DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

CONCURRENT AND REAL TIME PROGRAMMING
[INQ0091623] AA 2021-22

Lab 7

Distributed IPC with Sockets

Gabriele Manduchi <gabriele.manduchi@unipd.it>

Andrea Rigoni Garola <andrea.rigonigarola@unipd.it>

Why distributed computing

Distributed applications involving network communication are used in embedded systems for a variety of reasons, among which are

- **Computing Power** : Whenever the computing power needed by the application cannot be provided by a single computer, it is necessary to distribute the application among different machines, each carrying out a part of the required computation and coordinating with the others via the network.

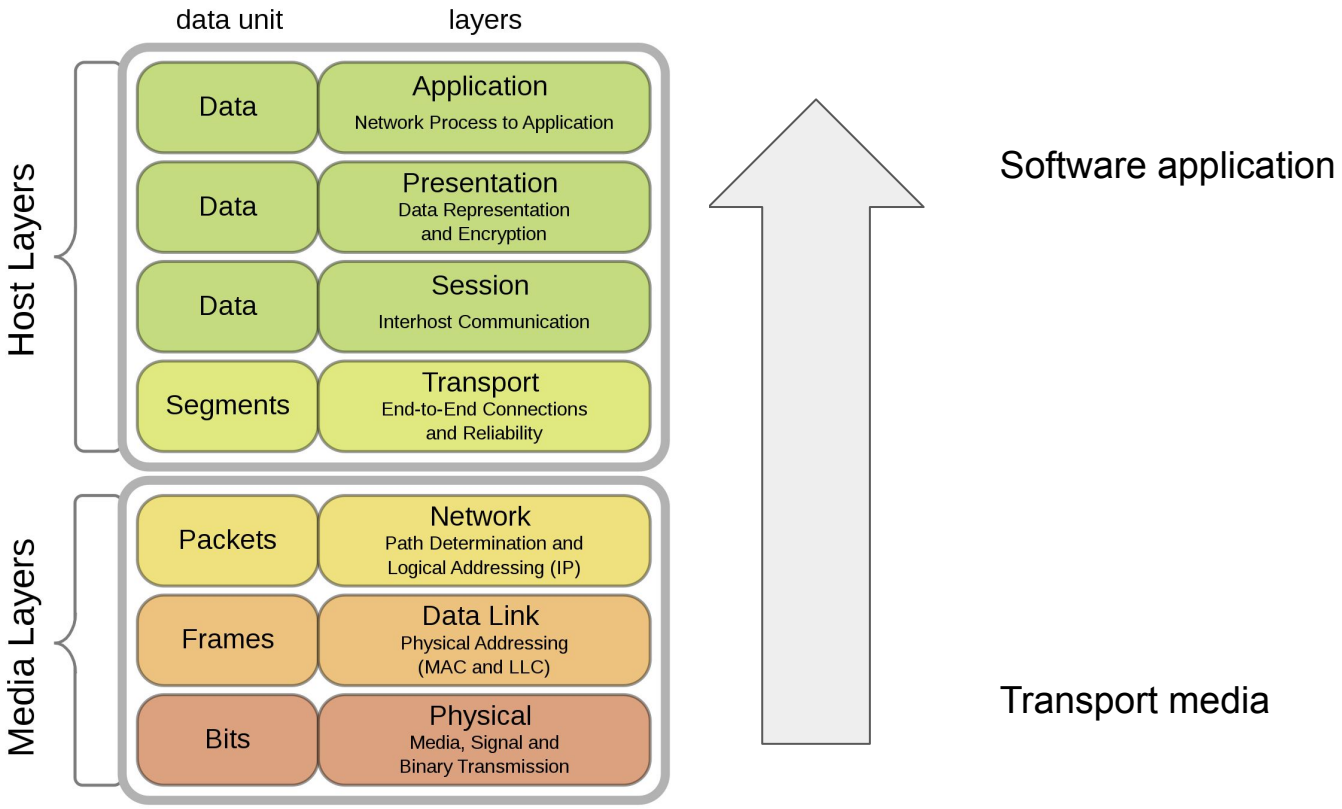
- **Distributed Data**: Often, an embedded system is required to acquire and elaborate data coming from different locations in the controlled plant. In this case, one or more computers will be dedicated to data acquisition and first-data processing. They will then send preprocessed data to other computers that will complete the computation required for the control loop.

- **Single Point of failure**: For some safety-critical applications, such as aircraft control, it is important that the system does not exhibit a single point of failure, that is, the failure of a single computer cannot bring the system down. In this case, it is necessary to distribute the computing load among separate machines so that, in case of failure of one of them, another one can resume the activity of the failed component.

Osi model

Every time different computers are connected for exchanging information, it is necessary that they strictly adhere to a communication protocol.

To fit any possible constraint added by transport media and the software architecture the complexity of the network protocol has been divided in several layers called the OSI model.



