Network Programming

Compiled By:

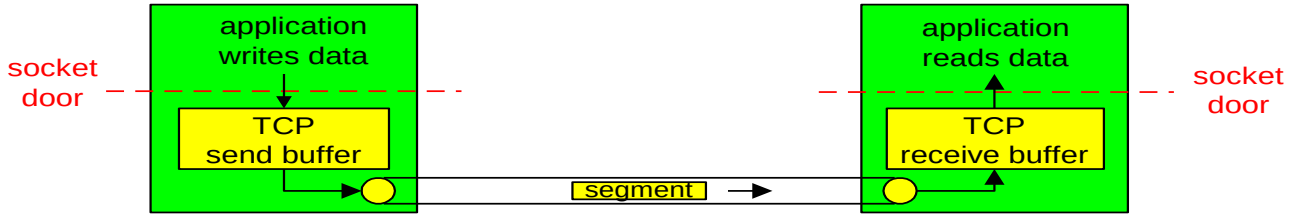
Madan Kadariya

NCIT

Transmission Control Protocol (TCP)

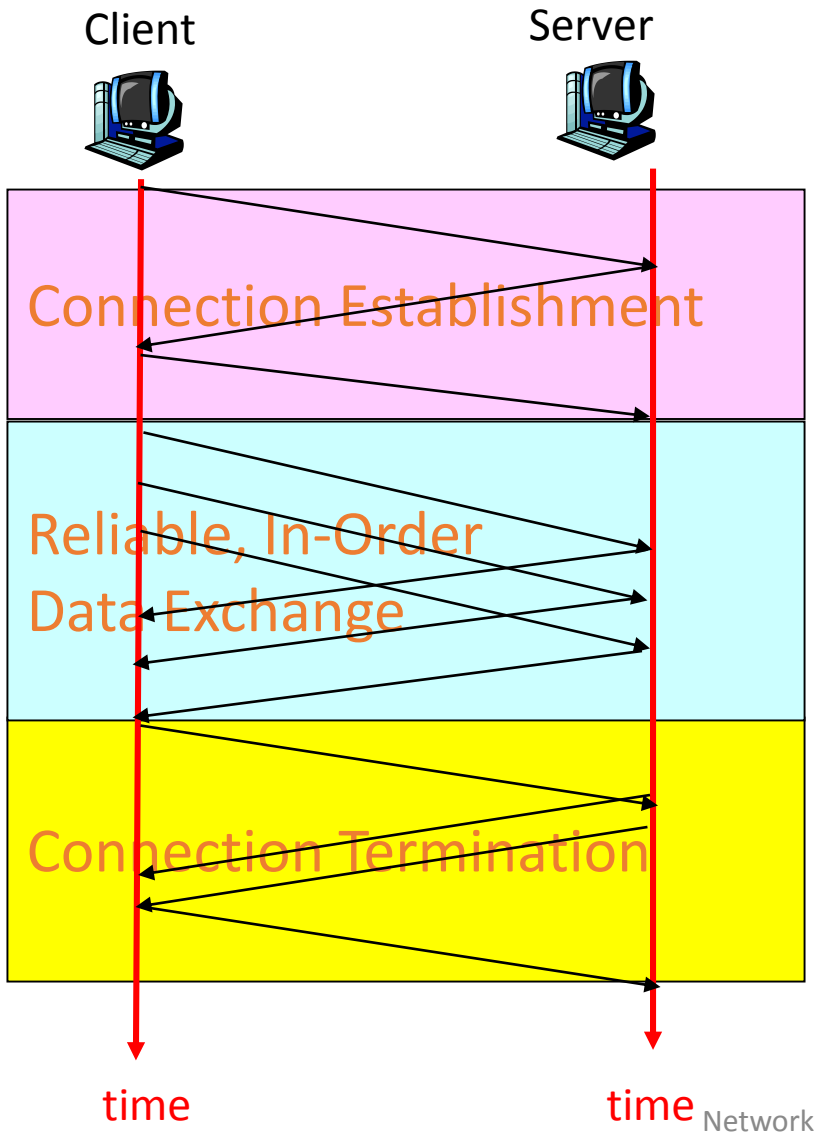
TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581



- point-to-point (unicast):
  - one sender, one receiver
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
  - State resides only at the END systems – Not a virtual circuit!
- full duplex data:
  - bi-directional data flow in same connection (A->B & B->A in the same connection)
  - MSS: maximum segment size
- reliable, in-order *byte steam:*
  - no "message boundaries"
- send & receive buffers
  - buffer incoming & outgoing data
- flow controlled:
  - sender will not overwhelm receiver
- congestion controlled:
  - sender will not overwhelm network

