COMP 8567 Advanced Systems Programming

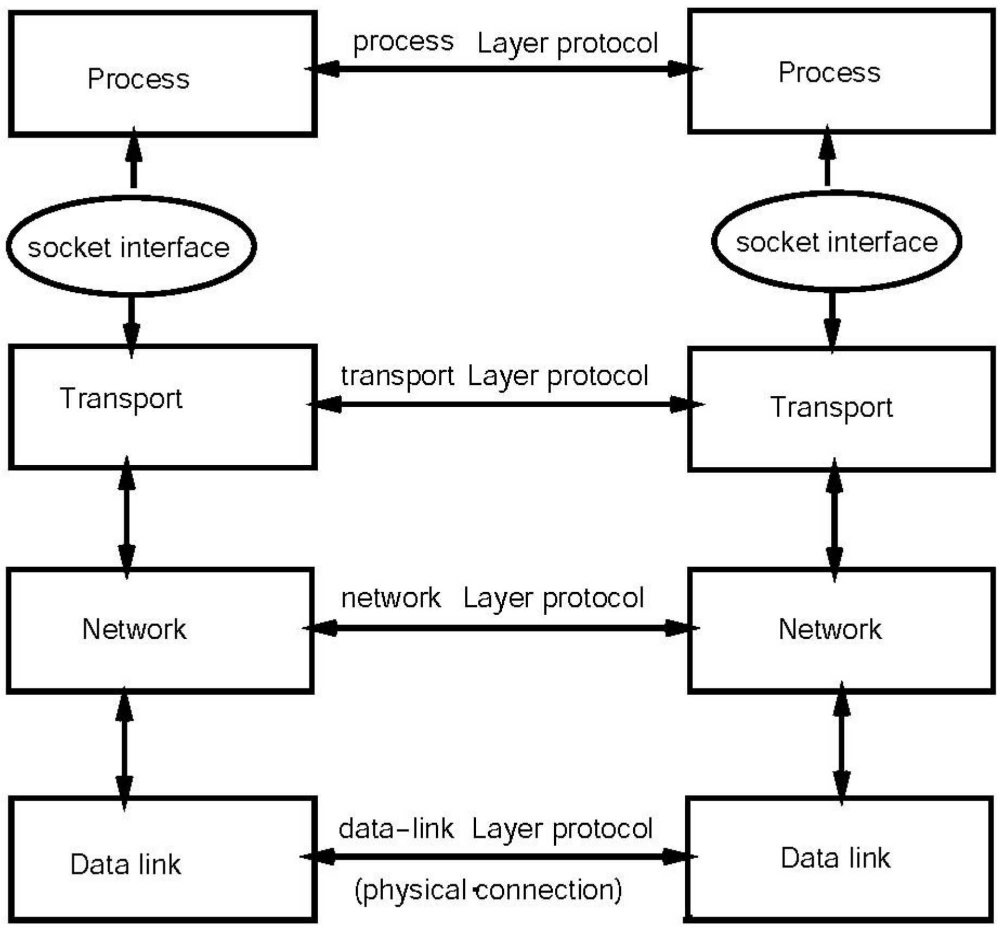
Sockets

Created by Dr. Boubakeur Boufama
Revised by Dr. Prashanth Ranga

Outline

- Inter-process Communication over a network
- Sockets Introduction
- Different Kinds of Sockets
- Socket Address Structure
- Generic Socket Address Structure
- Creating Endpoints for Communication: socket()
- Initiating a connections on a socket: connect()
- Binding a name to a socket: bind()
- Listening for connections on a socket: listen()
- Accepting a connection on a socket: accept()
- Examples of socket programming
- Summary