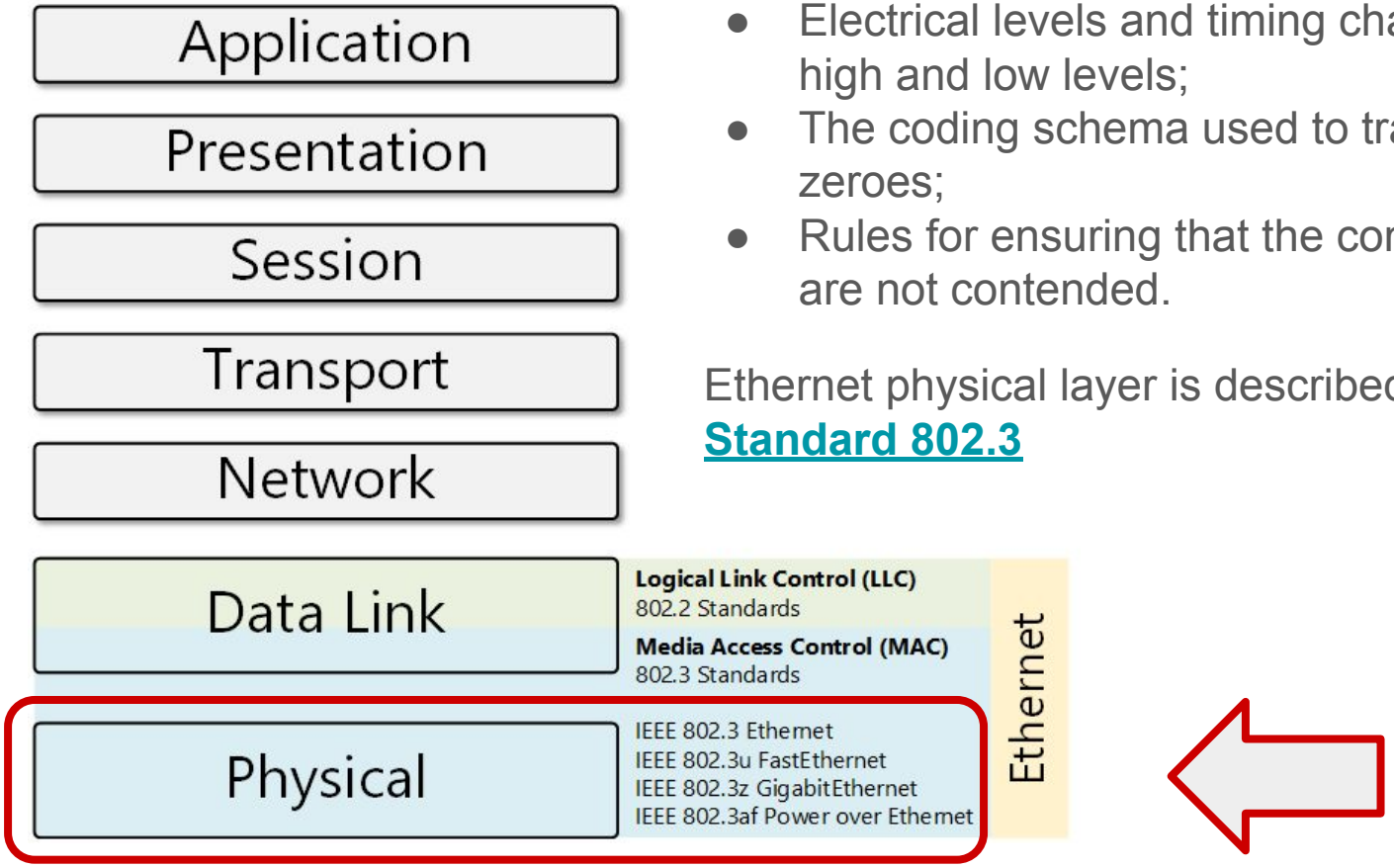
ETHERNET EXAMPLE: physical layer

Ethernet protocol defines the **physical layer** and the data **link layer**.

The physical layer defines the electrical characteristics of the communication media, including:

- Number of communication lines;
- Impedance for input and output electronics;
- Electrical levels and timing characteristics for high and low levels;
- The coding schema used to transmit ones and zeroes;
- Rules for ensuring that the communication links are not contended.

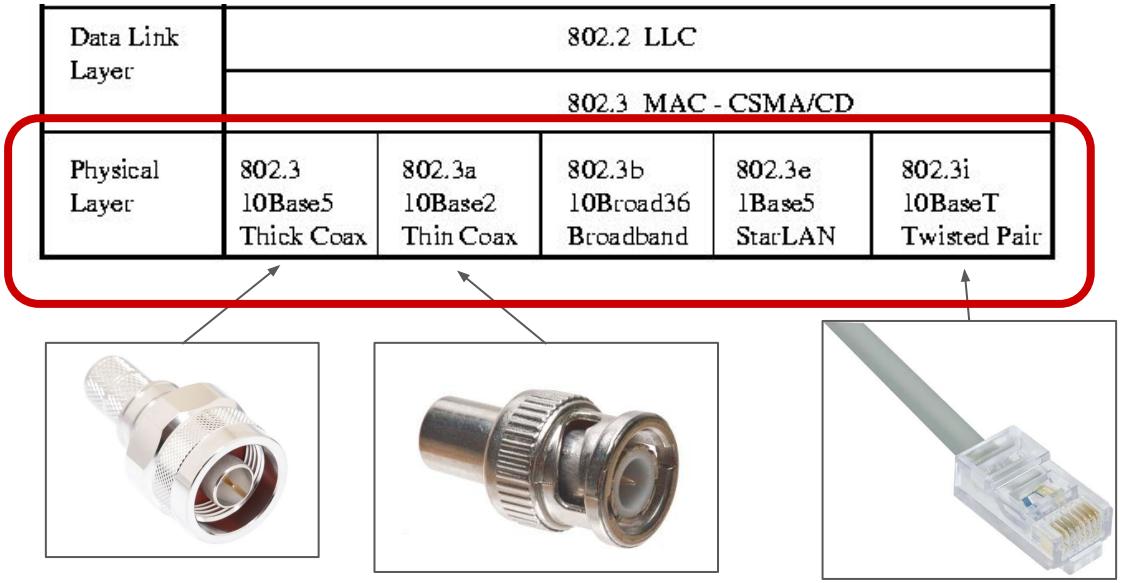
Ethernet physical layer is described by the **IEEE Standard 802.3**



ETHERNET EXAMPLE: physical layer

Ethernet protocol defines the **physical layer** and the data **link layer**.

The physical layer defines the electrical characteristics of the communication media, including:

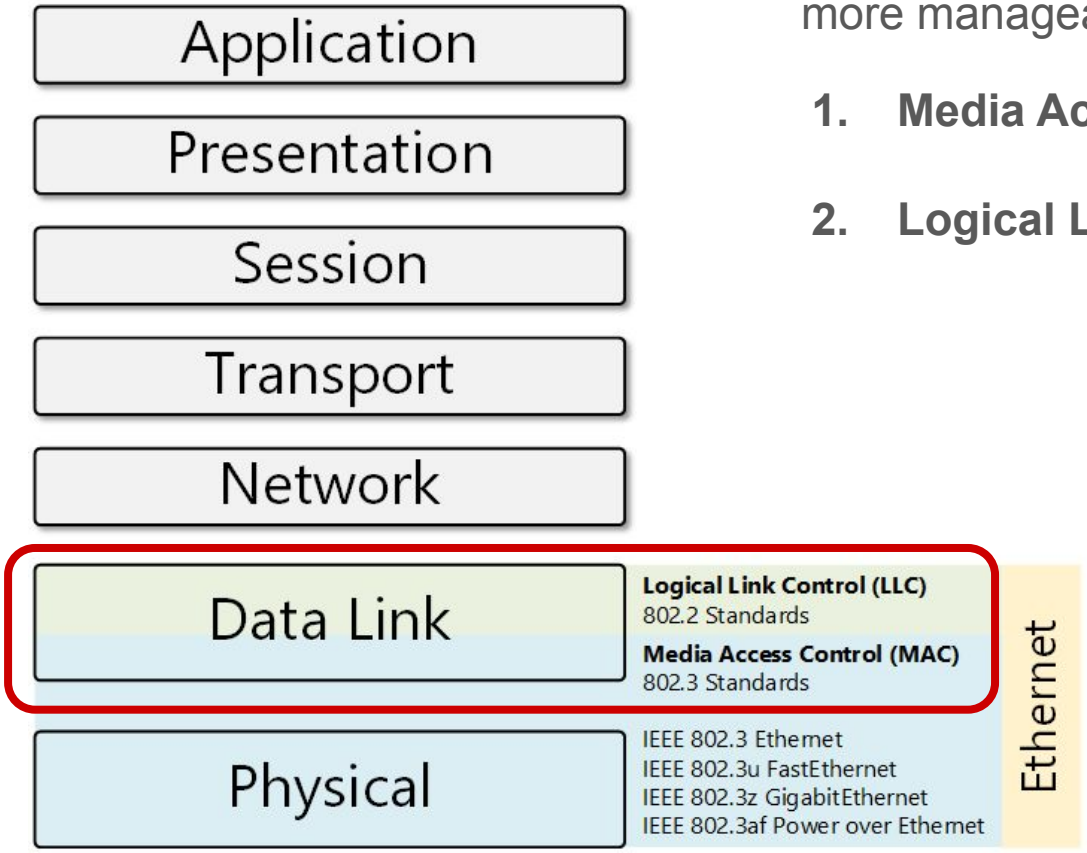


ETHERNET EXAMPLE: data link

The **data link** layer defines **how information is coded by using the transmitted logical zeroes and ones** (how logical zeroes and ones are transmitted is defined by the physical layer).

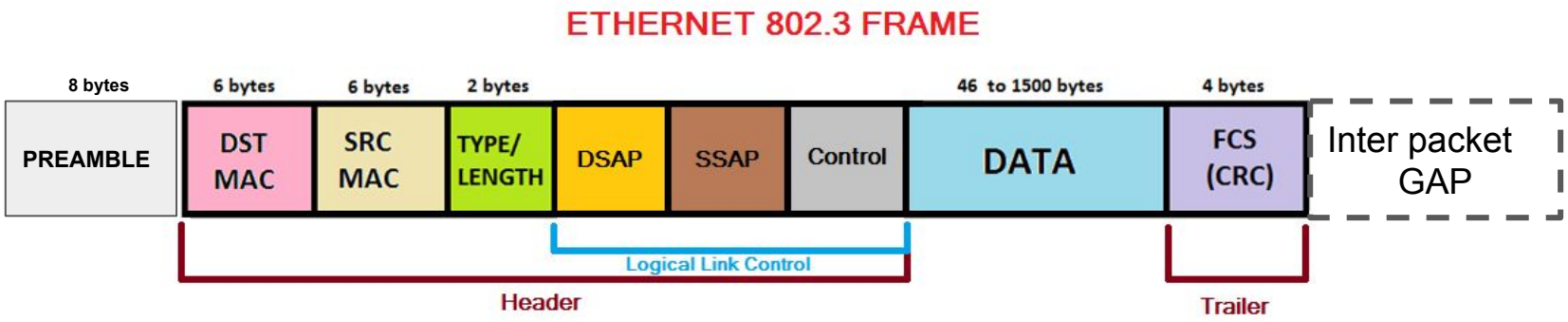
Due to its complexity, the ethernet data link layer is split into two or more sublayers to make it more manageable:

1. **Media Access Control (MAC)**
2. **Logical Link Control (LLC)**



Data link Frames:

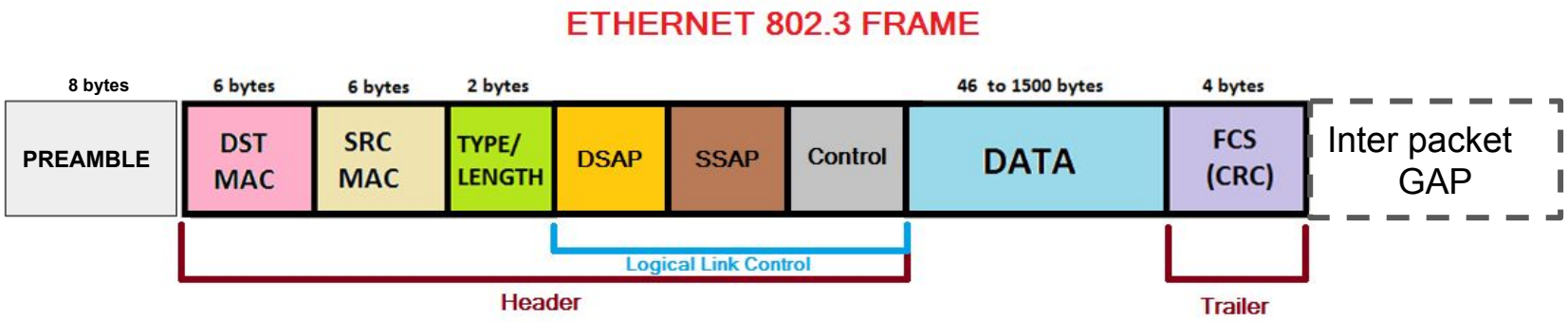
Data exchanged over Ethernet is grouped in Frames, and every frame is a packet of binary data that contains the following fields:



1. **Preamble and Start Frame** Identifier (8 octets): Formed by a sequence of identical octets with a predefined value (an octet in network communication terminology corresponds to a byte), followed by a single octet whose value differs only for the least significant bit. The preamble and the start frame identifier are used to detect the beginning of the frame in the received bit stream.
2. **MAC Destination** (6 octets): the Media Access Control (MAC) address of the designated receiver for the frame.
3. **MAC Source** (6 octets): The MAC address of the sender of the frame.
4. **Packet Length** (2 octets): Coding either the length of the data frame or other special information about the packet type.

Data link Frames:

Data exchanged over Ethernet is grouped in Frames, and every frame is a packet of binary data that contains the following fields:



5. **LLC (IEEE 802.2)**

6. **Payload** (46–1500 octets): Frame data.

7. **CRC** (4 octets): Cyclic Redundancy Check (CRC) used to detect possible communication errors. This field is obtained from the frame content at the time the frame is sent, and the same algorithm is performed when the packet is received. If the new CRC value is different form the CRC field, the packet is discarded because there has indeed been a transmission error.