Typical TCP Transaction



- A TCP Transaction consists of 3 Phases
  1. Connection Establishment
     - Handshaking between client and server
  2. Reliable, In-Order Data Exchange
     - Recover any lost data through retransmissions and ACKs
  3. Connection Termination
     - Closing the connection

Protocol Comparison

Services/Features	SCTP	TCP	UDP
Connection-oriented	yes	yes	no
Full duplex	yes	yes	yes
Reliable data transfer	yes	yes	no
Partial-reliable data transfer	optional	no	no
Ordered data delivery	yes	yes	no
Unordered data delivery	yes	no	yes
Flow control	yes	yes	no
Congestion control	yes	yes	no
ECN capable	yes	yes	no
Selective ACKs	yes	optional	no
Preservation of message boundaries	yes	no	yes
Path MTU discovery	yes	yes	no
Application PDU fragmentation	yes	yes	no
Application PDU bundling	yes	yes	no
Multistreaming	yes	no	no
Multihoming	yes	no	no
Protection against SYN flooding attacks	yes	no	n/a
Allows half-closed connections	no	yes	n/a
Reachability check	yes	yes	no
Psuedo-header for checksum	no (uses vtags)	yes	yes
Time wait state	for vtags	for 4-tuple	n/a

TCP Connection Establishment

The following scenario occurs when a TCP connection is established.

1. The server must be prepared to accept an incoming connection. This is normally done by calling **socket, bind**, and **listen** and is called a passive open.
2. The client issues an active open by calling **connect**. This causes the client TCP to send a "**synchronize**" (**SYN**) segment, which tells the server the client's initial sequence number for the data that the client will send on the connection. Normally, there is no data sent with the **SYN**; it just contains an IP header, a TCP header, and possible TCP options.
3. The server must acknowledge (**ACK**) the client's **SYN** and the server must also send its own **SYN** containing the initial sequence number for the data that the server will send on the connection. The server sends its **SYN** and the **ACK** of the client's **SYN** is a single segment.
4. The client must acknowledge the server's SYN.

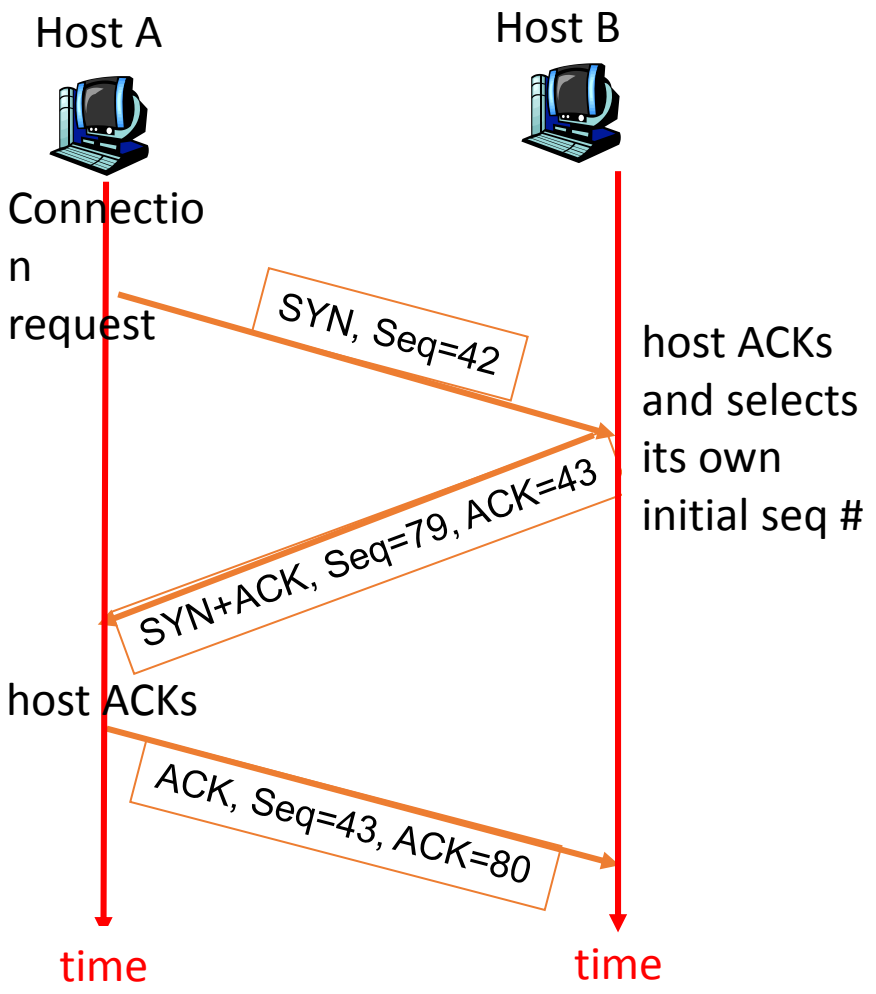
Connection Establishment (cont)

