end-point, but it doesn't state which is the addressing mechanism
- *Sockets* provide a generic addressing mechanism; for TCP/IP it must be specified (*IP address, port*)
- Other protocols suite may use other addressing mechanism

Sockets| Assigning an address

- Assigning an address to an existing socket is made with **bind()**

```
int bind ( int sockfd,
           const struct sockaddr *addr,
           int addrlen );
```

- It returns: 0 if successful, -1 on error

errno variable will contain the corresponding error code: EACCES , EADDRINUSE, EBADF, EINVAL, ENOTSOCK,…

Sockets| Assigning an address

Example:

```
#define PORT 2021
struct sockaddr_in adresa;
int sd;

sd = socket (AF_INET, SOCK_STREAM, 0))  // TCP
adresa.sin_family = AF_INET; // establish socket family
adresa.sin_addr.s_addr = htonl (IPaddress); //IP address
adresa.sin_port = htons (PORT);  //port

if (bind (sd, (struct sockaddr *) &adresa, size of (adresa) == -1)
       {
              perror ("Eroare la bind().\n");
              …
       }
```

Sockets| Assigning an address

- bind() uses:
  - The server wants to attach a socket to a default port (to provide services via that port)
  - The client wants to attach a socket to a specified port
  - The client asks the operating system to assign any available port
- Normally, the client does not require attachment to a specified port
- Choose any free port:

```
adresa.sin_port = htons(0);
```

Sockets| Assigning an address

- Choosing the IP address - bind()
  - If the host has multiple IP addresses assigned?
  - How to solve platform independence?

To attach a socket to your local IP address, **INADDR_ANY** constant will be used instead

Sockets| Assigning an address

- IP address conversion:

int inet_aton (const char *cp, struct in_addr *inp);

ASCII “x.x.x.x” -> 32 bits internal representation (*network byte order*)

char *inet_ntoa(struct in_addr in);

32 bits representation (*network byte order*)-> ASCII “x.x.x.x”

Obs: **[@fenrir ~]$** man inet_addr

Sockets| Assigning an address

- Observations:
  - ForPv6 INADDR_ANY will be replaced by IN6ADDR_ANY(netinet/in.h):

    serv.sin6_addr = in6addr_any;
  - The conversion function for IPv6 (that can be used for IPv4) are:

    inet_pton()

    inet_ntop()

Sockets| listen()

- Passive interaction:
  - The system core will wait for connection requests directed to the address where the socket is attached

    3-way handshake

  - The received connections will be placed in a queue

    int listen(int sockfd, int backlog);

    The number of connections in the queue

    TCP socket attached to an address

  - It returns: 0 – succes, -1 - error

Sockets| listen()

- Observations:
  - The *backlog* value depends on the application (usually 5)
  - HTTP servers should specify a bigger value for *backlog* (due the multiple requests)

Sockets| accept()

- Accepting the connections from clients
  - When the application is ready to address a new connection, the system will interrogate for another connection

```
int accept (int sockfd,
            struct sockaddr *cliaddr,
            socklen_t *addrlen);
```

Socket TCP (passive mode)

- It must initially be equal to the length of the **cliaddr** structure
- It will return the number of bytes used in **cliaddr**

It returns the socket descriptor corresponding to the client end-point or -1 in an error case

Sockets| connect()

- Trying to establish a connection to the server

3-way handshake

Socket TCP
- It does not require attaching with bind(); the operating system will assign a local address (IP, port)

```
int connect (int sockfd,
             const struct sockaddr *serv_addr,
             socklen_t addrlen);
```

Contains server adress (IP, port)

It returns: success ->0, erorr -> -1

I/O TCP| read()

int read(int sockfd, void *buf, int max);

- The call is usually a blocking one, read() returns only when data are available
- Reading from a TCP socket may return fewer bytes than the maximum number desired
  - We must be prepared to read byte-by-byte at a time (see previous course)
- If the communication partner closes the connection and there are no data to receive, 0 (EOF) is returned
- Errors: EINTR – a signal interrupted the reading, EIO –I/O error, EWOULDBLOCK – the *socket* set in a non-blocking mode, doesn't have data