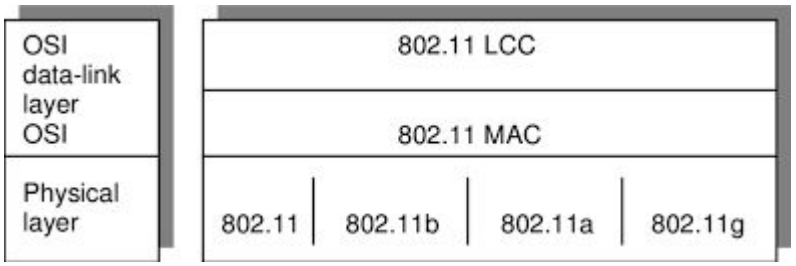
8. **Interframe Gap** (12 octets): Minimum number of bytes between different frames.

Other protocols

Ethernet has largely replaced competing wired LAN technologies such as Token Ring, FDDI and ARCNET.

Token ring -> IEEE 802.5 (1989) It uses a special three-byte frame called a token that is passed around a logical ring of workstations or servers.

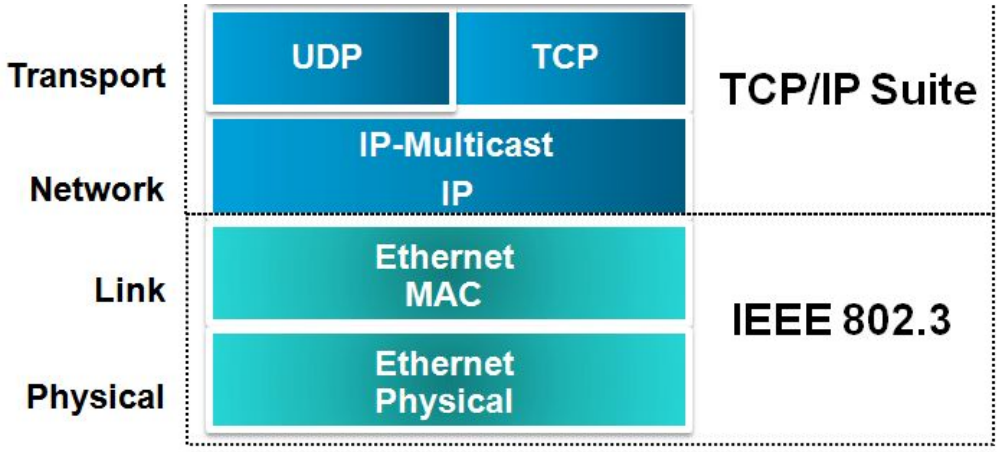
Ethernet has been built for cables, WiFi IEEE 802.11 is a wireless alternative.



Network and Transport

Many different network communication layers are defined for different network protocols, and every layer, normally built on top of one or more other layers, provides some added functionality in respect of that provided by the layers below.

Here we shall consider the **Internet Protocol (IP)**, which addresses the Network Layer, and the **Transmission Control Protocol (TCP)**, addressing the Transport Layer. IP and TCP together implement the well-known **TCP/IP protocol**. Both protocols are specified and discussed in detail in a number of Request for Comments (RFC), a series of informational and standardization documents about Internet.

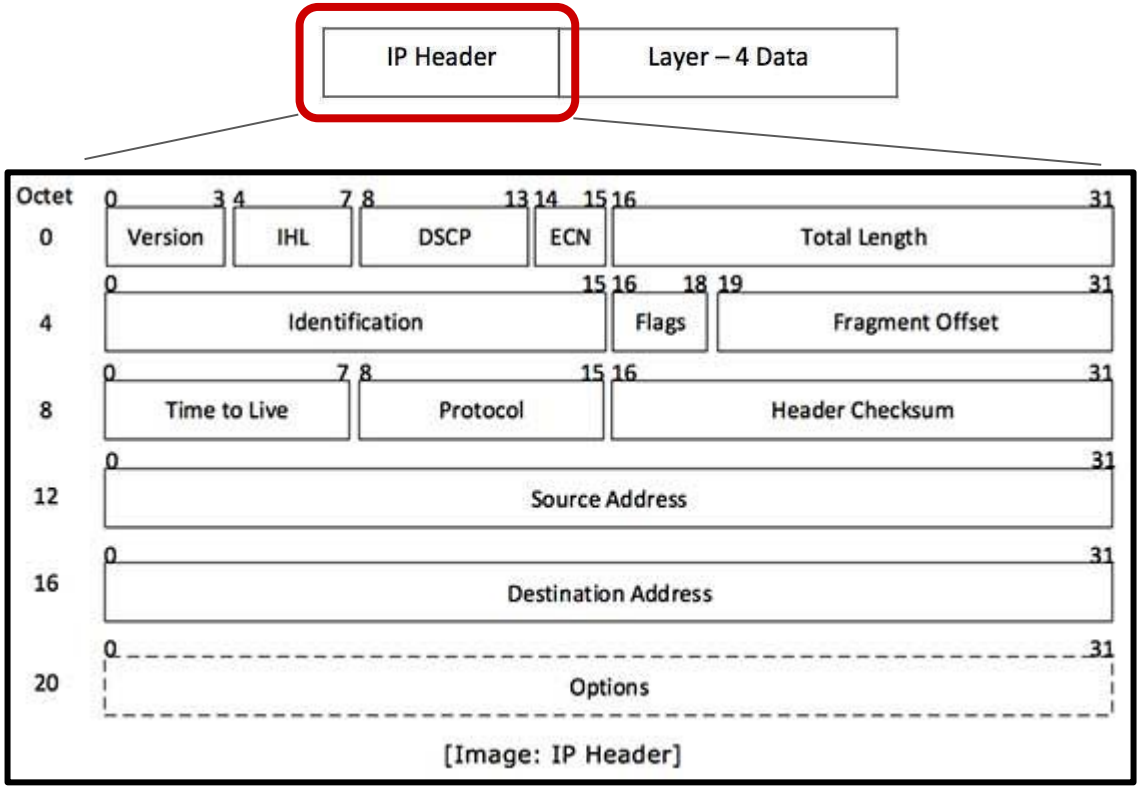


IP

The IP defines the functionality needed for handling the transmission of packets along one or more networks and performs two basic functions:

1. **Addressing:** It defines an hierarchical addressing system using IP addresses represented by a 4-byte integer (**IPv4**).

2. **Routing:** It defines the rules for achieving communication among different networks, that is, getting packets of data from source to destination by sending them from network to network. Based on the destination IP address, the data packet will be sent over the local networks from router to router up to the final destination.



