E-MODULE

CLASS-MSc(COMPUTER SCIENCE)-III SEM

SUBJECT-NETWORK PROGRAMMING

TOPIC-TCP CONNECTION ESTABLSIHMENT AND TERMINATION

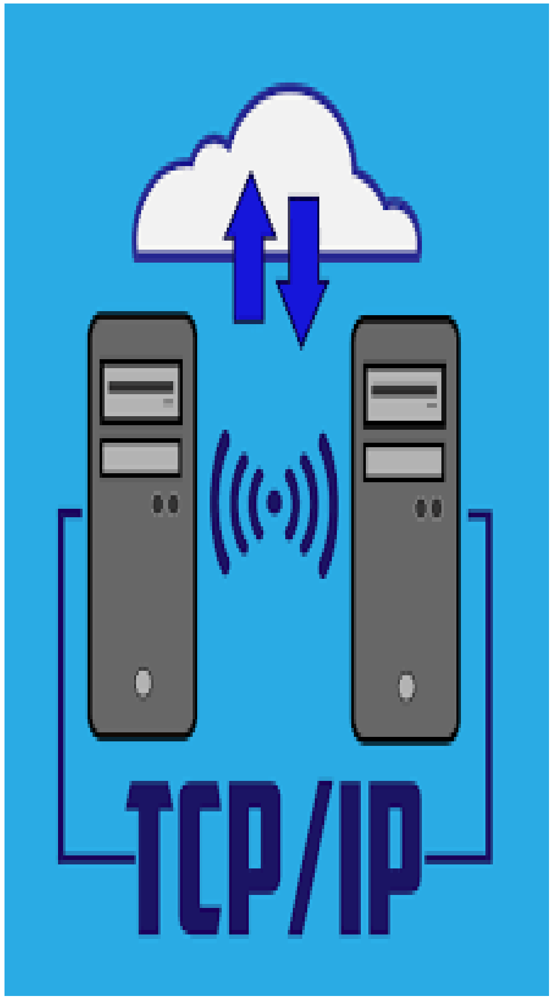
SUBMITTED BY:-

SAKSHI KAPOOR

(DEPARTMENT OF COMPUTER SCIENCE AND IT)

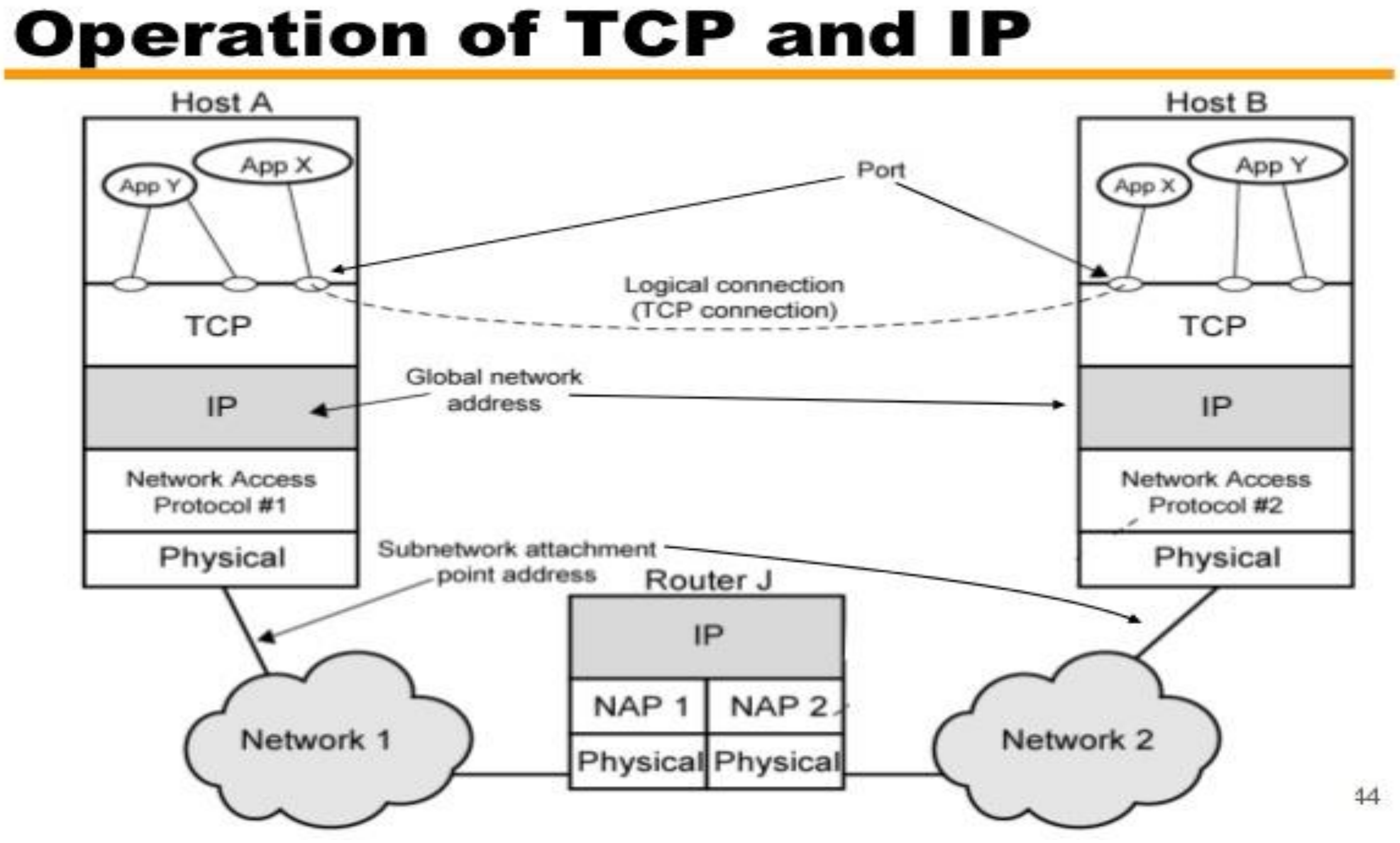
INTRODUCTION TO TRANSMISSION CONTROL PROTOCOL (TCP)

1. Transmission Control Protocol/Internet Protocol (**TCP/IP**) is the language a computer uses to access the Internet.
2. It consists of a suite of protocols designed to establish a network of networks to provide a host with access to the Internet.
3. **TCP** is one of the main protocols in **TCP**/IP networks. Whereas the IP protocol deals only with packets, **TCP** enables two hosts to establish a connection and exchange streams of data.
4. **TCP** guarantees delivery of data and also guarantees that packets will be delivered in the same order in which they were sent.
5. The **Transmission Control Protocol** (TCP) is one of the main **protocols** of the Internet **protocol** suite. It originated in the initial network implementation in which it complemented the Internet **Protocol** (IP). Therefore, the entire suite is commonly referred to as TCP/IP.