IPC over a network



Port Number — (Common Ports)	Description
1	TCP Port Service Multiplexer (TCPMUX)
5	Remote Job Entry (RJE)
7	ECHO
18	Message Send Protocol (MSP)
20	FTP — Data
21	FTP — Control
22	SSH Remote Login Protocol
23	Telnet
25	Simple Mail Transfer Protocol (SMTP)
29	MSG ICP
37	Time
42	Host Name Server (Nameserv)
43	WhoIs
49	Login Host Protocol (Login)
53	Domain Name System (DNS)
69	Trivial File Transfer Protocol (TFTP)
70	Gopher Services
79	Finger
80	HTTP
103	X.400 Standard
108	SNA Gateway Access Server
109	POP2
110	POP3
115	Simple File Transfer Protocol (SFTP)
118	SQL Services
119	Newsgroup (NNTP)
137	NetBIOS Name Service
139	NetBIOS Datagram Service
143	Interim Mail Access Protocol (IMAP)
150	NetBIOS Session Service
156	SQL Server
161	SNMP
179	Border Gateway Protocol (BGP)
190	Gateway Access Control Protocol (GACP)
194	Internet Relay Chat (IRC)
197	Directory Location Service (DLS)
389	Lightweight Directory Access Protocol (LDAP)
396	Novell Netware over IP
443	HTTPS
1080	Socks

Sockets

Sockets are the tradional UNIX IPC mechanism that allows local/distant processes to talk to each other.

IPC using sockets is based on the client/server paradigm.
A typical scenario can be described as follows.

- The server process creates a named socket, whose name is known by client processes, and listens on that sockets for requests from clients.
- A client process can talk to the server process by
  - creating an unnamed socket and,
  - requesting it to be connected to the server's named socket
- If succesful, one file descriptor is returned to the client and another one to the server. These file descriptors can be used **for read and write** allowing the server and client to communicate.

Note : Socket connections are bidirectional.

Different kinds of sockets

Three attributes may differentiate between different kinds of sockets:

- The domain : **AF_INET** for internet and **AF_UNIX** for same machine IPC.
  Note that **AF** stands for Address Family.
- The type of communication : **SOCK_STREAM**, reliable byte stream connection(TCP) and, **SOCK_DGRAM**,unreliable connectionless (UDP).
- The protocol : the low-level protocol used for communication. This parameter is usually set to 0 in system calls, which means “use the correct/default protocol”.

Different Types of Socket Addresses <netinet/in.h> //Contains the definition of the IP family

- struct sockaddr_in // IPV4
- struct sockaddr_in6 //IPV6
- struct sockaddr_un //Solaris

- Generic Socket Address
  struct sockaddr;

Socket Address Structure IPV4

```
struct sockaddr_in {
sa_family_t sin_family; /* address family: AF_INET */
in_port_t sin_port; /* port in network byte order */
struct in_addr sin_addr; /* internet address */ };

//port number is 16 bits (64 k port addresses)
//IP address if 32 bits


/* Internet address */

struct in_addr {
uint32_t s_addr; /* address in network byte order */
};
```

Here is a version for **IPv6**:

```
struct sockaddr_in6{
  sa_family_t      sin6_family;   // AF_INET6
  in_port_t        sin6_port;     // port number
  uint32_t         sin6_flowinfo; // IPv6 flow
                                  // information
  struct in6_addr sin6_addr;      // IPv6 address
  uint32_t         sin6_scope_id; // Scope ID
                                  // (new in 2.4)
};

struct in6_addr {
    unsigned char   s6_addr[16]; // IPv6 address
};
```

Unix socket address structure

The structure is called *sockaddr_un*, defined in $<\ sys/un.h\ >$ ($< linux/un.h >$)

Here is a version from **Solaris**:

```
struct sockaddr_un {
sa_family_t sun_family;         // AF_UNIX
char sun_path[108];             // path name
};
```

→ **Generic Socket Address Structure** *Socket address structures* are always passed by address when passed as a parameter.

Because there are several kinds of socket structures, socket functions prototypes take a pointer to the generic socket address structure, which represents any socket address structure parameter.

The generic address structure is called *sockaddr*, defined in < $sys/socket.h$ >
Here is the definition of the structure :

```
struct   sockaddr{
    uint8_t                 sa_len;
    sa_family_t             sin_family;
    char sa_data[14]; // protocol-specific address
};
```

