COMPUTER ENGINEERING LABORATORY

Luigi Rizzo

luigi.rizzo@unipd.it

October 2024-January 2025

UNIVERSITÀ DEGLI STUDI DI PADOVA

Socket programming

Università degli Studi di Padova

The client-server model

In modern operating systems, the services available on the network are mainly based on the client/server model. This architecture allows systems to share resources and cooperate to achieve an objective through the presence of two categories of subjects, service programs, called servers, which receive requests and provide responses, and user programs, called clients.

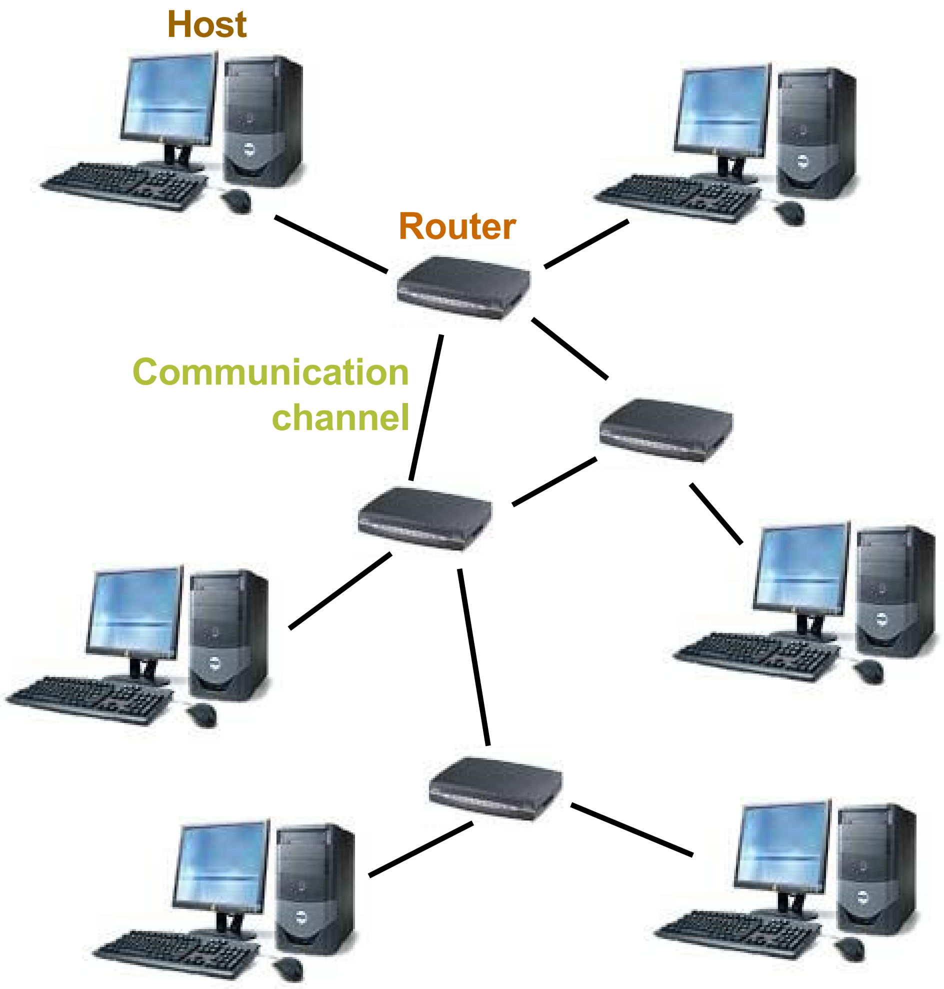
A server is (normally) able to respond to more than one client, so it is possible that many programs can interact simultaneously. What distinguishes the model, however, is that the architecture of the interaction is always in terms of many towards one, the server, who comes to take on a privileged role.

All the fundamental services of the Internet follow this model, such as web pages, e-mail, ftp, telnet, and practically every service that is provided over the network.

The client-server model

- Computer Network
  - hosts, routers, communication channels
- **Hosts** run applications
- **Routers** forward information
- **Packets**: sequence of bytes
  - contain control information
  - e.g. destination host
- **Protocol** is an agreement
  - meaning of packets
  - structure and size of packets

e.g. Hypertext Transfer Protocol (HTTP)