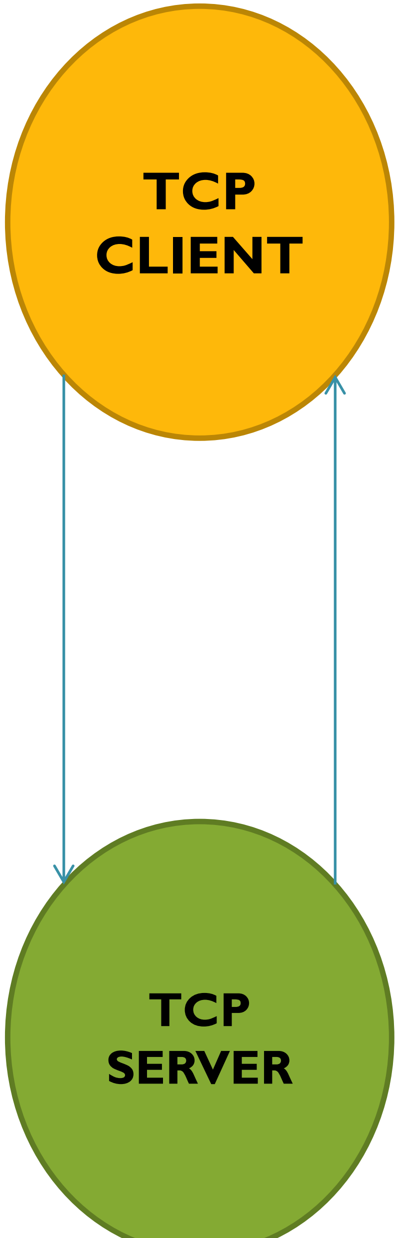


WORKING OF TCP/IP

A TCP client establishes a connection with a given server, exchanges data with that server across the connection, and then terminates the connection



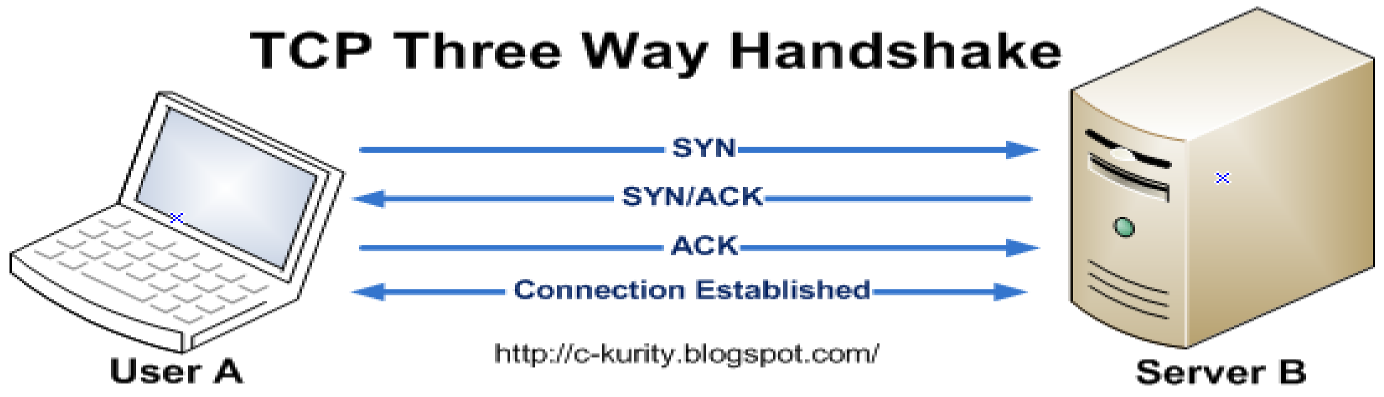
TCP CONNECTION ESTABLISHMENT



1. As TCP is the connection-oriented and reliable protocol therefore it first establishes a connection with the server and then only the communication and data transfer between client and server will take place.
2. The TCP Client and TCP Sever will first make a reliable connection to each other and then the process of packet exchange will come in process.
3. TCP also ensures the satisfactory transmission of packets from client to server or vice versa.
4. TCP also make this sure that after the complete transmission of data , the connection will be terminated so that a reliable dialogue session can be take place between client and server.

PROCESS OF CONNECTION ESTABLISHMENT

✓ **(THREE WAY HANDSHAKE)**



To establish a connection, the three-way (or 3-step) handshake occurs:

1. SYN: The active open is performed by the client sending a SYN to the server. The client sets the segment's sequence number to a random value A.
2. SYN-ACK: In response, the server replies with a SYN-ACK. The acknowledgment number is set to one more than the received sequence number (A + 1), and the sequence number that the server chooses for the packet is another random number, B.
3. ACK: Finally, the client sends an ACK back to the server. The sequence number is set to the received acknowledgement value i.e. A + 1, and the acknowledgement number is set to one more than the received sequence number i.e. B + 1.

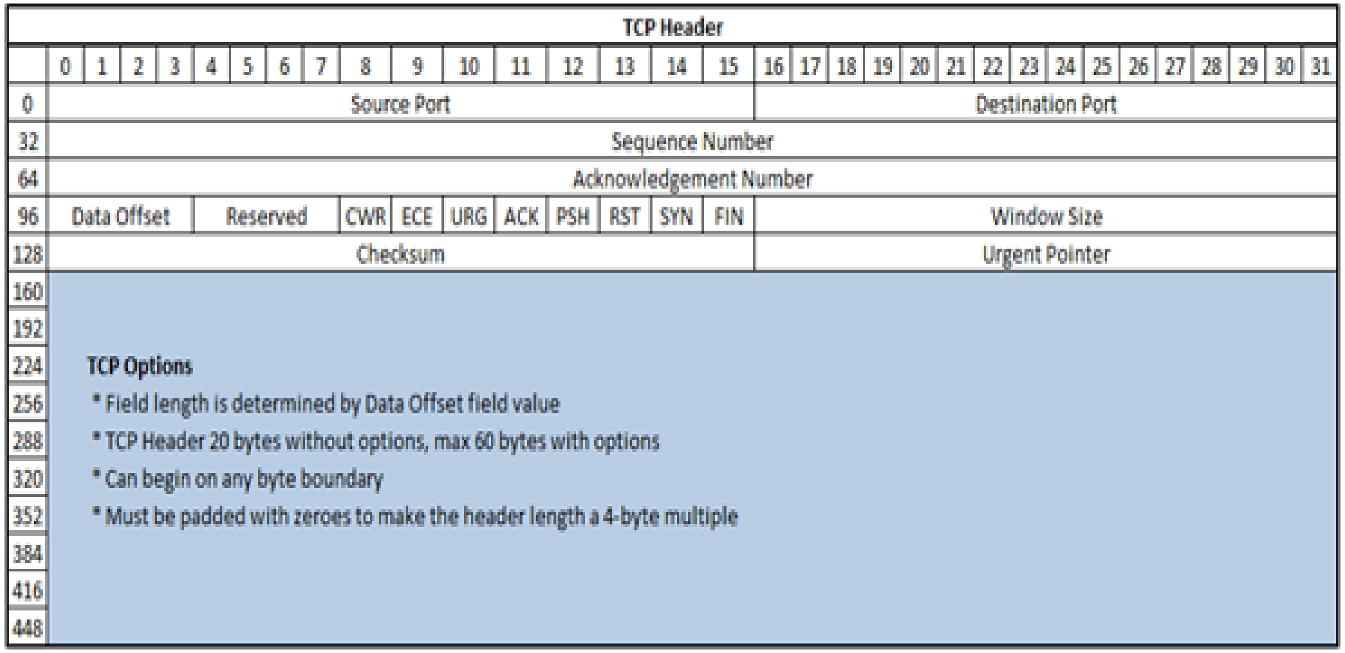
TCP Options

Each SYN can contain TCP options. Commonly used options include the following:

MSS option With this option, the TCP sending the SYN announces its *maximum segment size*, the maximum amount of data that it is willing to accept in each TCP segment, on this connection. The sending TCP uses the receiver's MSS value as the maximum size of a segment that it sends.

Window scale option The maximum window that either TCP can advertise to the other TCP is 65,535, because the corresponding field in the TCP header occupies 16 bits. But, high-speed connections, common in today's Internet or long delay paths (satellite links) require a larger window to obtain the maximum throughput possible. This newer option specifies that the advertised window in the TCP header must be scaled (left shifted) by 0–14 bits, providing a maximum window of almost one gigabyte (65,535 x214). Both end-systems must support this option for the window scale to be used on a connection. To provide interoperability with older implementations that do not support this
option, the following rules apply. TCP can send the option with its SYN as part of an active open. But, it can scale its windows only if the other end also sends the option with its SYN. Similarly, the server's TCP can send this option only if it receives the option with the client's SYN. This logic assumes that implementations ignore options that they do not understand, which is required and common, but unfortunately, not guaranteed with all implementations.

Timestamp option. This option is needed for high-speed connections to prevent possible data corruption caused by old, delayed, or duplicated segments. Since it is a newer option, it is negotiated similarly to the window scale option.



	TCP Header																															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	Source Port																Destination Port															
32	Sequence Number																															
64	Acknowledgement Number																															
96	Data Offset				Reserved				CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
128	Checksum																Urgent Pointer															
160																																
192																																
224	**TCP Options**																															
256	* Field length is determined by Data Offset field value																															
288	* TCP Header 20 bytes without options, max 60 bytes with options																															
320	* Can begin on any byte boundary																															
352	* Must be padded with zeroes to make the header length a 4-byte multiple																															
384																																
416																																
448																																

Figure 1. TCP Header

WORKING OF THREE WAY HANDSHAKE

The three-way handshake begins with the initiator sending a TCP segment with the SYN control bit flag set.

TCP allows one side to establish a connection. The other side may either accept the connection or refuse it. If we consider this from application layer point of view, the side that is establishing the connection is the client and the side waiting for a connection is the server.

TCP identifies two types of OPEN calls:

Active Open. In an Active Open call a device (client process) using TCP takes the active role and initiates the connection by sending a TCP SYN message to start the connection.

Passive Open A passive OPEN can specify that the device (server process) is waiting for an active OPEN from a specific client. It does not generate any TCP message segment. The server processes listening for the clients are in Passive Open mode.

Three-way Handshake

Step 1. Device A (Client) sends a TCP segment with SYN = 1, ACK = 0, ISN (Initial Sequence Number) = 2000.

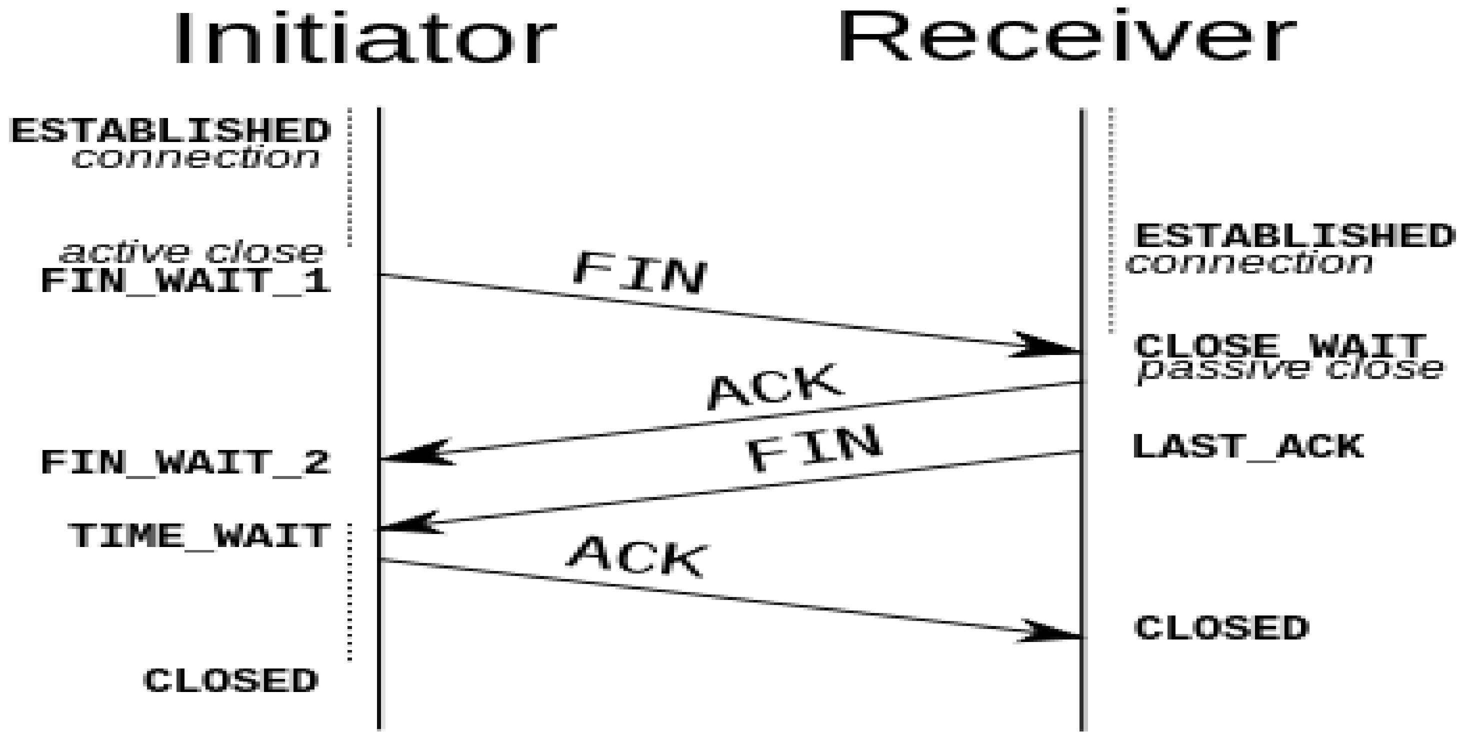
The Active Open device (Device A) sends a segment with the SYN flag set to 1, ACK flag set to 0 and an Initial Sequence Number 2000 (For Example), which marks the beginning of the sequence numbers for data that device A will transmit. SYN is short for SYNchronize. SYN flag announces an attempt to open a connection. The first byte transmitted to Device B will have the sequence number ISN+1.