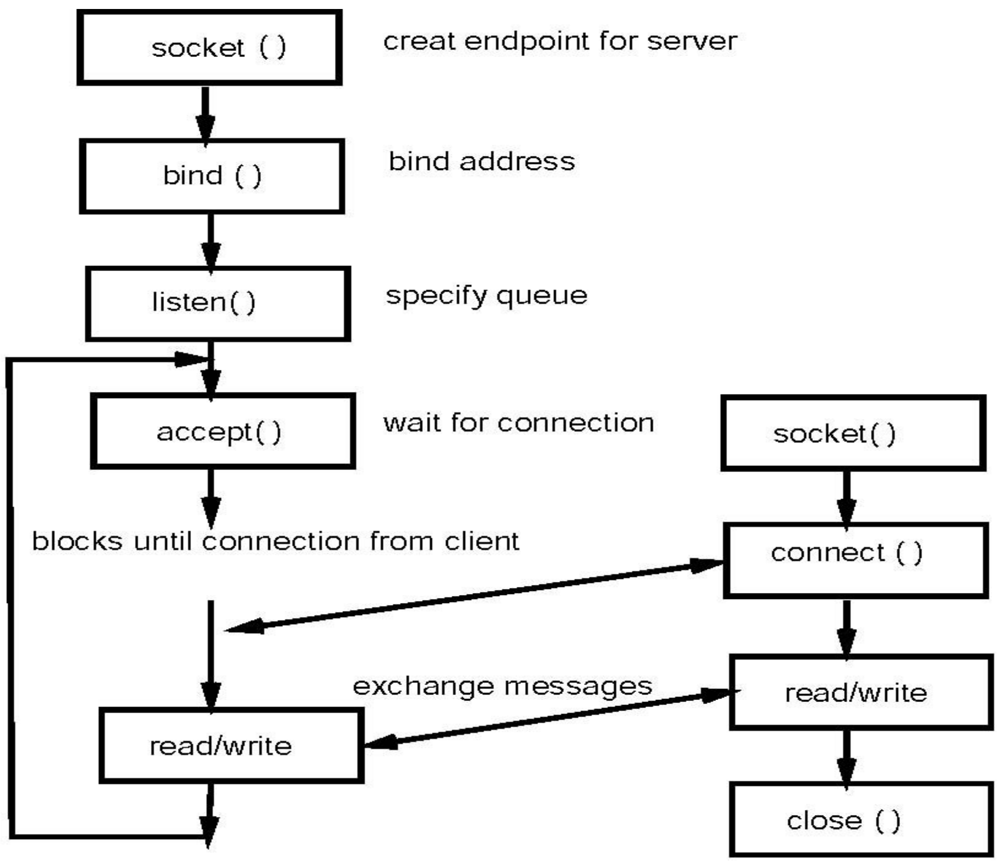
Example : The *bind()* function prototype is
int bind(int, struct sockaddr *, socklen_t);

→ Any call to these functions should **cast the pointer** to the protocol-specific socket address structure to be a **pointer to a generic socket address structure.**
For example :

```
struct sockaddr_in sv; // IPv4 socket, server
            :
bind(sfd, (struct sockaddr*)&sv, sizeof(sv));
```

Socket based client/server IPC

The following figure shows the typical scenario for a connection-oriented communication using sockets.



Creating endpoints for communication : socket()

Synopsis:

int socket(int domain, int type, int protocol); *socket()* creates an endpoint for communication and returns a file descriptor referencing the socket. In case of failure, *socket()* returns -1.

Calling *socket()* is the first thing a process must do in order to perform any network I/O operation.

Example:

sd = socket(AF_INET, SOCK_STREAM, 0);

When a realiable byte-stream connection is requested across the internet.

Note: header files and libraries to be linked are

- Includes : < *sys/types.h* > and < *sys/socket.h* >

Initiating a connection on a socket : connect()

Synopsis:

int connect(int s,struct sockaddr *srv,int len)

Returns 0 when successful and -1 otherwise.

connect() is used by a *TCP* **client** to establish a connection with a *TCP* server.

The parameters have the following meanings:

- **s** is a socket decsriptor that was returned by *socket()*.
- *srv* is a pointer to a socket address structure object, which must contain the IP address and port number of the server
- *len* is the size of the socket address structure.

connect() only returns when a connection is established or when an error occurs.

Binding a name to a socket : bind()

Synopsis:

int bind(int s, struct sockaddr *sp, int len);

Returns 0 when successful and -1 otherwise.

bind() assigns a local protocol address to a socket.

In case of the Internet, the protocol address consists of a 32-bit IPv4 address and a 16-bit port number.

bind() is called by a server to bind their local IP and a port number to a socket.