[Image: IP Header]

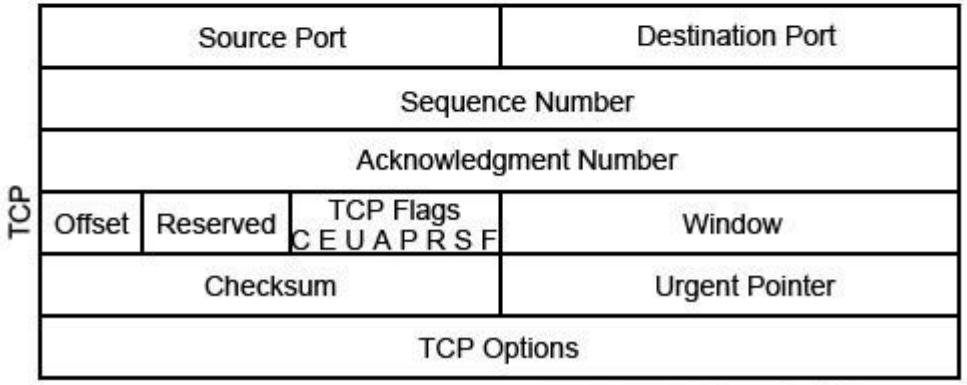
IP

- **Version** − Version no. of Internet Protocol used (e.g. IPv4).
- **IHL** − Internet Header Length; Length of entire IP header.
- **DSCP** − Differentiated Services Code Point; this is Type of Service.
- **ECN** − Explicit Congestion Notification; It carries information about the congestion seen in the route.
- **Total Length** − Length of entire IP Packet (including IP header and IP Payload).
- **Identification** − If IP packet is fragmented during the transmission, all the fragments contain same identification number. to identify original IP packet they belong to.
- **Flags** − As required by the network resources, if IP Packet is too large to handle, these ‘flags’ tells if they can be fragmented or not. In this 3-bit flag, the MSB is always set to ‘0’.
- **Fragment Offset** − This offset tells the exact position of the fragment in the original IP Packet.
- **Time to Live** − To avoid looping in the network, every packet is sent with some TTL value set, which tells the network how many routers (hops) this packet can cross. At each hop, its value is decremented by one and when the value reaches zero, the packet is discarded.
- **Protocol** − Tells the Network layer at the destination host, to which Protocol this packet belongs to, i.e. the next level Protocol. For example protocol number of ICMP is 1, TCP is 6 and UDP is 17.
- **Header Checksum** − This field is used to keep checksum value of entire header which is then used to check if the packet is received error-free.
- **Source Address** − 32-bit address of the Sender (or source) of the packet.
- **Destination Address −** 32-bit address of the Receiver (or destination) of the packet.
- **Options** − This is optional field, which is used if the value of IHL is greater than 5. These options may contain values for options such as Security, Record Route, Time Stamp, etc.

TCP

TCP provides reliable, ordered, and error-checked delivery of a stream of octets (bytes) between applications running on hosts communicating via an IP network.

Major internet applications such as the World Wide Web, email, remote administration, and file transfer rely on TCP, which is part of the Transport Layer of the TCP/IP suite. SSL/TLS often runs on top of TCP.