Step 2. Device B (Server) receives Device A's TCP segment and returns a TCP segment with SYN = 1, ACK = 1, ISN = 5000 (Device B's Initial Sequence Number), Acknowledgment Number = 2001 (2000 + 1, the next sequence number Device B expecting from Device A).

Step 3. Device A sends a TCP segment to Device B that acknowledges receipt of Device B's ISN, With flags set as SYN = 0, ACK = 1, Sequence number = 2001, Acknowledgment number = 5001 (5000 + 1, the next sequence number Device A expecting from Device B)

This handshaking technique is referred to as the Three-way handshake or SYN, SYN-ACK, ACK.

After the three-way handshake, the connection is open and the participant computers start sending data using the sequence and acknowledge numbers.

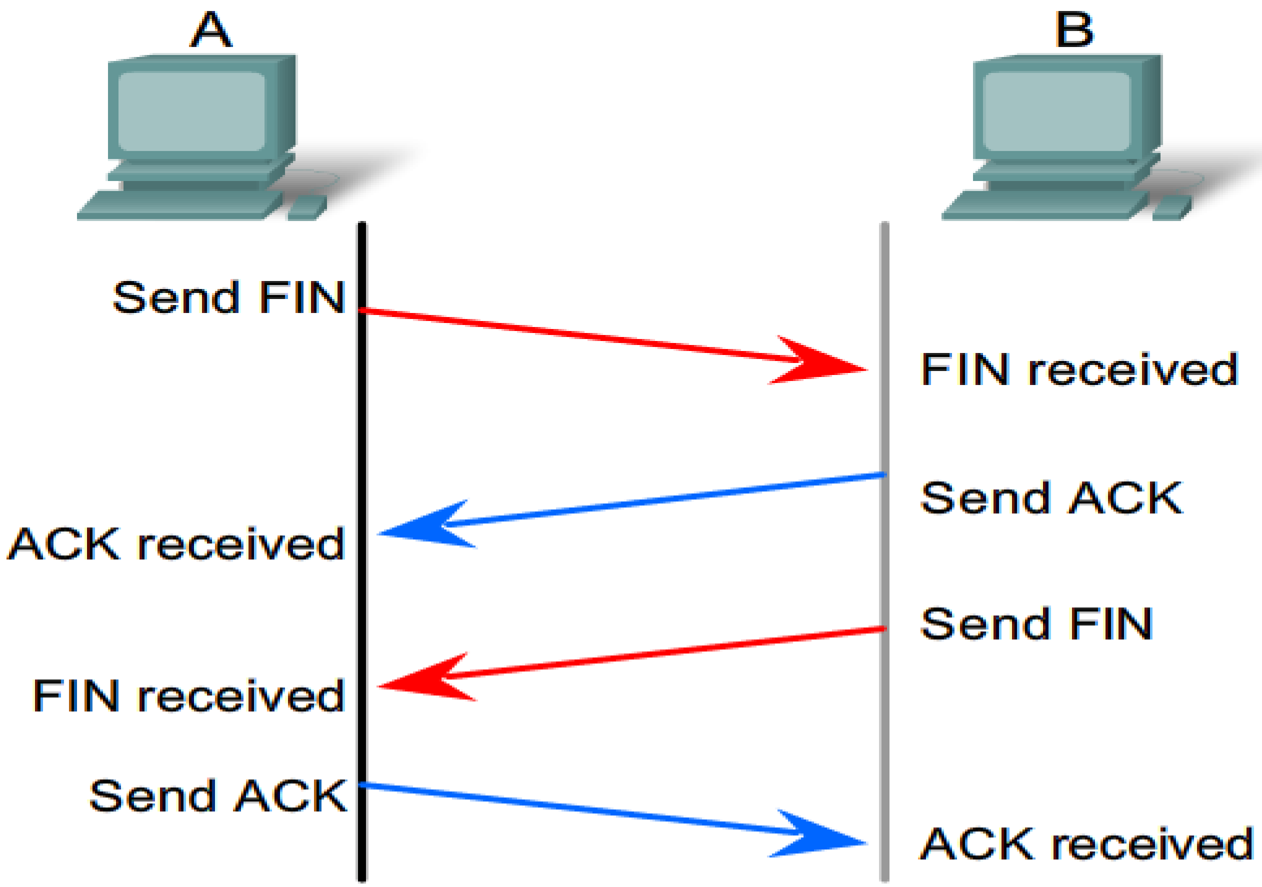


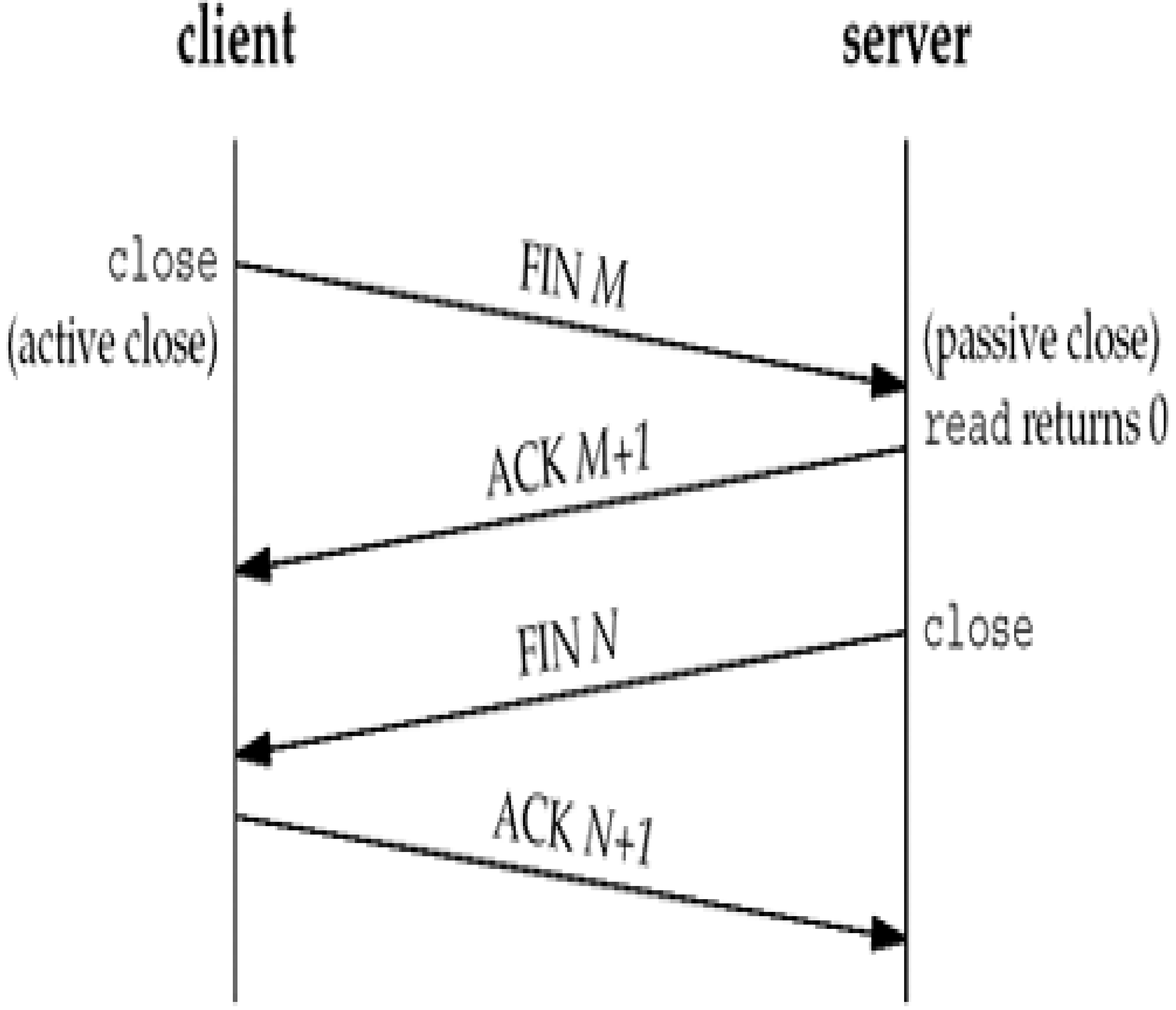
AN IMAGE OF THREE WAY HANDSHAKE

TCP CONNECTION TERMINATION

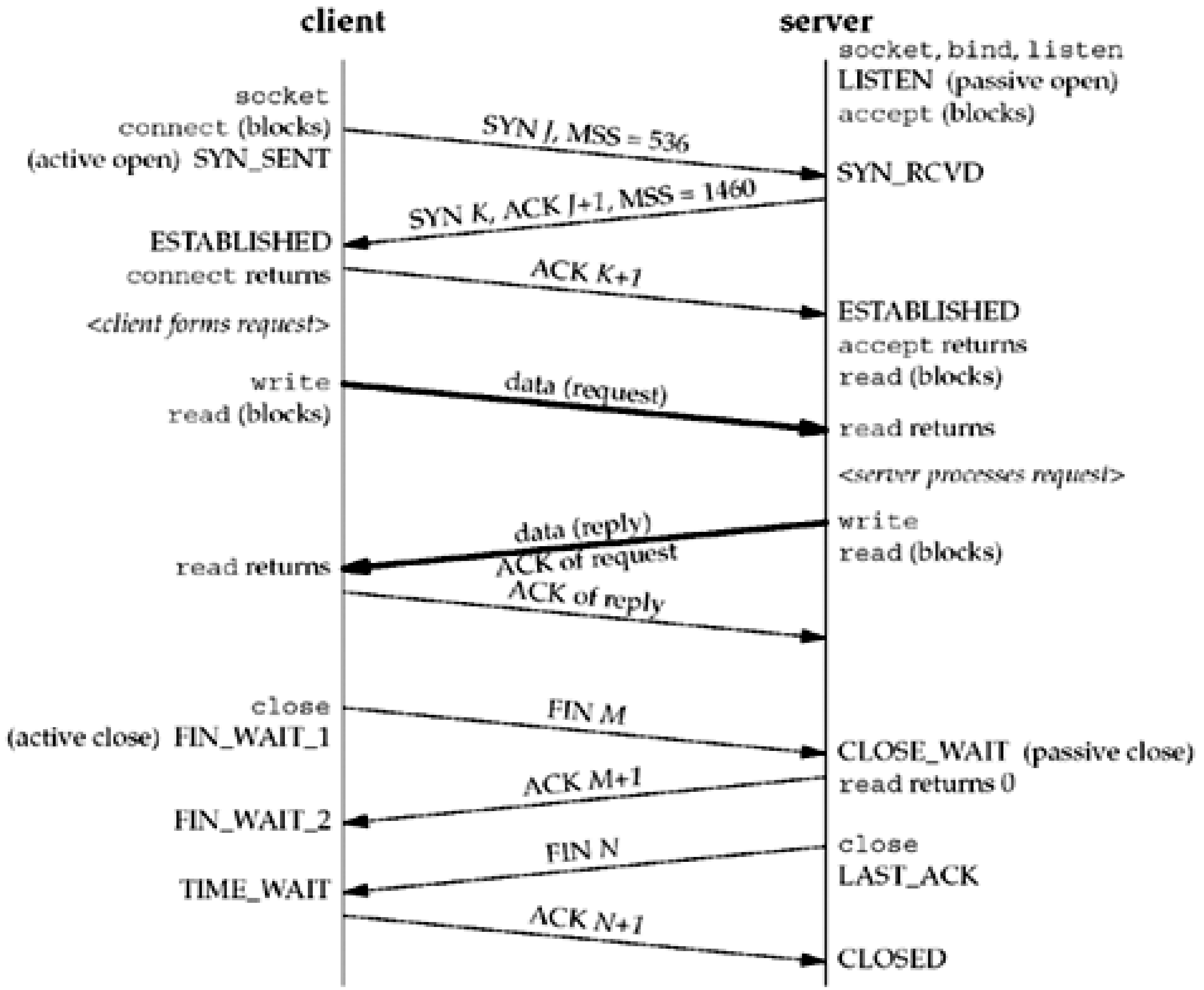
1. The connection termination phase uses a four-way handshake, with each side of the connection terminating independently.
2. When an endpoint wishes to stop its half of the connection, it transmits a FIN packet, which the other end acknowledges with an ACK.
3. Therefore, a typical tear-down requires a pair of FIN and ACK segments from each TCP endpoint. After both FIN/ACK exchanges are concluded, the side which sent the first FIN before receiving one waits for a timeout before finally closing the connection, during which time the local port is unavailable for new connections; this prevents confusion due to delayed packets being delivered during subsequent connections.
4. A connection can be "half-open", in which case one side has terminated its end, but the other has not. The side that has terminated can no longer send any data into the connection, but the other side can. The terminating side should continue reading the data until the other side terminates as well.
5. It is also possible to terminate the connection by a 3-way handshake, when host A sends a FIN and host B replies with a FIN & ACK (merely combines 2 steps into one) and host A replies with an ACK. This is perhaps the most common method.

It is possible for both hosts to send FINs simultaneously then both just have to ACK. This could possibly be considered a 2-way handshake since the FIN/ACK sequence is done in parallel for both directions.