Listening for connections on a socket : listen()

Synopsis : int listen(int s, int backlog);
Returns 0 when successful and -1 otherwise.

listen() is called only by a **TCP server** to accept connections from client sockets that will issue a *connect()*.
s is a file desciptor of a socket that has been already created.
backlog defines the maximum length the queue of pending connections may grow to.

listen() is normally called after the calls to *socket()* and *bind()*.

Accepting a connection on a socket : accept()

Synopsis:

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);

Returns a file descriptor for a new socket when successful and -1 otherwise.

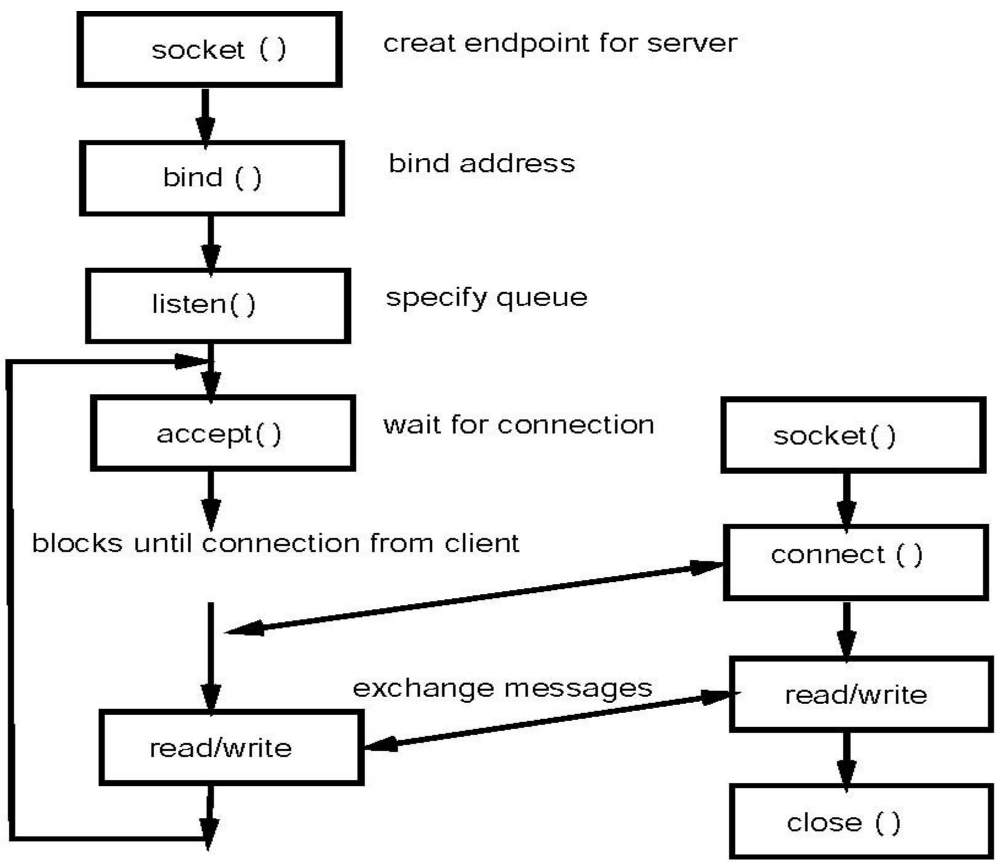
accept() is called by a TCP server to extract the **first connection** in the queue of pending connections, creates a new socket with the properties of s, and allocates a new file descriptor for the newly created socket.

If no pending connections are present on the queue, accept() **blocks the caller** until a connection is present.

Usually, the file descriptor *s* is called the *listening socket* while the returned value is called the *connected socket*.

Socket based client/server IPC

The following figure shows the typical scenario for a connection-oriented communication using sockets.



Read and Write operations performed by a single process (each) in server and client