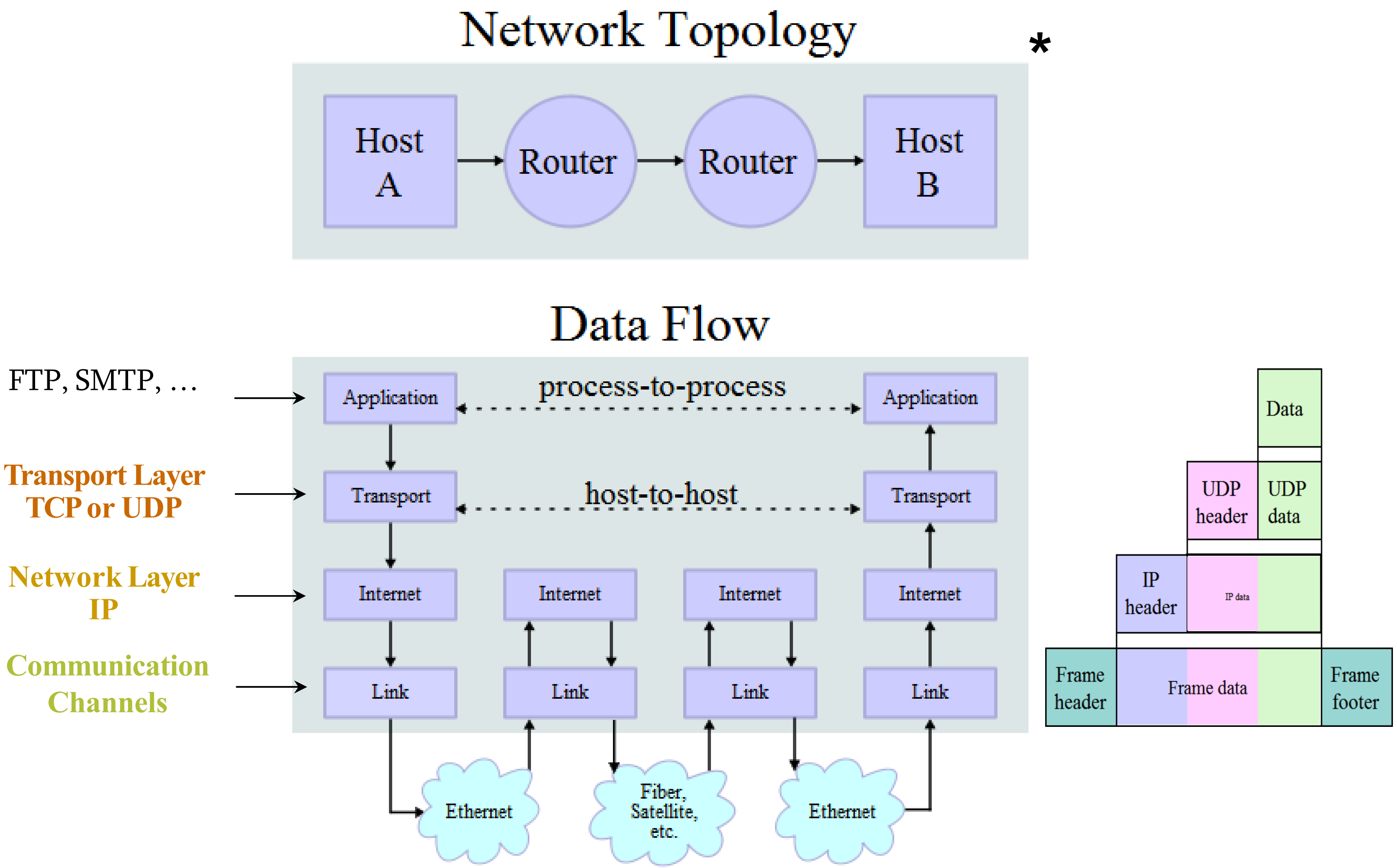


Protocol Families -TCP/IP

- Several protocols for different problems
  - **Protocol Suites** or **Protocol Families:** TCP/IP

- TCP/IP provides end-to-end connectivity specifying how data should be
  - formatted,
  - addressed,
  - transmitted,
  - routed, and
  - received at the destination
- can be used in the internet and in stand-alone private networks
- it is organized into **layers**

Protocol Families -TCP/IP



* image is taken from "http://en.wikipedia.org/wiki/TCP/IP_model"

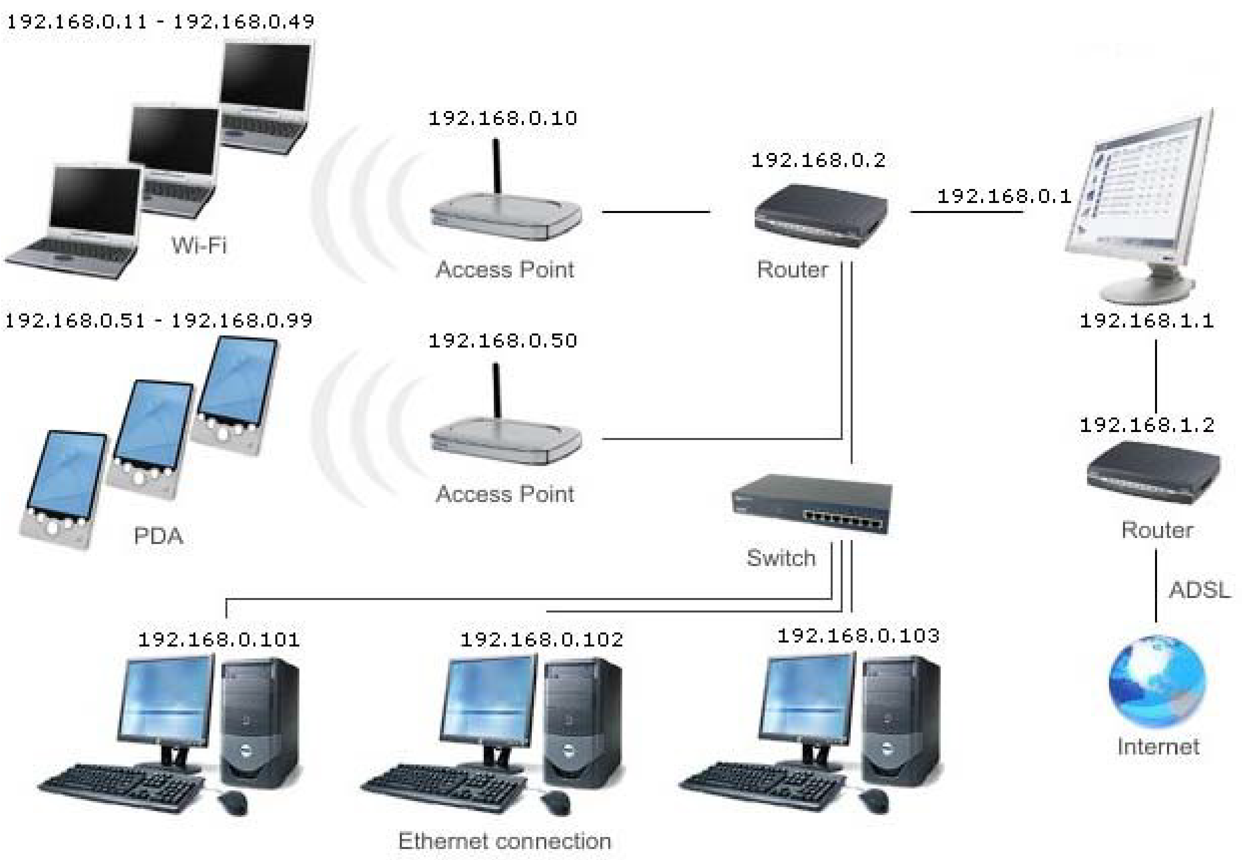
The sockets

Concerning system programming, there is the socket interface which provides a user-friendly abstraction of the basic mechanisms for implementing client/server programs. A socket (“**socket**”) is a communication end between processes.

Socket Programming is a method to connect two nodes over a network to establish a means of communication between those two nodes. A node represents a computer or a physical device with an internet connection. A socket is the endpoint used for connecting to a node. The signals required to implement the connection between two nodes are sent and received using the sockets on each node respectively.

Local Area Network Addresses - IPv4

- The 32 bits of an IPv4 address are broken into 4 octets, or 8 bit fields (0-255 value in decimal notation).