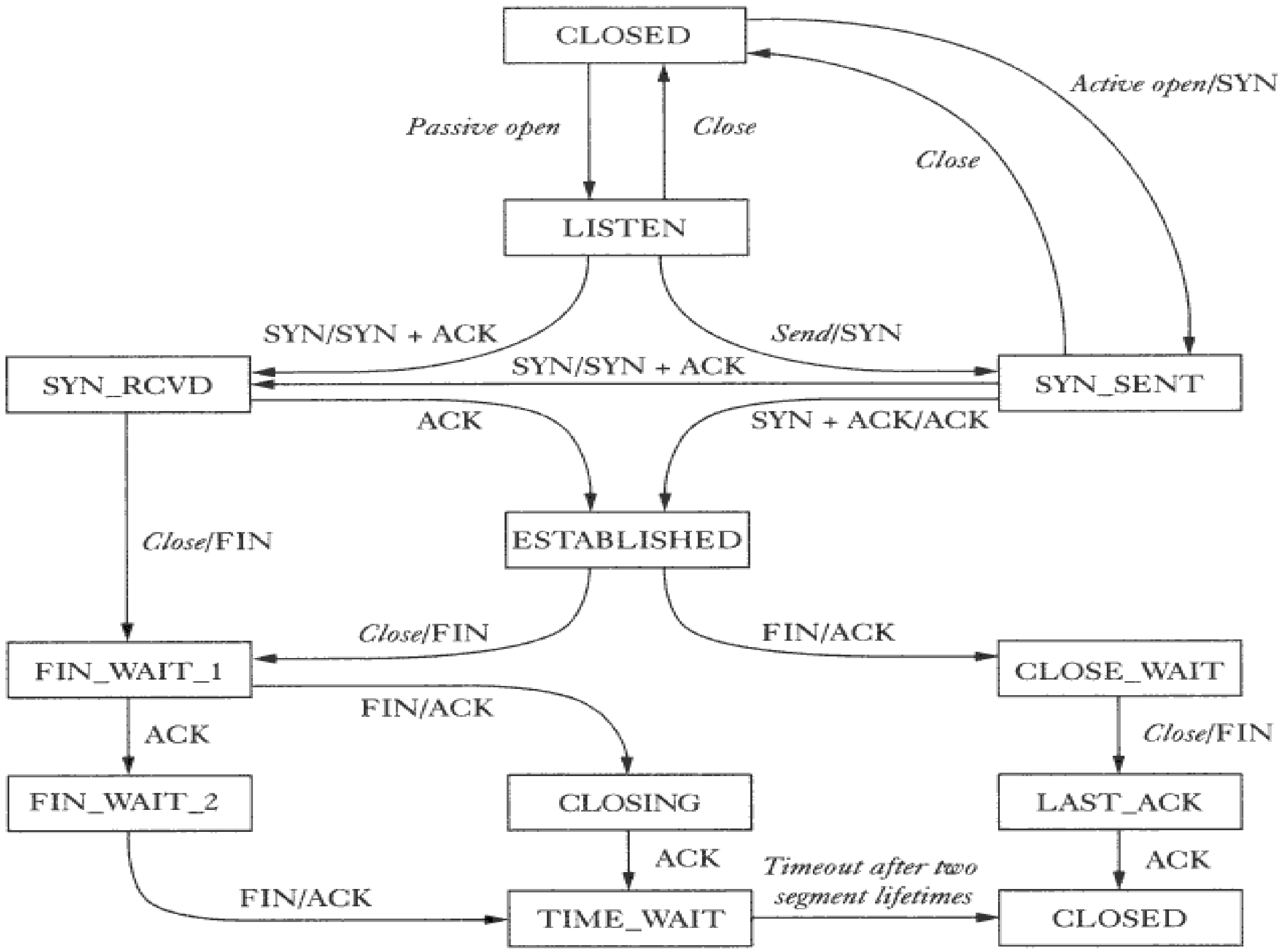

client
server
close
(active close)
FIN M
(passive close)
read returns 0
ACK M+1
close
FIN N
ACK N+1

TCP CONNECTION ESTABLSIHMENT , DATA TRANSFER AND CONNECTION TERMINATION



1. Once a connection is established, the client forms a request and sends it to the server. We assume this request fits into a single TCP segment (i.e., less than 1,460 bytes given the server's announced MSS).
2. The server processes the request and sends a reply, and we assume that the reply fits in a single segment (less than 536 in this example).
3. We show both data segments as bolder arrows. Notice that the acknowledgment of the client's request is sent with the server's reply.
4. This is called *piggybacking* and will normally happen when the time it takes the server to process the request and generate the reply is less than around 200 ms. If the server takes longer, say one second, we would see the acknowledgment followed later by the reply.