I/O TCP| write()

int write(int sockfd, const void *buf, int count);

- The call is usually a blocking one
- Errors:
  - EPIPE – write to a offline socket
  - EWOULDBLOCK – normally writes blocks until the writing operation is complete; if the socket is set in non-blocking mode and some problems occur, it returns this error immediately

I/O TCP| Example

```
#define MAXBUF 127  /*  reading buffer length*/
…
char *cerere= “da-mi ceva”;
char buf [MAXBUF]; /* response buffer*/
char *pbuf= buf; /* buffer pointer */
int n, lung = MAXBUF; /* the nr. of bytes read, the nr. of free bytes
in buffer */
…
/* send the request*/
write(sd, cerere, strlen(cerere));

/* wait the response*/
while ((n = read (sd, pbuf, lung)) > 0) {
      pbuf+= n;
      lung -= n;
}
```

Example of communication between client and sever

Closing the connection| close()

int close(int sockfd)

- Effect:
  - closes the connection;
  - Frees the memory associated with the *socket*
    - for processes that share the same socket, it decreases the number of references to that socket; if it reaches 0, than the socket is deleted
- Problems:
  - The server cannot end the connection, it doesn’t know if and when the client sends other demands
  - The client doesn’t know if the data reaches the server

Closing the connection| shutdown()

- Unidirectional closing
  - When a client finishes to send requests, it can call shutdown() to specify that data will be sent no longer (the socket is not deleted)
  - The server will receive EOF and after sending the last answer to the client, it will close the connection

```
#include <sys/socket.h>
int shutdown (int sockfd, int how);
```

0 – future reading from the socket will not be allowed (SHUT_RD);
1 – future writings will not be allowed (SHUT_WR);
2 - read/write operations are no longer allowed (SHUT_RDWR)

Metaphor for Good Relationships

Copyright Dr. Laura's Network Programming Corp.

To succeed in relationships…

- you need to establish your own identity. `bind()`
- you need to be open & accepting. `accept()`
- you need to establish contacts. `connect()`
- you need to take things as they come, not as you expect them. ***read might return 1 byte***
- you need to handle problems as they arise. ***check for errors***

[Retele de calculatoare –
curs 2007-2008, Sabin Buraga]

Client/Server Model

- **TCP iterative server:**
  - Creates *socket* to address clients: socket()
  - Prepares data structures (sockaddr_in)
  - Attaches the socket to the local address (port): bind()
  - Prepares the socket to listen in order to establish connections with clients listen()
  - The expectation of making a connection with a particular client (passive open): accept()
  - Processes customer requests, using the socket returned by accept(): sequence of read()/write() calls
  - Closes (unidirectional close) of the connection: close(), shutdown()

Model client/server

- **TCP client model:**
  - Creates a *socket* to connect to a server: socket()
  - Prepares data structures (sockaddr_in)
  - Attaches the socket: bind() – optional
  - Connects to the server (active open): connect()
  - Service request/receives the results sent by the server: sequence of read()/write() calls
  - Closes (unidirectional close) the connection: close(), shutdown()