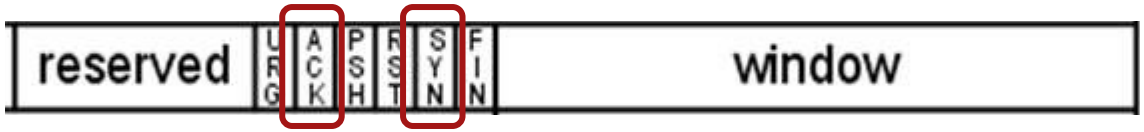


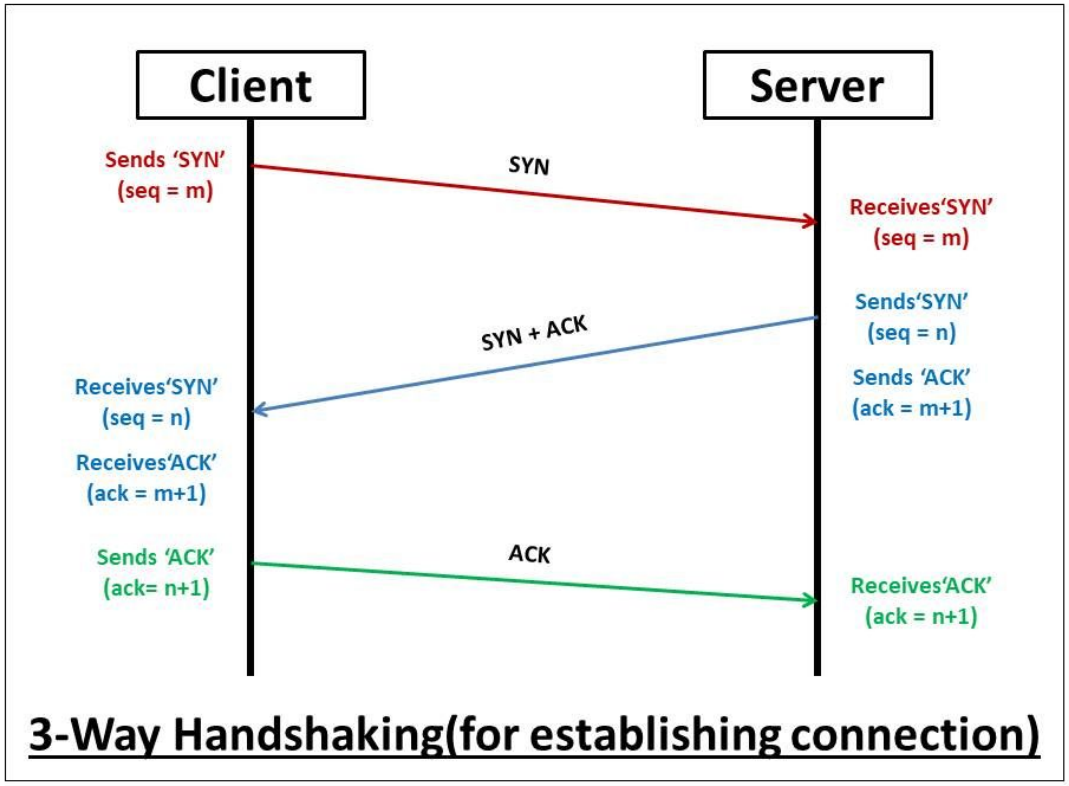
TCP

- **Source TCP port number** (2 bytes): TCP port number of the sending device.
- **Destination TCP port number** (2 bytes): TCP port receiving device.
- **Sequence number** (4 bytes or 32 bits): Message senders use sequence numbers to mark the ordering of a group of messages.
- **Acknowledgment number** (4 bytes or 32 bits): Both senders and receivers use the acknowledgment numbers field to communicate the sequence numbers of messages that are either recently received or expected to be sent.
- **TCP data offset** (4 bits): The data offset field stores the total size of a TCP header in multiples of four bytes. A header not using the optional TCP field has a data offset of 5 (representing 20 bytes), while a header using the maximum-sized optional field has a data offset of 15 (representing 60 bytes).
- **Control flags** (up to 9 bits): TCP uses a set of six standard and three extended control flags each an individual bit representing On or Off to manage data flow in specific situations.
- **Window size** (2 bytes or 16 bits): TCP senders use a number, called window size, to regulate how much data they send to a receiver before requiring an acknowledgment in return.
  TCP checksum (2 bytes or 16 bits): The checksum value inside a TCP header is generated by the protocol sender as a mathematical technique to help the receiver detect messages that are corrupted or tampered with.
- **Urgent pointer** (2 bytes or 16 bits): The urgent pointer field is often set to zero and ignored, but in conjunction with one of the control flags, it can be used as a data offset to mark a subset of a message as requiring priority processing.

TCP 3-Way Handshake Process

The 3-Way Handshake process is the defined set of steps that takes place in the TCP for creating a secure and reliable communication link and also closing it.



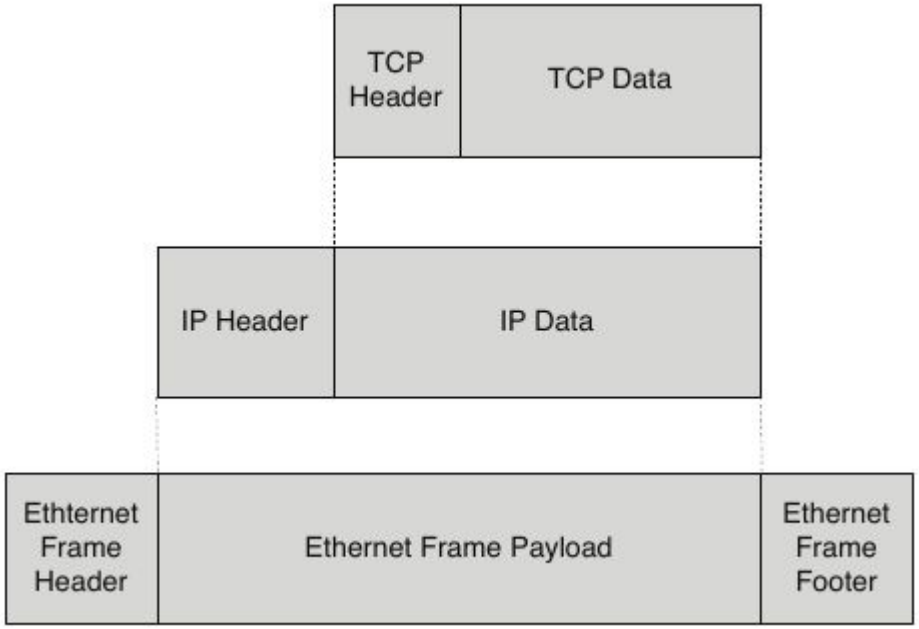


TCP/IP Stack

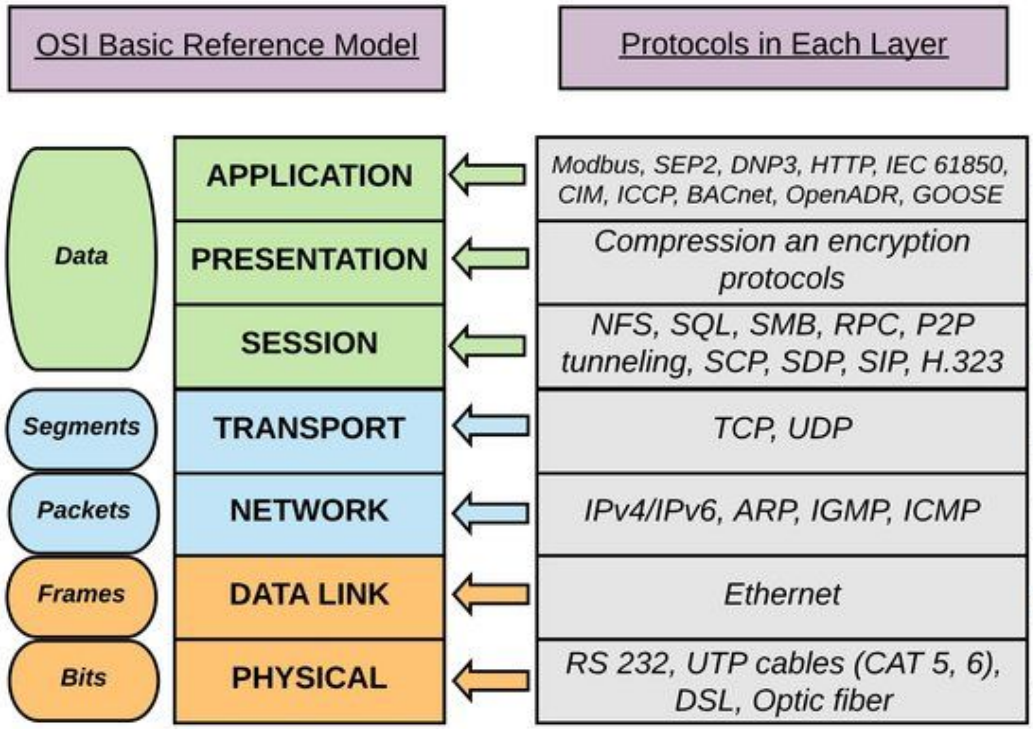
All protocols are organized that one layer knows nothing about the upper layers.

Being the TCP layer built on top of the Internet layer, the latter cannot know anything about the structure of the TCP data packet, which is contained in the data part of the Internet packet and considered payload.

For this reason this organization is called the TCP Stack.



OSI model vs Protocols



Sockets

A **network socket** is a software structure within a network node of a computer network that serves as an endpoint for sending and receiving data across the network.

The structure and properties of a socket are defined by an application programming interface (API) for the networking architecture. Sockets are created only during the lifetime of a process of an application running in the node.

A socket is a **pseudo-file** that represents a network connection. Once a socket has been created (identifying the other host and port), writes to that socket are turned into network packets that get sent out, and data received from the network can be read from the socket.