TCP vs UDP

- Both use **port numbers**
  - application-specific construct serving as a communication endpoint
  - 16-bit unsigned integer, thus ranging from 0 to 65535
  - to provide **end-to-end** transport
- UDP: User Datagram Protocol
  - no acknowledgements
  - no retransmissions
  - out of order, duplicates possible
  - connectionless, i.e., app indicates destination for each packet
- TCP: Transmission Control Protocol
  - reliable **byte-stream channel** (in order, all arrive, no duplicates)
    - similar to file I/O
  - flow control
  - connection-oriented
  - bidirectional

TCP vs UDP

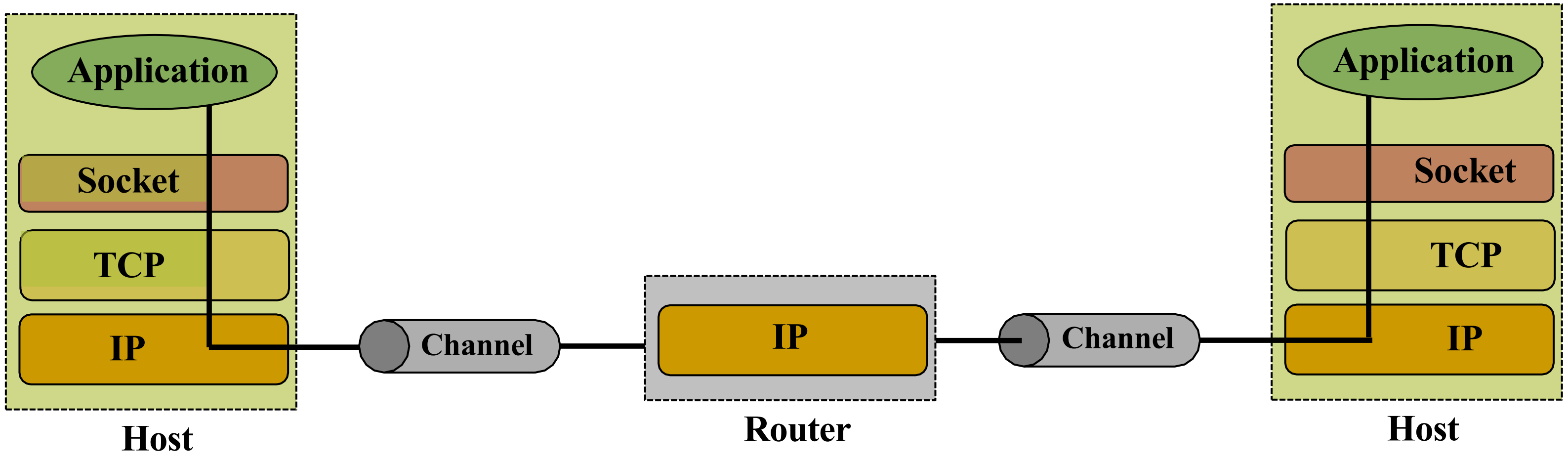
- TCP is used for services with a large data capacity, and a persistent connection
- UDP is more commonly used for quick lookups, and single use query-reply actions.

- Some common examples of TCP and UDP with their default ports:

DNS lookup	UDP	53	Domain Name System lookup
FTP	TCP	21	
HTTP	TCP	80	
SNMP	UDP	161	Simple Network Management Protocol
Telnet	TCP	23	
DHCP	UDP	67/68	Dynamic Host Configuration Protocol

Berkley Sockets

- Universally known as **Sockets**
- It is an abstraction through which an application may send and receive data
- Provide **generic access** to interprocess communication services
- Standard API for networking



Sockets

- Uniquely identified by
  - an internet address
  - an end-to-end protocol (e.g. TCP or UDP)
  - a port number
- Two types of (TCP/IP) sockets
  - Stream sockets (e.g. uses TCP)
    - provide reliable byte-stream service
  - Datagram sockets (e.g. uses UDP)
    - provide best-effort datagram service
    - messages up to 65.500 bytes
- Socket extend the convectional UNIX I/O facilities
  - file descriptors for network communication
  - the read and write system calls are extended

internal data structure

Family: AF_INET
Service: SOCK_STREAM
Local_IP:
Remote_IP:
Local_Port:
Remote_Port:
…

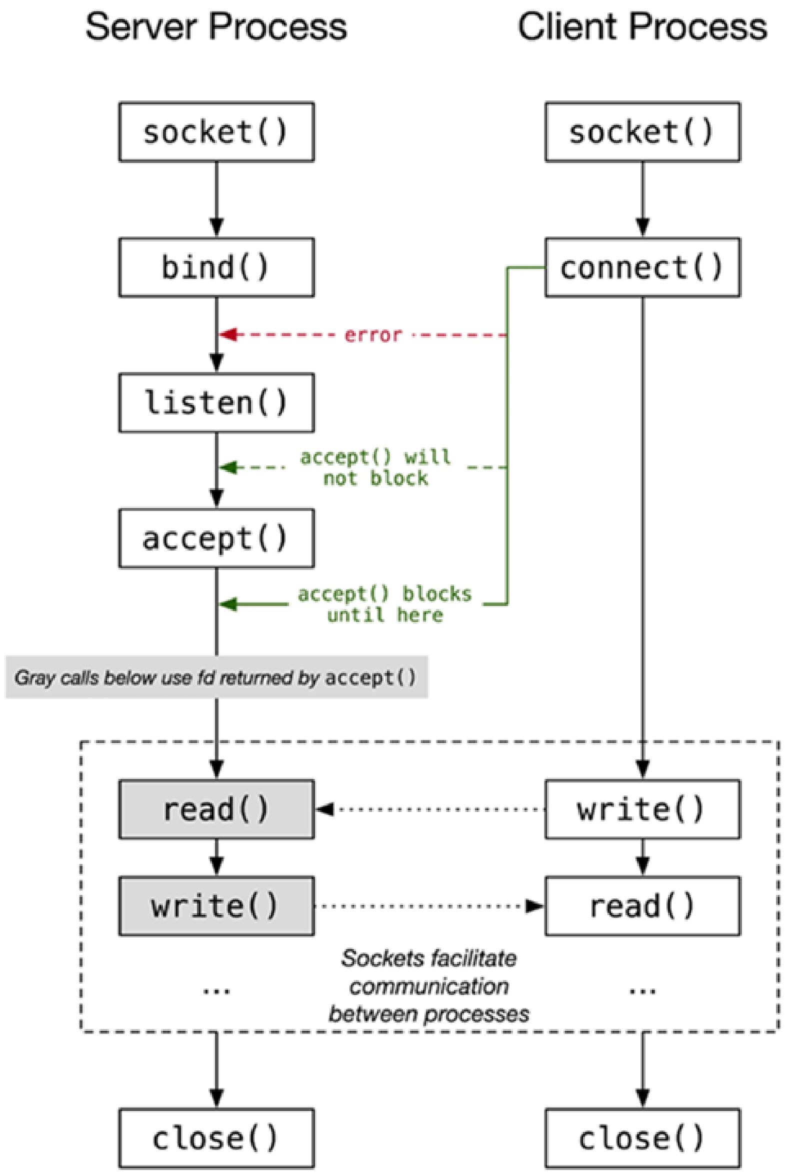
In simpler terms, a socket allows applications to talk to each other, either over the internet (e.g., between a client and server) or within the same computer.