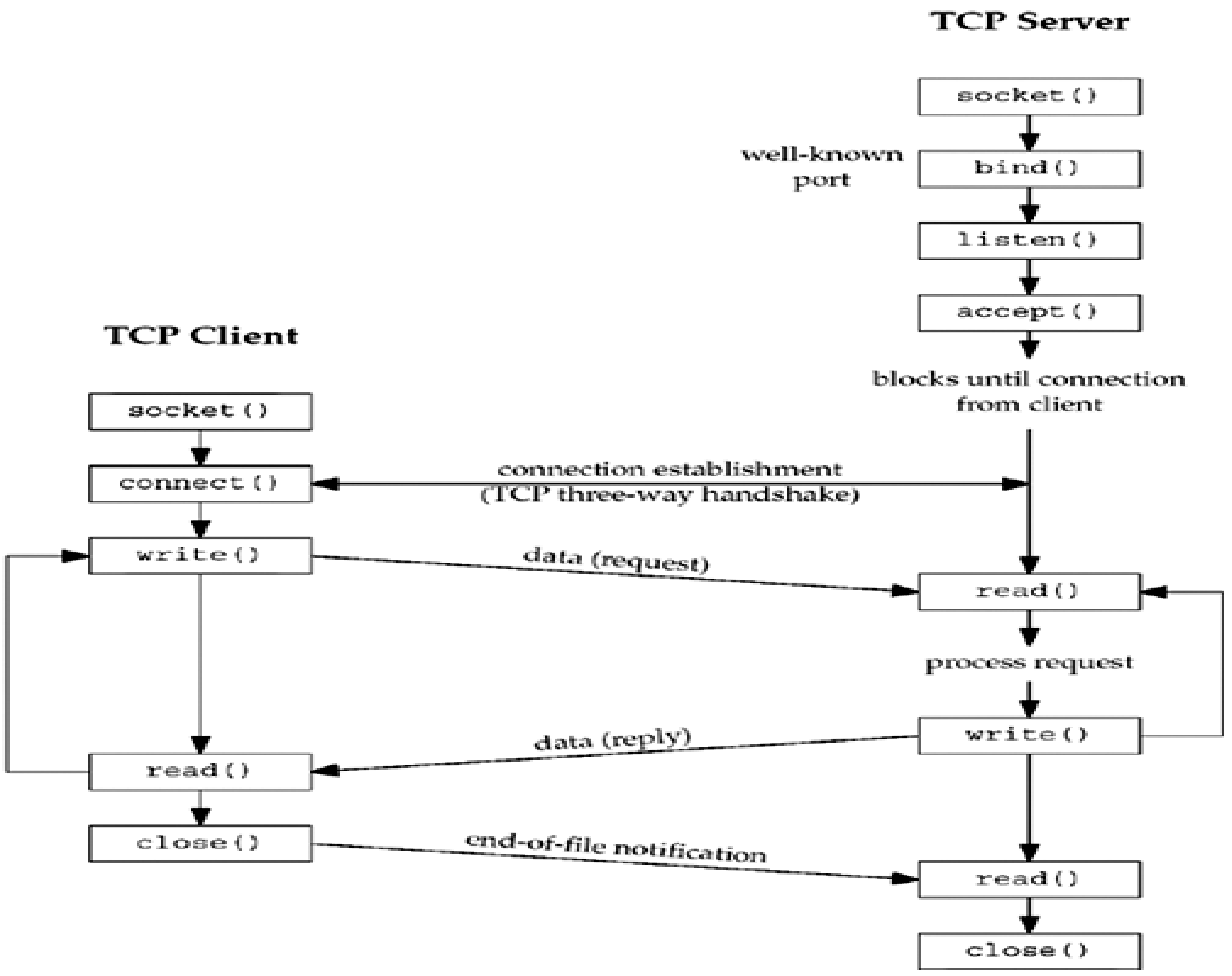
TCP STATE TRANSITION DIAGRAM



Advantages of TCP

- ☐ TCP guarantees three things: that your data gets there, that it gets there in order, and that it gets there without duplication. (the truth, the whole truth, and nothing but the truth...)
- ☐ TCP does Flow Control and Congestion Control
- ☐ Since its in the OS, handling incoming packets has fewer context switches from kernel to user space and back; all the reassembly, flow control, etc is done by the kernel.
- ☐ Routers may notice TCP packets and treat them specially. they can buffer and retransmit them
- ☐ TCP has good relative throughput on a modem or a LAN.

TCP CLIENT/SERVER



TCP SERVER MAIN() AND STR_ECHO()

```
#include "unp.h"
int
main(int argc, char **argv)
{
int listenfd, connfd;
pid_t childpid;
socklen_t clilen;
struct sockaddr_in cliaddr, servaddr;
listenfd = Socket (AF_INET, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
servaddr.sin_port = htons (SERV_PORT);
Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
Listen(listenfd, LISTENQ);
for ( ; ; ) {
clilen = sizeof(cliaddr);
connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
if ( (childpid = Fork()) == 0) { /* child process */
Close(listenfd); /* close listening socket */
str_echo(connfd); /* process the request */
exit (0);
}
Close(connfd); /* parent closes connected socket */
}
}
```

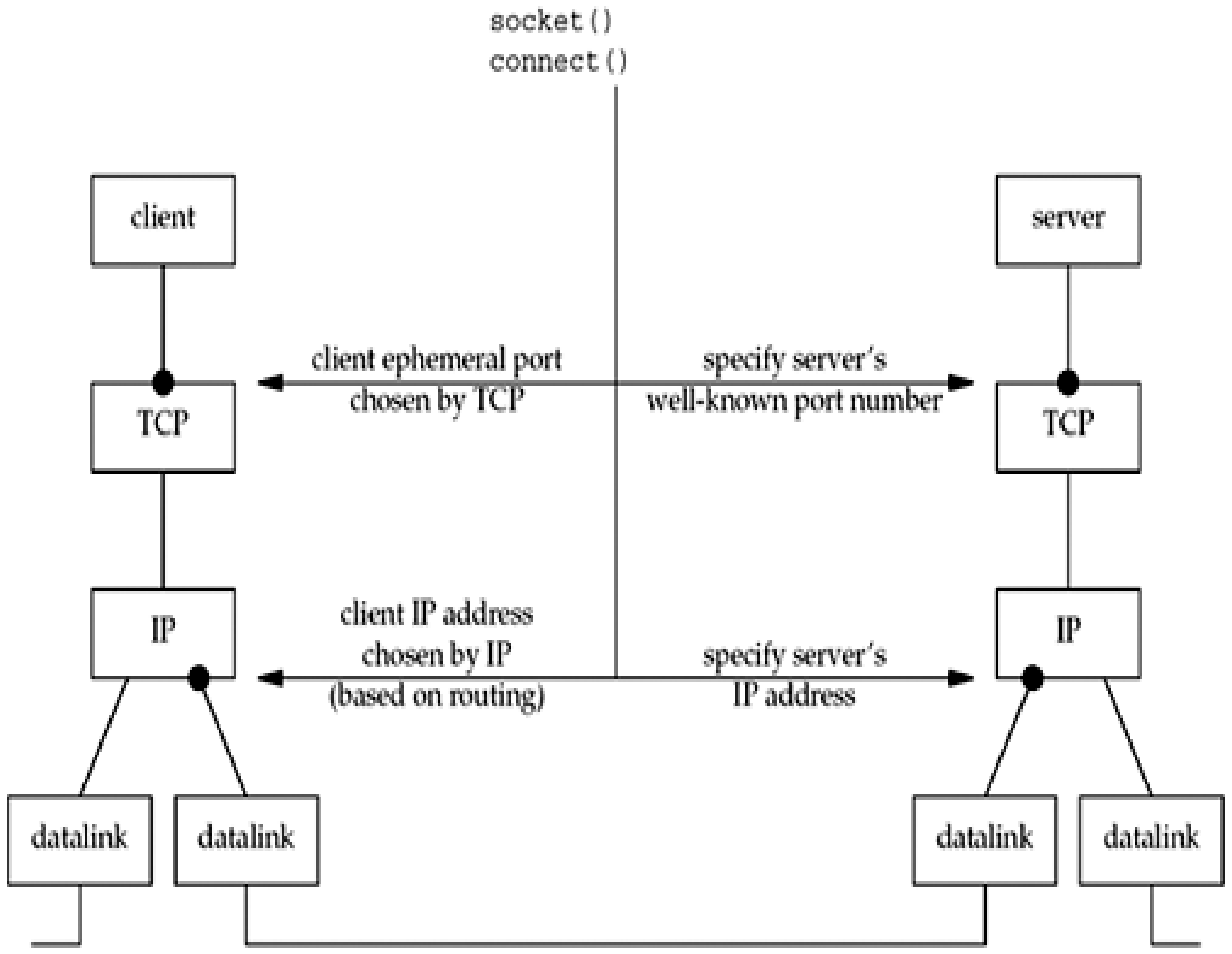
```
#include "unp.h"
void
str_echo(int sockfd)
{
ssize_t n;
char buf[MAXLINE];
again:
while ( (n = read(sockfd, buf, MAXLINE)) > 0)
Writen(sockfd, buf, n);
if (n < 0 && errno == EINTR)
goto again;
else if (n < 0)
err_sys("str_echo: read error");
}
```