PSEUDO CODE

```
Socket mysocket = getSocket(type = "TCP")
bind(mysocket, address, port)
listen(mysocket)
recv(mysocket, str)
send(mysocket, "Hello, from server!")
close(mysocket)
```

```
Socket mysocket = getSocket(type = "TCP")
connect(mysocket, address = "1.2.3.4", port = "80")
send(mysocket, "Hello, from client!")
recv(mysocket, str)
close(mysocket)
```

Socket types

Several types of Internet socket are available:

Datagram sockets (UDP)

Connectionless sockets, which use **User Datagram Protocol (UDP).** Each packet sent or received on a datagram socket is individually addressed and routed. Order and reliability are not guaranteed with datagram sockets, so multiple packets sent from one machine or process to another may arrive in any order or might not arrive at all.

Stream sockets (TCP)

Connection-oriented sockets, which use **Transmission Control Protocol (TCP)**, Stream Control Transmission Protocol (SCTP) or Datagram Congestion Control Protocol (DCCP). A stream socket provides a sequenced and unique flow of error-free data without record boundaries, with well-defined mechanisms for creating and destroying connections and reporting errors. **A stream socket transmits data reliably, in order, and with out-of-band capabilities.**

Raw sockets

Allow direct sending and receiving of IP packets without any protocol-specific transport layer formatting. With other types of sockets, the payload is automatically encapsulated according to the chosen transport layer protocol (e.g. TCP, UDP), and the socket user is unaware of the existence of protocol headers that are broadcast with the payload.

Request a **socket**

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

DOMAIN (some):

Name	Purpose	Man page
AF_UNIX	Local communication	unix(7)
AF_LOCAL	Synonym for AF_UNIX	
AF_INET	IPv4 Internet protocols	ip(7)
AF_APPLETALK	AppleTalk	ddp(7)
AF_INET6	IPv6 Internet protocols	ipv6(7)
AF_NETLINK	Kernel user interface device	netlink(7)
AF_PACKET	Low-level packet interface	packet(7)
AF_RDS	Reliable Datagram Sockets (RDS) protocol	rds(7)
...		

TYPE:

SOCK_STREAM

SOCK_DGRAM

SOCK_RAW

EXAMPLE

```
int fd = socket(AF_INET,      // network versus AF_LOCAL
                SOCK_STREAM, // reliable, bidirectional, arbitrary payload size
                0);          // system picks underlying protocol (TCP)
if (fd < 0) report("socket", 1); // terminate
```

bind address (Server)

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

bind() assigns the address specified by *addr* to the socket referred to by the file descriptor *sockfd*. *addrlen* specifies the size, in bytes, of the address structure pointed to by *addr*.

The actual structure passed for the *addr* argument will depend on the address family. The *sockaddr* structure is defined as something like:

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

EXAMPLE

```
/* bind the server's local address in memory */
  struct sockaddr_in saddr;
  memset(&saddr, 0, sizeof(saddr));          /* clear the bytes */
  saddr.sin_family = AF_INET;                /* versus AF_LOCAL */
  saddr.sin_addr.s_addr = htonl(INADDR_ANY); /* host-to-network endian */
  saddr.sin_port = htons(PortNumber);        /* for listening */

  if (bind(fd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0)
    report("bind", 1); /* terminate */
```

listen and **accept** incoming connections (Server)

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

listen() marks the socket referred to by *sockfd* as a passive socket, that is, as a socket that will be used to accept incoming connection requests using accept.
The *backlog* argument defines the maximum length to which the queue of pending connections for *sockfd* may grow. When the queue is full, the client may receive an error with an indication of **ECONNREFUSED.**

accept() is used with connection-based sockets. It extracts the first connection request on the queue of pending connections for the listening socket, *sockfd*, creates a new connected socket, and returns a new file descriptor referring to that socket.

The *sockfd* argument is a file descriptor that refers to a socket of type **SOCK_STREAM** (or **SOCK_SEQPACKET**)

```
/* listen to the socket */
 if (listen(fd, MaxConnects) < 0) /* listen for clients, up to MaxConnects */
   report("listen", 1); /* terminate */
 while (1) {
   struct sockaddr_in caddr; /* client address */
   int len = sizeof(caddr);  /* address length could change */

   int client_fd = accept(fd, (struct sockaddr*) &caddr, &len);  /* accept blocks */
   if (client_fd < 0) {
     report("accept", 0); /* don't terminate, though there's a problem */
     continue;
     }
…  } /* while */
```

connect (Client)

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

The **connect**() system call connects the socket referred to by the file descriptor *sockfd* to the address specified by *addr*. The *addrlen* argument specifies the size of *addr*. The format of the address in *addr* is determined by the address space of the socket *sockfd*.

```
/* get the address of the host */
struct hostent* hptr = gethostbyname(Host); /* localhost: 127.0.0.1 */
if (!hptr) report("gethostbyname", 1); /* is hptr NULL? */
if (hptr->h_addrtype != AF_INET)       /* versus AF_LOCAL */
  report("bad address family", 1);

/* connect to the server: configure server's address 1st */
struct sockaddr_in saddr;
memset(&saddr, 0, sizeof(saddr));
saddr.sin_family = AF_INET;
saddr.sin_addr.s_addr =
   ((struct in_addr*) hptr->h_addr_list[0])->s_addr;
saddr.sin_port = htons(PortNumber); /* port number in big-endian */

if (connect(sockfd, (struct sockaddr*) &saddr, sizeof(saddr)) < 0)
  report("connect", 1);
```

TcpClient

TcpClient.c

CODING EXAMPLE CODING EXAMPLE

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

#define FALSE 0
#define TRUE 1

// Receive routine
static int receive(int sd, char *retBuf, int size) {
 int totSize, currSize;
 totSize = 0;
 while(totSize < size) {
   currSize = recv(sd, &retBuf[totSize], size - totSize, 0);
   if(currSize <= 0)
/* An error occurred */
     return -1;
   totSize += currSize;
 }
 return 0;
}
```

TcpClient.c

CODING EXAMPLE

CODING EXAMPLE

```
int main(int argc, char **argv)
{
 char hostname[100];
 char command[256];
 char *answer;
 int  sd;
 int port;
 int stopped = FALSE;
 int len;
 unsigned int netLen;
 struct sockaddr_in sin;
 struct hostent *hp;
/* Check number of arguments and get IP address and port */
 if (argc < 3) {
   printf("Usage: client <hostname> <port>\n");
   exit(0);
 }
 sscanf(argv[1], "%s", hostname);
 sscanf(argv[2], "%d", &port);

/* Resolve the passed name and store the resulting long representation
  in the struct hostent variable */
 if ((hp = gethostbyname(hostname)) == 0) {
   perror("gethostbyname");
   exit(1);
 }
```

TcpClient.c

CODING EXAMPLE CODING EXAMPLE