The client-server model

Servers are normally divided into two main categories, and are called concurrent or iterative, based on their behavior.

- An iterative server responds to the request by sending data and remains busy and does not respond to further requests until it has provided a response to the request. Once the response is complete the server becomes available again.
- A concurrent server, instead, when processing the request, creates a child process (or a thread) responsible for providing the requested services, to immediately wait for further requests. In this way, with multitasking systems, multiple requests can be satisfied simultaneously. Once the child process has finished its work it is usually terminated, while the original server always remains active.

Client and server model state diagram



- The nodes are divided into two types, server node and client node.
- The client node sends the connection signal and the server node receives the connection signal sent by the client node.
- The connection between a server and client node is established using the socket over the transport layer of the internet.
- After a connection has been established, the client and server nodes can share information between them using the read and write commands.
- After sharing of information is done, the nodes terminate the connection.

Server stages

Different stages must be performed on the server node to receive a connection sent by the client node.

- Socket creation
- Setsockopt
- Bind
- Listen
- Accept
- Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() / send() system calls.

Client stages

The client-side sends the connection requests to the server-side. To send these requests several stages have to be performed on the client side too.

- Socket Connection
- Connect
- Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() / send() system calls.

Server stages

- Socket creation

The first stage deals with the creation of a socket, which is the basic component for sending or receiving signals between nodes. The sys/socket.h header has the necessary functions to create a socket in C. In socket programming in C, a socket can be created by the socket() function with syntax,

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

example

int server_fd = socket(AF_INET, SOCK_STREAM, 0);

Server stages

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

The **domain** represents the address family over which the communication will be performed. The domain is pre-fixed values present in the sys/socket.h header. Some domains are,

- **AF_LOCAL** or **AF_UNIX** is used for local communication or in the case where the client and server are on the same node. These sockets are called **UNIX domain sockets**.
- **AF_INET** is used to represent the IPv4 address of the client to which a connection should be made. Similarly, **AF_INET6** is used for IPv6 addresses. These sockets are called **internet domain sockets**.
- **AF_BLUETOOTH** is used for low-level Bluetooth connection.

Server stages

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

The **type** represents the type of communication used in the socket. Some mostly used types of communication are,

- **SOCK_STREAM** uses the TCP (Transmission Control Protocol) to establish a connection. This type provides a reliable byte stream of data flow and is a connection-based protocol. These sockets are called **stream sockets**.
- **SOCK_DGRAM** uses the UDP (User Datagram Protocol) which is unreliable and a connectionless protocol. These sockets are also called **datagram sockets**.

Server stages

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

The **protocol** represents the protocol used in the socket. This is represented by a number. When there is only one protocol in the protocol family, the protocol number will be 0, or else the specific number for the protocol has to be specified.

The **socket()** function creates a socket and returns a file descriptor which represents an open file that will be utilized by the socket in reading and writing operations and the file descriptor is used to represent the socket in later stages. In case of an error in creating the socket, -1 is returned by the socket() function.

Server stages

- Setsockopt

The setsockopt() function in socket programming in C is used to specify some options for the socket to control the behavior of the socket. The syntax is,

*int setsockopt(int socket_descriptor, int level, int option_name, const void *value_of_option, socklen_t option_length);*

example

int opt = 1;
setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

Server stages

*int setsockopt(int socket_descriptor, int level, int option_name, const void *value_of_option, socklen_t option_length);*

The **socket_descriptor** is the value of the file descriptor returned by the socket() function.

The **level** parameter represents the level at which the option for the socket must be applied. The SOL_SOCKET represents the socket level and IPPROTO_TCP represents the TCP level.

Server stages

*int setsockopt(int socket_descriptor, int level, int option_name, const void *value_of_option, socklen_t option_length);*

The **option_name** specifies the rules or options that should be modified for the socket. Some useful options are,

- SO_DEBUG is used to enable the recording of debugging information.
- SO_REUSEADDR is used to enable the reusing of local addresses in the bind() function.
- SO_SNDBUF is used to set the maximum buffer size that can be sent using the socket connection.
- SO_LINGER is used to set that socket lingers on close.

Server stages

*int setsockopt(int socket_descriptor, int level, int option_name, const void *value_of_option, socklen_t option_length);*

The **option_value** is used to specify the value for the options set in the option_name parameter.

The **option_length** is the length of the variable used to set the option value.

The function returns a value of 0 of data type int on the successful application of the option and a value of -1 on failure.

Server stages

- Bind

The bind() function in socket programming in C is used to assign an address to a socket created using the socket() function. The syntax of bind() function is,

*int bind(int socket_descriptor , const struct sockaddr *address, socklen_t length_of_address);*

The **socket_descriptor** is the value of the file descriptor returned by the socket() function.

Server stages

*int bind(int socket_descriptor , const struct sockaddr *address, socklen_t length_of_address);*

The **address** is a structure of type sockaddr. We usually use a structure of type sockaddr_in to represent this information, because information such as port and address can only be stored in this structure. The sockaddr_in is cast to the sockaddr data type when calling the bind() function.

The **length_of_address** represents the size of the address passed as the second parameter.

The function returns 0 on binding the address and port successfully or returns -1 on failure.