TCP CLIENT MAIN() AND STR_CLI()

```
#include "unp.h"
int main(int argc, char **argv)
 { int sockfd;
struct sockaddr_in servaddr;
if (argc != 2)
err_quit("usage: tcpcli
<IPaddress>");
sockfd = Socket(AF_INET,
SOCK_STREAM, 0);
bzero(&servaddr,
sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port =
htons(SERV_PORT);
Inet_pton(AF_INET, argv[1],
&servaddr.sin_addr);
Connect(sockfd, (SA *)
&servaddr, sizeof(servaddr));
str_cli(stdin, sockfd); /* do it all */
exit(0); }
```

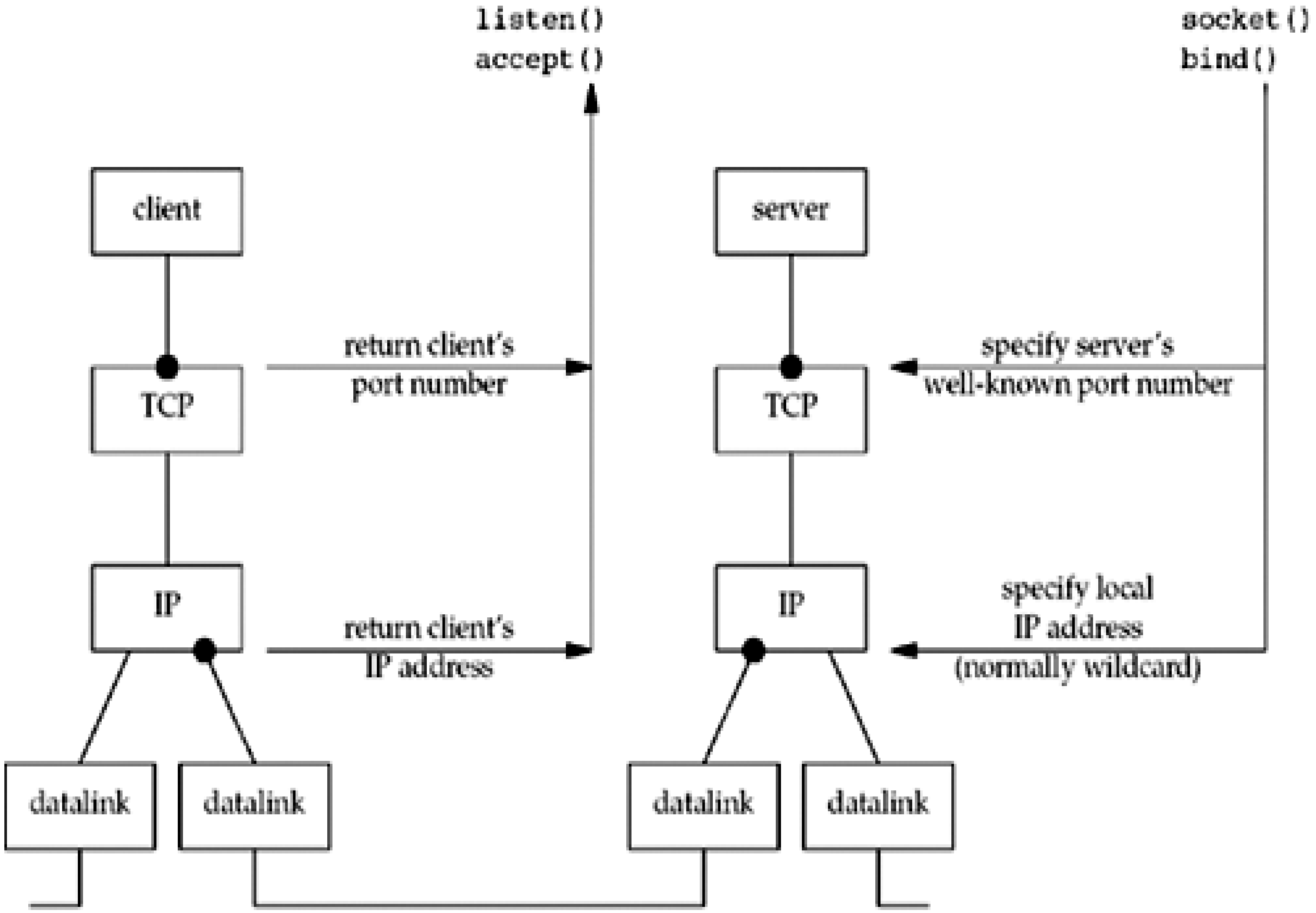
```
#include "unp.h"
void
str_cli(FILE *fp, int sockfd)
{
char sendline[MAXLINE],
recvline[MAXLINE];
while (Fgets(sendline, MAXLINE, fp)
!= NULL) {
Writen(sockfd, sendline, strlen
(sendline));
if (Readline(sockfd, recvline,
MAXLINE) == 0)
err_quit("str_cli: server terminated
prematurely");
Fputs(recvline, stdout);
}
}
```

CLIENT PRESPECTIVE


socket()
connect()
client
server
client ephemeral port chosen by TCP
specify server's well-known port number
TCP
TCP
IP
IP
client IP address chosen by IP (based on routing)
specify server's IP address
datalink
datalink
datalink
datalink

1. **Connect() The foreign IP address and foreign port must be specified by the client in the call to connect. The two local values are normally chosen by the kernel as part of the connect function.**
2. **Bind() The client has the option of specifying either or both of the local values, by calling bind before connect, but this is not common.**
3. **Getsockname() The client can obtain the two local values chosen by the kernel by calling getsockname after the connection is established**

SERVER PRESPECTIVE



listen()
accept()
socket()
bind()
client
server
TCP
TCP
return client's port number
specify server's well-known port number
IP
IP
return client's IP address
specify local IP address (normally wildcard)
datalink
datalink
datalink
datalink

1. Bind() The local port (the server's well-known port) is specified by bind. Normally, the server also specifies the wildcard IP address in this call.
2. Getsockname()If the server binds the wildcard IP address on a multihomed host, it can determine the local IP address by calling getsockname ()after the connection is established.
3. Accept()The two foreign values are returned to the server by accept.
4. Getpeername()If another program is execed by the server that calls accept, that program can call getpeername to determine the client's IP address and port, if necessary

REFERENCES

1. **Unix Network Programming Volume 1:The Sockets Networking API - Vol. 1** by Steavens/ Bill Fenner / Rudoff
2. **Unix Systems Programming** Paperback – 2008 by Kay Robbins
3. **Network Management: Principles and Practice, 2e** Paperback – 2010 by Subramanian
4. www.tcpguide.com
5. **Data Communications and Networking** Paperback – 1 Jul 2017 by Forouzan

THANK YOU