```
/* fill in the socket structure with host information */
 memset(&sin, 0, sizeof(sin));
 sin.sin_family = AF_INET;
 sin.sin_addr.s_addr = ((struct in_addr *)(hp->h_addr_list[0]))->s_addr;
 sin.sin_port = htons(port);
/* create a new socket */
 if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
   perror("socket");  exit(1);  }
/* connect the socket to the port and host  specified in struct sockaddr_in */
 if (connect(sd,(struct sockaddr *)&sin, sizeof(sin)) == -1)  {
   perror("connect");   exit(1); }

 while(!stopped)  {
/* Get a string command from terminal */
   printf("Enter command: ");
   scanf("%s", command);
   if(!strcmp(command, "quit"))  break;
   len = strlen(command); // Send first the number of characters in the command
   netLen = htonl(len); // Convert the integer number into network byte order
   /* Send number of characters */
   if(send(sd, &netLen, sizeof(netLen), 0) == -1)   {
     perror("send");     exit(1);   }
   /* Send the command */
   if (send(sd, command, len, 0) == -1)   {
     perror("send");     exit(0);   }
```

TcpClient.c


```
/* Receive the answer: first the number of characters  and then the answer itself */
   if(receive(sd, (char *)&netLen, sizeof(netLen))) {
     perror("recv");  exit(0);    }
/* Convert from Network byte order */
   len = ntohl(netLen);
/* Allocate and receive the answer */
   answer = malloc(len + 1);
   if(receive(sd, answer, len))    {
     perror("recv");  exit(1);  }
   answer[len] = 0;
   printf("%s\n", answer);
   free(answer);
   if(!strcmp(command, "stop"))
     break;
 } /* END WHILE */


/* Close the socket */
 close(sd);
 return 0;
}
```

TcpServer

TcpServer.c

CODING EXAMPLE CODING EXAMPLE

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>


static int receive(int sd, char *retBuf, int size)  {
 int totSize, currSize;
 totSize = 0;
 while(totSize < size) {
   currSize = recv(sd, &retBuf[totSize], size - totSize, 0);
   if(currSize <= 0)
     /* An error occurred */
     return -1;
   totSize += currSize;
 }
 return 0;
}
```

TcpServer.c

CODING EXAMPLE CODING EXAMPLE

```
/* Handle an established  connection  routine receive is listed in the previous example */
static void handleConnection(int currSd)  {
 unsigned int netLen;
 int len, exit_status = 0;
 char *command, *answer;
 for(;;)  {
   if(receive(currSd, (char *)&netLen, sizeof(netLen)))   break;
   len = ntohl(netLen);
   command = malloc(len+1);
   receive(currSd, command, len);
   command[len] = 0;
   if(strcmp(command,"help") == 0)
     answer = strdup("server is active. \n\n" );
   else if (strcmp(command,"stop") == 0) {
     answer = strdup("closing server connection" );
     exit_status = 1;   }
   else  answer = strdup("invalid command (try help)." );
/* Send the answer back */
   len = strlen(answer);
   netLen = htonl(len);
   if (send(currSd, &netLen, sizeof(netLen), 0) == -1)  break;
   if (send(currSd, answer, len, 0) == -1)  break;
   free(command);
   free(answer);
   if (exit_status) break;
 }
/* The loop is most likely exited when the connection is terminated */
 printf("Connection terminated \n");
 close(currSd);
}
```

TcpServer.c

CODING EXAMPLE CODING EXAMPLE

```
/* Main Program */
int main(int argc, char *argv[]) {
 int    sd, currSd;
 int    sAddrLen, port;
 int    len;
 unsigned int netLen;
 char *command, *answer;
 struct   sockaddr_in sin, retSin;
 if(argc < 2) {  printf("Usage: server <port>\n");   exit(0);  }
 sscanf(argv[1], "%d", &port);
 /* Create a new socket */
 if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
   perror("socket");   exit(1);  }
   /* set socket options REUSE ADDRESS */
   int reuse = 1;
   if (setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (const char*)&reuse, sizeof(reuse)) < 0)
       perror("setsockopt(SO_REUSEADDR) failed");
#ifdef SO_REUSEPORT
   if (setsockopt(sd, SOL_SOCKET, SO_REUSEPORT, (const char*)&reuse, sizeof(reuse)) < 0)
       perror("setsockopt(SO_REUSEPORT) failed");
#endif
/* Initialize the address (struct sokaddr_in) fields */
 memset(&sin, 0, sizeof(sin));
 sin.sin_family = AF_INET;
 sin.sin_addr.s_addr = INADDR_ANY;
 sin.sin_port = htons(port);
```

TcpServer.c


```
/* Bind the socket to the specified port number */
 if (bind(sd, (struct sockaddr *) &sin, sizeof(sin)) == -1)
 {   perror("bind");   exit(1);  }
/* Set the maximum queue length for clients requesting connection to 5 */
 if (listen(sd, 5) == -1)
 {   perror("listen");   exit(1); }

 sAddrLen = sizeof(retSin);
/* Accept and serve all incoming connections in a loop */
 for(;;) {
   if ((currSd = accept(sd, (struct sockaddr *) &retSin, &sAddrLen)) == -1)
      {     perror("accept");     exit(1);   }
/* When execution reaches this point a client established the connection.
   The returned socket (currSd) is used to communicate with the client */
   printf("Connection received from %s\n", inet_ntoa(retSin.sin_addr));
   handleConnection(currSd);
 }
  return 0; // never reached
}
```

TcpThreadedServer

TcpThreadedServer.c

CODING EXAMPLE

```c
/* Thread routine. It calls routine handleConnection() defined in the previous program. */
static void *connectionHandler(void *arg) {
   int currSock = *(int *)arg;
   handleConnection(currSock);
   free(arg);
   pthread_exit(0);
   return NULL;
}

/* Main Program */
int main(int argc, char *argv[])
{
...
 pthread_t threads[MAX_THREADS];
...
/* Accept and serve all incoming connections in a loop */
 for(int i=0; i<MAX_THREADS; ++i)
 {
   /* Allocate the current socket. It will be freed just before thread termination. */
   currSd = (int*)malloc(sizeof(int));
   if ((*currSd = accept(sd, (struct sockaddr *) &retSin, &sAddrLen)) == -1)
   { perror("accept"); exit(1); }
   printf("Connection received from %s\n", inet_ntoa(retSin.sin_addr));
   /* Connection received, start a new thread serving the connection */
   pthread_create(&threads[i], NULL, connectionHandler, currSd);
 }
```