Server stages

*int bind(int socket_descriptor , const struct sockaddr *address, socklen_t length_of_address);*

example

```
struct sockaddr_in address;
socklen_t addrlen = sizeof(address);

address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(PORT);
bind(server_fd, (struct sockaddr*)&address, sizeof(address));
```

struct sockaddr_in

The struct **sockaddr_in** data type is a specialized structure used to represent IPv4 socket addresses in socket programming. It is built upon the generic struct sockaddr and includes fields specific to IPv4 addresses, such as the IP address and port number.

```
struct sockaddr_in {
    sa_family_t    sin_family;   // Address family (e.g., AF_INET)
    uint16_t       sin_port;     // Port number (network byte order)
    struct in_addr sin_addr;     // IPv4 address (32-bit)
    char           sin_zero[8];  // Padding to match size of sockaddr
};
```

struct sockaddr_in

Fields in struct sockaddr_in

- sin_family:
  - specifies the address family.
  - for IPv4, this is always set to AF_INET (Address Family - Internet).

address.sin_family = AF_INET;

- sin_port:
  - represents the port number on which the socket communicates.
  - it must be stored in network byte order (big-endian format).
  - use the htons() function (host-to-network short) to convert the port number.

address.sin_port = htons(8080); // Convert 8080 to network byte order

struct sockaddr_in

- sin_addr:
  - represents the IPv4 address.
  - it is defined as:

```
struct in_addr {
    uint32_t s_addr;  // 32-bit IPv4 address (network byte order)
};
```

  - Use the inet_addr() or inet_pton() functions to convert a string IP address to the required format.

```
address.sin_addr.s_addr = inet_addr("127.0.0.1");
```

or

```
inet_pton(AF_INET, "127.0.0.1", &address.sin_addr);
```

struct sockaddr_in

- sin_zero:
  - an unused field consisting of 8 bytes of padding.
  - it exists to make sockaddr_in the same size as the generic struct sockaddr.
  - it is typically set to 0 using memset():
    memset(address.sin_zero, 0, sizeof(address.sin_zero));

The struct sockaddr is a generic socket address structure in C yipically defined as:

```
struct sockaddr {
    sa_family_t sa_family;   // Address family (e.g., AF_INET for IPv4)
    char sa_data[14];        // Protocol-specific address data
};
```

Server stages

- Listen

The listen() function in socket programming is used to make the server node wait and listen for connections from the client node on the port and address specified by the bind() function. The syntax is,

int listen(int socket_descriptor, int back_log);

The **socket_descriptor** represents the value of the file descriptor returned by the socket() function.

Server stages

- *int listen(int socket_descriptor, int backlog);*

The **backlog** marks the maximum number of connection requests that can be made to the server by client nodes at a time. The number of requests made after the number specified by backlog may cause an error or will be ignored by the server if the options for retransmission are set.

The function returns 0 on listening on the address and port specified or returns -1 on failure.

example

listen(server_fd, 3);

Server stages

- Accept

The accept() function is used to establish a connection between the server and the client nodes for the transfer of data. This call typically blocks until a client connects with the server. The syntax is,

*int accept(int socket_descriptor, struct sockaddr *address, socklen_t *length_of_address);*

The **socket_descriptor** represents the value of the file descriptor returned by the socket() function.

Server stages

int accept(int socket_descriptor, struct sockaddr address, socklen_t* length_of_address);*

The **address** is the variable of the sockaddr_in structure in which the address of the socket returned from the function will be stored.

The **length_of_address** depicts the size of the address parameter.

The accept() function creates a new socket from the first connection request for the specified socket_descriptor and returns the file descriptor of the new socket. The file descriptor of this new socket is used in the read() and write() functions to send and receive data to and from the client node.

Server stages

The accept() function is called by a server application after it has set up a listening socket using listen(). It extracts the first connection request from the queue of pending connections, creates a new socket for the connection, and returns a file descriptor for this new socket. The new socket can then be used for communication with the client.

example

```
struct sockaddr_in clientAddr;
socklen_t addrSize = sizeof (clientAddr);

nw_socket = accept(server_fd, (struct sockaddr_in*)&clientAddr,  &addrSize));
```

Server stages

- read

The read() function is used to receive data between client and server. The syntax of read() function is,

*ssize_t read(int socket_descriptor, void *buffer, size_t size);*

The **socket_descriptor** represents the value of the socket descriptor returned by the accept() function.

The **buffer** represents the memory location where the data read is stored.

The **size** represents the maximum number of data bytes that can be stored in buffer.

Server stages

The read() function, on success, returns the number of bytes read (zero indicates end of stream). It is not an error if this number is smaller than the number of bytes requested. On error, -1 is returned, and errno is set to indicate the error.

example

```
ssize_t valread;
char buffer[1024];
valread = read(nw_socket, buffer, 1024 - 1);
// subtract 1 for the null terminator at the end
```

Server stages

- write

The write() function is used to send data between client and server. The syntax of write() function is,

*ssize_t write(int socket_descriptor, void *buffer, size_t count);*

The **socket_descriptor** represents the value of the socket descriptor returned by the accept() function.

The **buffer** represents the memory location where the data to be sent is stored.

The **count** represents the number of data bytes that are stored in buffer.

Server stages

The write() function, on success, returns the number of bytes written. On error, the function write() returns -1, and errno is set to indicate the error.

example

```
char hello[] = "Hello from server";
write(nw_socket, hello, strlen(hello));
```

Server stages

- close

The close() function deallocates the socket descriptor passed as argument. To deallocate means to make the socket descriptor available for return by subsequent calls to socket(). The syntax is close() function is

int close(socket_descriptor);

The **socket_descriptor** represents the value of the socket descriptor returned by the socket() or by the accept() function.

Upon successful completion, close() function returns 0; otherwise, -1 and errno is set to indicate the error.

Client stages

- Socket Connection

Similar to the server-side, the client-side also needs to create a socket using the socket() function. This will create a socket that can send the connection request to the server. The client can connect the socket to the address of the server using the connect() system call.

```
// Create client socket
cliSoc  = socket (AF_INET, SOCK_STREAM, 0);
if (cliSoc  < 0) {
    perror ("Error in socket creation");
    exit (1);
}
```

Client stages

- Connect

The connect() function is used to send the connection request and connect to the server node. The syntax of the function is,

*int connect(int socket_descriptor, const struct sockaddr *address, socklen_t length_of_address);*

The **socket_descriptor** represents the value of the file descriptor returned by the socket() function during the creation of a socket on the client-side.

The **address** represents the structure with the information of the address and port number of the server node to which the connection is to be made.

Client stages

*int connect(int socket_descriptor, const struct sockaddr *address, socklen_t length_of_address);*

The **length_of_address** is the size of the address structure used in the second parameter.

```
// Set server address parameters
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons (PORT);
serverAddr.sin_addr.s_addr = inet_addr (SERVER_IP);
// Connect to the server
if (connect (cliSoc , (struct sockaddr*) & serverAddr, sizeof (serverAddr)) < 0) {
    perror("Error in connecting to server");
    exit (1);
}
```

Client stages

*int connect(int socket_descriptor, const struct sockaddr *address, socklen_t length_of_address);*

The connect() function returns a value of 0 on successfully connecting with the server and returns a value of -1 on error or the connection fails.