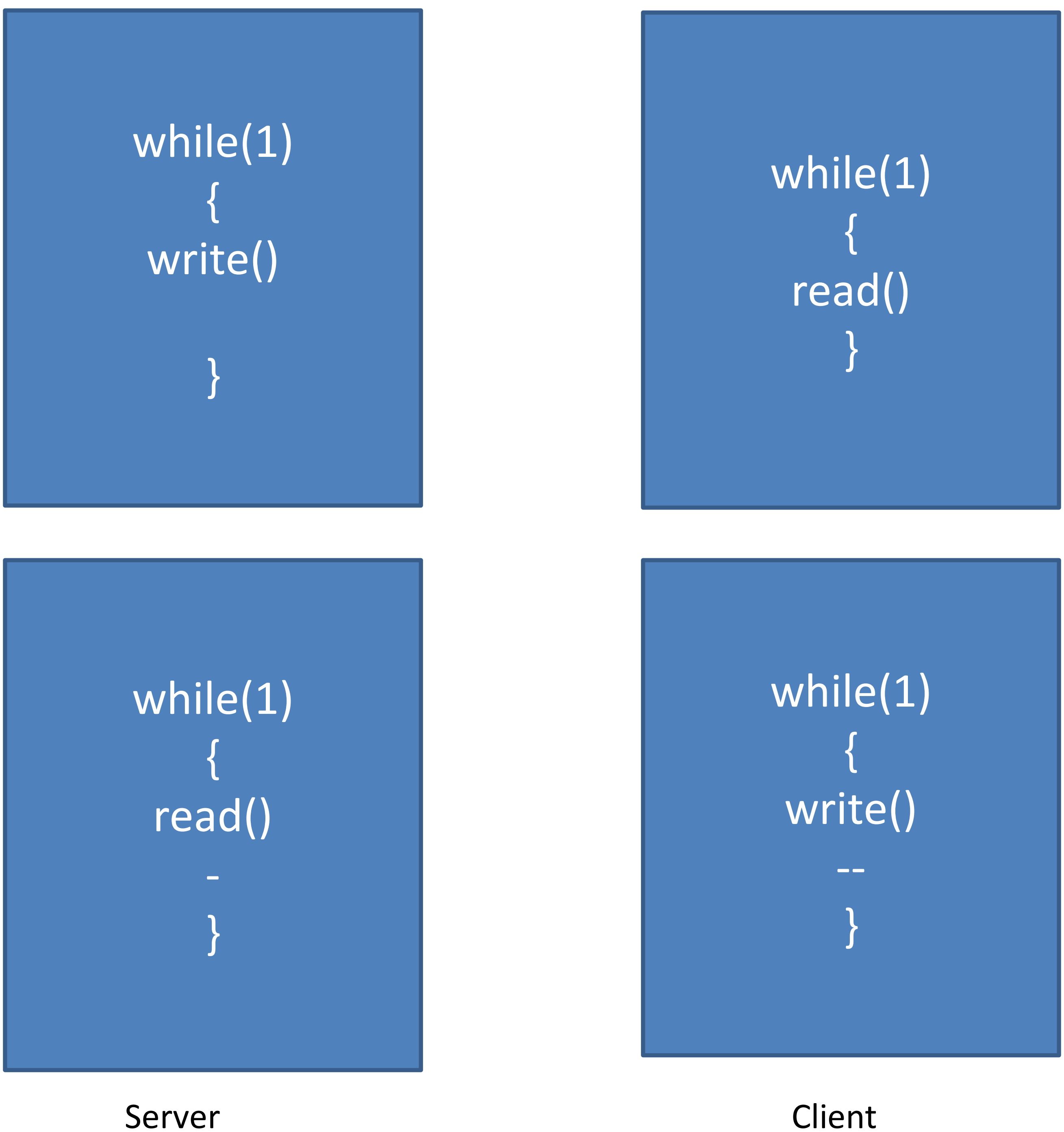
```
while(1)
{
write()
--
---
--
read()
}
```

Server

```
while(1)
{
read()
--
---
--
write()
}
```

Client

Read and Write operations performed by two separate process (each) in server and client



EXAMPLES : IMPLENTATION OF CLIENT/SERVER APPLICATION

Examples //Also available on Blackboard

tcpserver1,tcpclient1, server1,client1,test1,test2

Tcpserver2 tcpclient2 tcpclient3 tcpclient3

A synchronized client-server message exchange (tcpserver2, tcpclient2)

A texting client-server program (tcpserver3/bserver3, tcpclient3)

Summary

- Inter-process Communication over a network
- Sockets Introduction
- Different Kinds of Sockets
- Socket Address Structure
- Generic Socket Address Structure
- Creating Endpoints for Communication: socket()
- Initiating a connections on a socket: connect()
- Binding a name to a socket: bind()
- Listening for connections on a socket: listen()
- Accepting a connection on a socket: accept()
- Examples of socket programming

THANK YOU