General model – TCP server/client

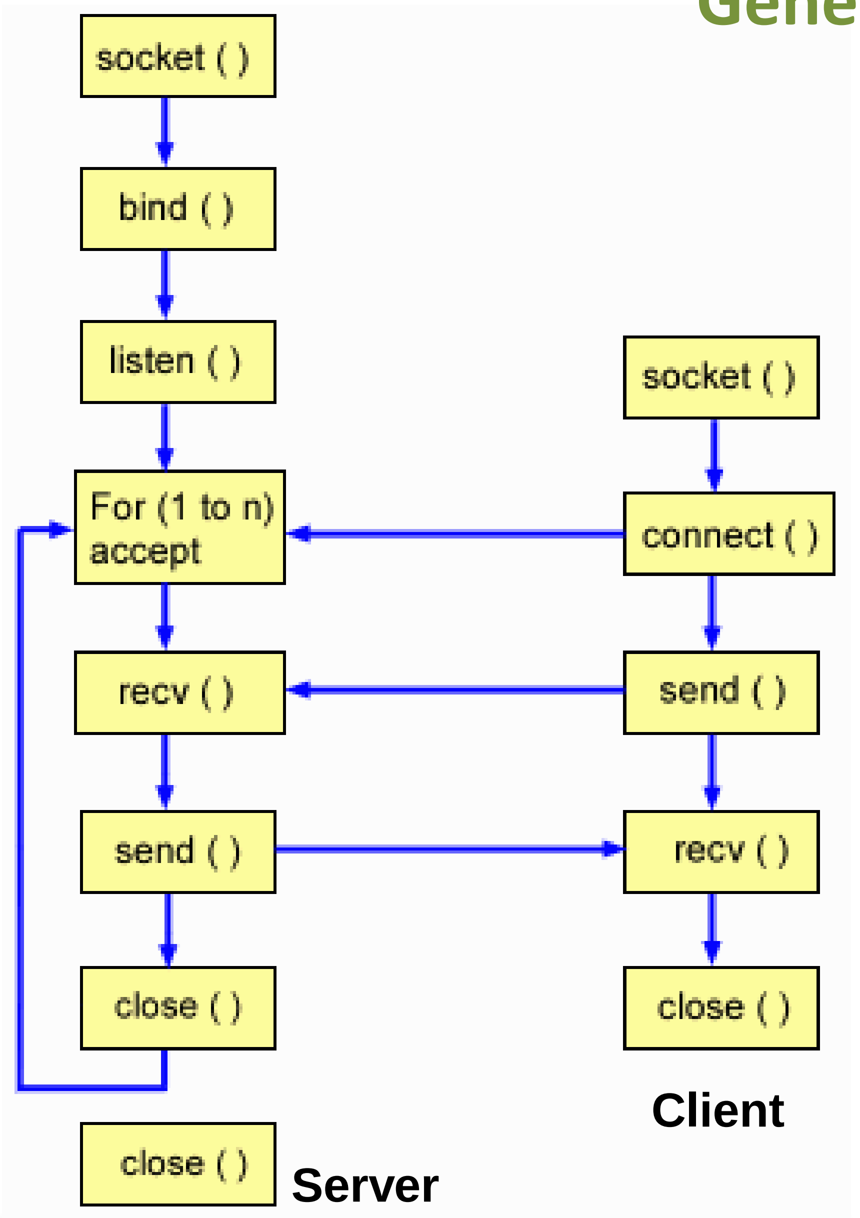
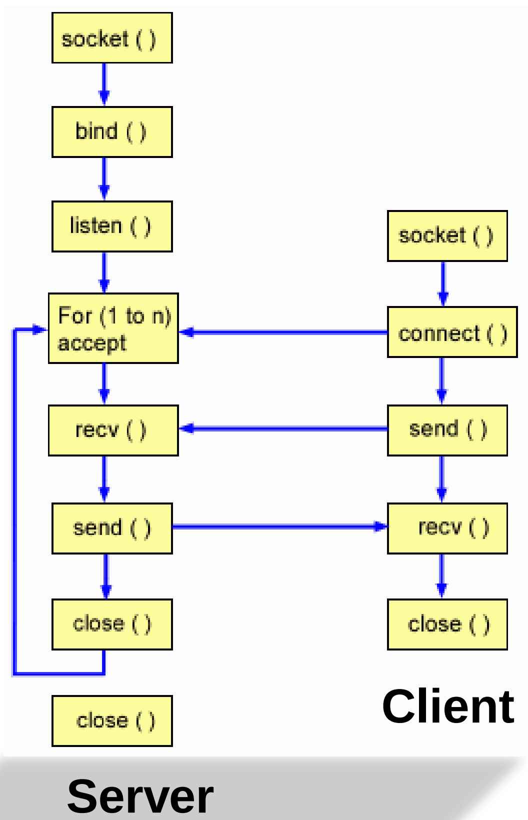


Figure: TCP Iterative Server - sequence of events

[http://publib.boulder.ibm.com]



DEMO ☺

Example of TCP iterative server / client

Summary

- Client/server paradigm
- API for network programming
- BSD *Socket*
  - Characteristics
  - Primitives
- TCP client/server model

Questions?