Similar to the server-side, the client-side also can invoke the read() and write() functions to send and receive data between client and server and the close() function to close the socket stream.

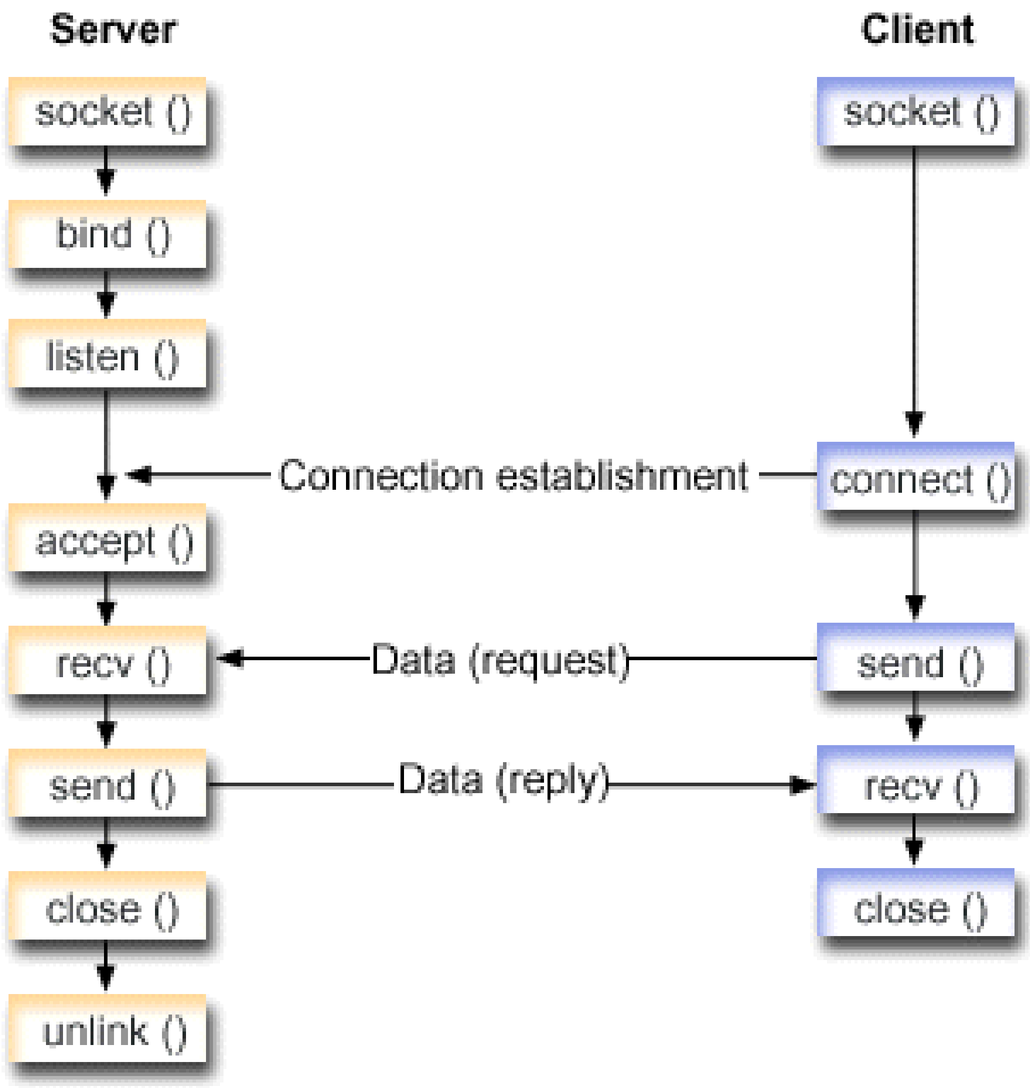
Let’s see an implementation in which one hello message is exchanged between server and client to demonstrate the AF_INET client/server model.

Concurrent server stages

A concurrent server, as said, when processing the request, creates a child process (or a thread) responsible for providing the requested services, to immediately wait for further requests.

```
pid_t pid = fork();
if (pid == 0) {
    // Child process
    close (serverSocket);
    // receive messages from the client, read() and send() functions invocation
}
else if (pid > 0 ) {
    // Parent process
    close(clientSocket);
}
```

Using AF_UNIX address family



Sockets that use the AF_UNIX address family can be connection-oriented (type SOCK_STREAM) or connectionless (type SOCK_DGRAM).

Both types are reliable because there are no external communication functions connecting the two processes.

Using AF_UNIX address family

Socket flow of events for a server application that uses AF_UNIX address family.

The **socket()** API returns a socket descriptor, which represents an endpoint. The statement also identifies the UNIX address family with the stream transport (SOCK_STREAM) being used for this socket. You can also use the socketpair() API to initialize a UNIX socket.

After the socket descriptor is created, the **bind()** API gets a unique name for the socket.

Using AF_UNIX address family

The name space for UNIX domain sockets consists of path names. When a sockets program calls the bind() API, an entry is created in the file system directory. If the path name already exists, the bind() fails. Thus, a UNIX domain socket program should always call an unlink() API to remove the directory entry when it ends.

The **listen()** allows the server to accept incoming client connections.

The server uses the **accept()** function to accept an incoming connection request. The accept() call will block indefinitely waiting for the incoming connection to arrive.

Using AF_UNIX address family

The **recv()** API receives data from the client application.

The **send()** API send data back to the client.

The **close()** API closes any open socket descriptors.

The **unlink()** API removes the UNIX path name from the file system.

Using AF_UNIX address family

Socket flow of events for a client application that uses AF_UNIX address family

The **socket()** API returns a socket descriptor, which represents an endpoint. The statement also identifies the UNIX address family with the stream transport (SOCK_STREAM) being used for this socket. You can also use the socketpair() API to initialize a UNIX socket.

After the socket descriptor is received, the **connect()** API is used to establish a connection to the server.

The **send()** API sends data bytes to the server.

Using AF_UNIX address family

The **recv()** API receives data bytes back from the server.

The **close()** API closes any open socket descriptors.

Let's see an implementation in which one hello message is exchanged between server and client to demonstrate the AF_UNIX client/server model.

Some useful functions


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

int getaddrinfo(const char  *nodename, const char *servname,
    const struct addrinfo *hints, struct addrinfo **res);


void freeaddrinfo(struct addrinfo *ai);
```

Some useful functions

*int getaddrinfo(const char *nodename, const char *servname,*
*const struct addrinfo *hints, struct addrinfo **res);*

The **getaddrinfo()** function translates the name of a service location (for example, a host name) and/or a service name and returns a set of socket addresses and associated information to be used in creating a socket with which to address the specified service.

The nodename and servname arguments are either null pointers or pointers to null-terminated strings. One or both of these two arguments must be a non-null pointer.

Some useful functions

*int getaddrinfo(const char *nodename, const char *servname,*
*const struct addrinfo *hints, struct addrinfo **res);*

It is part of the modern POSIX standard and is preferred over older functions like gethostbyname() and getservbyname() because it is protocol-independent (works for both IPv4 and IPv6) and thread-safe.

If the ***nodename*** argument is not null, it can be a descriptive name or can be an address string. Address strings using internet standard dot notation are valid if the specified address family is AF_INET or AF_UNSPEC.
If nodename is not null, the requested service location is named by nodename; otherwise, the requested service location is local to the caller.

Some useful functions

nodename:

- The hostname (e.g., "www.example.com") or IP address (e.g., "192.168.1.1" for IPv4 or "::1" for IPv6).
- If set to NULL, it means the local host.

Some useful functions

*int getaddrinfo(const char *nodename, const char *servname,*
*const struct addrinfo *hints, struct addrinfo **res);*

If ***servname*** argument is null, the call returns network-level addresses for the specified nodename. If servname is not null, it is a null-terminated character string identifying the requested service. This can be either a descriptive name or a numeric representation suitable for use with the address family or families. If the specified address family is AF_INET, AF_INET6 or AF_UNSPEC, the service can be specified as a string specifying a decimal port number.

- The service name (e.g., "http", "ftp") or the port number as a string (e.g., "80").
- If NULL, no service is specified.

Some useful functions


```
int getaddrinfo(const char  *nodename, const char *servname,
    const struct addrinfo *hints, struct addrinfo **res);
```

The addrinfo structure used by getaddrinfo() contains the following fields:

```
struct addrinfo {
    int             ai_flags;
    int             ai_family;
    int             ai_socktype;
    int             ai_protocol;
    socklen_t       ai_addrlen;
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next;
};
```

Some useful functions

*int getaddrinfo(const char *nodename, const char *servname,*
*const struct addrinfo *hints, struct addrinfo **res);*

If the argument ***hints*** is not null, it refers to a structure containing input values that may direct the operation by providing options and by limiting the returned information to a specific socket type, address family and/or protocol. In this hints structure every member other than ai_flags, ai_family, ai_socktype and ai_protocol must be zero or a null pointer.

Some useful functions

*int getaddrinfo(const char *nodename, const char *servname,*
*const struct addrinfo *hints, struct addrinfo **res);*

ai_family

This field specifies the desired address family for the returned addresses. Valid values for this field include AF_INET and AF_INET6. The value AF_UNSPEC indicates that getaddrinfo() should return socket addresses for any address family (either IPv4 or IPv6, for example) that can be used with node and service.

Some useful functions

*int getaddrinfo(const char *nodename, const char *servname,*
*const struct addrinfo *hints, struct addrinfo **res);*

ai_socktype

This field specifies the preferred socket type, for example SOCK_STREAM or SOCK_DGRAM. Specifying 0 in this field indicates that socket addresses of any type can be returned by getaddrinfo().

ai_protocol

This field specifies the protocol for the returned socket addresses. Specifying 0 in this field indicates that socket addresses with any protocol can be returned by getaddrinfo().

Some useful functions

*int getaddrinfo(const char *nodename, const char *servname,*
*const struct addrinfo *hints, struct addrinfo **res);*

If hints is a null pointer, the behavior must be as if it referred to a structure containing the value zero for the ai_flags, ai_socktype and ai_protocol fields, and AF_UNSPEC for the ai_family field.

Some useful functions

*int getaddrinfo(const char *nodename, const char *servname,
const struct addrinfo *hints, struct addrinfo **res);*

The getaddrinfo() function allocates and initializes a linked list of addrinfo structures, one for each network address that matches node and service, subject to any restrictions imposed by hints, and returns a pointer to the start of the list in ***res***. The items in the linked list are linked by the ai_next field.

There are several reasons why the linked list may have more than one addrinfo structure. Normally, the application should try using the addresses in the order in which they are returned.

Some useful functions

*int getaddrinfo(const char *nodename, const char *servname, const struct addrinfo *hints, struct addrinfo **res);*

If *hints.ai_flags* includes the AI_CANONNAME flag, then the ai_canonname field of the first of the addrinfo structures in the returned list is set to point to the official name of the host.

The *ai_family*, *ai_socktype*, and *ai_protocol* fields return the socket creation parameters (i.e., these fields have the same meaning as the corresponding arguments of socket() function). For example, ai_family might return AF_INET or AF_INET6; ai_socktype might return SOCK_DGRAM or SOCK_STREAM; and ai_protocol returns the protocol for the socket.

or one of the following nonzero error codes

Some useful functions

*int getaddrinfo(const char *nodename, const char *servname,*
*const struct addrinfo *hints, struct addrinfo **res);*

A pointer to the socket address is placed in the *ai_addr* field, and the length of the socket address, in bytes, is placed in the *ai_addrlen* field.

getaddrinfo() returns 0 if it succeeds, or one of the following nonzero error codes: EAI_ADDRFAMILY, EAI_AGAIN, EAI_BADFLAGS, EAI_FAIL, EAI_FAMILY, EAI_MEMORY, EAI_NODATA, EAI_NONAME, EAI_SERVICE, EAI_SOCKTYPE, EAI_SYSTEM.
The gai_strerror() function translates these error codes to a human readable string, suitable for error reporting.

or one of the following nonzero error codes

Some useful functions

```
int getaddrinfo(const char  *nodename, const char *servname,
    const struct addrinfo *hints, struct addrinfo **res);

struct addrinfo *result = NULL, *ptr = NULL, hints;
…
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
rc = getaddrinfo(“myhost.mydomain.com”, “8080”, &hints, &result);
for (ptr = result; ptr != NULL; ptr = ptr->ai_next) {
      mySocket = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
      …
}
```

Some useful functions


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

int getnameinfo(const struct sockaddr *sa, socklen_t salen,
    char *node, socklen_t nodelen, char *service,
    socklen_t servicelen, unsigned int flags);
```

The **getnameinfo()** function is the inverse of getaddrinfo() function. It translates a socket address to a node name and service location, all of which are defined as with getaddrinfo().
The argument sa points to a socket address structure to be translated.

Some useful functions


```
int main(int argc, char *argv[]) {
    struct addrinfo hints, *res, *p;
    int status;
    char ipstr[INET6_ADDRSTRLEN]; // Buffer to hold IP address (IPv4 or IPv6)

    if (argc != 2) {
        fprintf(stderr, "Usage: %s hostname\n", argv[0]);
        return 1;
    }

    // Initialize the hints structure
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;     // Allow IPv4 or IPv6
    hints.ai_socktype = SOCK_STREAM; // TCP socket

    // Call getaddrinfo to resolve hostname
    if ((status = getaddrinfo(argv[1], "http", &hints, &res)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status));
        return 2;
    }

    printf("IP addresses for %s:\n\n", argv[1]);
```

```
// Loop through all the results
    for (p = res; p != NULL; p = p->ai_next) {
        void *addr;
        char *ipver;
        // Get the pointer to the address itself
        if (p->ai_family == AF_INET) { // IPv4
            struct sockaddr_in *ipv4 = (struct sockaddr_in *)p->ai_addr;
            addr = &(ipv4->sin_addr);
            ipver = "IPv4";
        } else { // IPv6
            struct sockaddr_in6 *ipv6 = (struct sockaddr_in6 *)p->ai_addr;
            addr = &(ipv6->sin6_addr);
            ipver = "IPv6";
        }

        // Convert the IP to a string and print it
        inet_ntop(p->ai_family, addr, ipstr, sizeof ipstr);
        printf("  %s: %s\n", ipver, ipstr);
    }
    freeaddrinfo(res); // Free the linked list
    return 0;
}
```

The getnameinfo() function in socket programming is used to convert a socket address (e.g., an IP address and port) back into a human-readable string su

Some useful functions

*int getnameinfo(const struct sockaddr *sa, socklen_t salen,*
*char *node, socklen_t nodelen, char *service,*
socklen_t servicelen, unsigned int flags);

The getnameinfo() function in socket programming is used to convert a socket address (e.g., an IP address and port) back into a human-readable string such as a hostname and service name (or port number). It essentially performs the reverse operation of getaddrinfo().
It translates a binary socket address (e.g., struct sockaddr_in for IPv4 or struct sockaddr_in6 for IPv6) into a readable format, replacing older functions like gethostbyaddr() and inet_ntoa() and working seamlessly with IPv4 and IPv6.

The getnameinfo() function in socket programming is used to convert a **socket address** (e.g., an IP address and port) back into a **human-readable string** su

Some useful functions

*int getnameinfo(const struct sockaddr *sa, socklen_t salen,*
*char *node, socklen_t nodelen, char *service,*
socklen_t servicelen, unsigned int flags);

- sa (Socket Address): a pointer to the socket address structure (e.g., struct sockaddr_in or struct sockaddr_in6).
- salen: the size (in bytes) of the socket address structure.
- node: a pointer to a buffer where the hostname will be stored.
- nodelen: the size of the buffer allocated for the hostname.
- service: a pointer to a buffer where the service name (e.g., "http") or port number (e.g., "80") will be stored.
- servicelen: the size of the buffer allocated for the service name.
- flags: a set of flags that control the behavior of the function (explained below).

The getnameinfo() function in socket programming is used to convert a **socket address** (e.g., an IP address and port) back into a **human-readable string** su

Some useful functions

*int getnameinfo(const struct sockaddr *sa, socklen_t salen,*
*char *node, socklen_t nodelen, char *service,*
socklen_t servicelen, unsigned int flags);

If the argument node is non-NULL and the argument nodelen is nonzero, then the argument node points to a buffer able to contain up to nodelen characters that will receive the node name as a null-terminated string. If the argument node is NULL or the argument nodelen is zero, the node name will not be returned. If the node's name cannot be located, the numeric form of the node's address is returned instead of its name.

Some useful functions

*int getnameinfo(const struct sockaddr *sa, socklen_t salen,*
*char *node, socklen_t nodelen, char *service,*
socklen_t servicelen, unsigned int flags);

If the argument service is non-NULL and the argument servicelen is nonzero, then the argument service points to a buffer able to contain up to servicelen characters that will receive the service name as a null-terminated string. If the argument service is NULL or the argument servicelen is zero, the service name will not be returned. If the service's name cannot be located, the numeric form of the service address (for example, its port number) is returned instead of its name.

The arguments node and service cannot both be NULL.

Some useful functions

*int getnameinfo(const struct sockaddr *sa, socklen_t salen,*
*char *node, socklen_t nodelen, char *service,*
socklen_t servicelen, unsigned int flags);

The flags argument is a flag that changes the default actions of the function. By default the fully-qualified domain name (FQDN) for the host is returned, but

- If the flag bit NI_NOFQDN is set, only the nodename portion of the FQDN is returned for local hosts.
- If the flag bit NI_NUMERICHOST is set, the numeric form of the host's address is returned instead of its name, under all circumstances.

Some useful functions

*int getnameinfo(const struct sockaddr *sa, socklen_t salen,*
*char *node, socklen_t nodelen, char *service,*
socklen_t servicelen, unsigned int flags);

- If the flag bit NI_NAMEREQD is set, an error is returned if the host's name cannot be located.
- If the flag bit NI_NUMERICSERV is set, the numeric form of the service address is returned (for example, its port number) instead of its name, under all circumstances.
- If the flag bit NI_DGRAM is set, this indicates that the service is a datagram service (SOCK_DGRAM). The default behavior is to assume that the service is a stream service (SOCK_STREAM).

Some useful functions

*int getnameinfo(const struct sockaddr *sa, socklen_t salen,*
*char *node, socklen_t nodelen, char *service,*
socklen_t servicelen, unsigned int flags);

On success, 0 is returned, and node and service names, if requested, are filled with null-terminated strings, possibly truncated to fit the specified buffer lengths. On error, one of the following nonzero error codes is returned: EAI_AGAIN, EAI_BADFLAGS, EAI_FAIL, EAI_FAMILY, EAI_MEMORY, EAI_NONAME, EAI_OVERFLOW, EAI_SYSTEM.

The gai_strerror() function translates these error codes to a human readable string, suitable for error reporting.