Three way handshake:

Step 1: client host sends TCP SYN segment to server
- specifies a random initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data



Three-way handshake

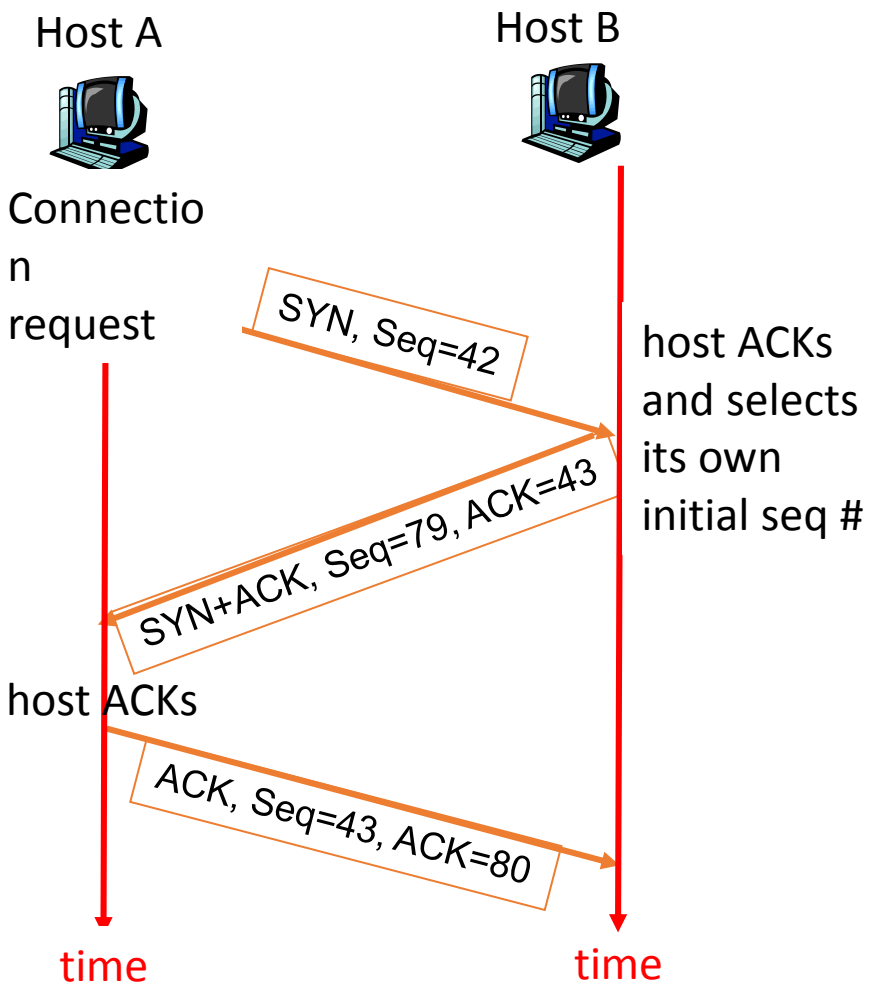
Connection Establishment (cont)

Seq. #'s:

- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK



Three-way handshake

TCP Starting Sequence Number Selection

- Why a random starting sequence #? Why not simply choose 0?
  - To protect against two incarnations of the same connection reusing the same sequence numbers too soon
  - That is, while there is still a chance that a segment from an earlier incarnation of a connection will interfere with a later incarnation of the connection
- How?
  - Client machine seq #0, initiates connection to server with seq #0.
  - Client sends one byte and client machine crashes
  - Client reboots and initiates connection again
  - Server thinks new incarnation is the same as old connection

TCP Connection Termination

1. One application calls close first, and we say that this end performs the active close. This end's TCP sends a FIN segment, which means it is finished sending data.
2. The other end that receives the FIN performs the passive close. The received FIN is acknowledged by TCP. The receipt of the FIN is also passed to the application as an end-of-file, since the receipt of the FIN means the application will not receive any additional data on the connection.
3. Sometime later, the application that received the end-of-file will close its socket. This causes its TCP to send a FIN.
4. The TCP on the system that receives this final FIN (the end that did the active close) acknowledges the FIN.

TCP Connection Termination

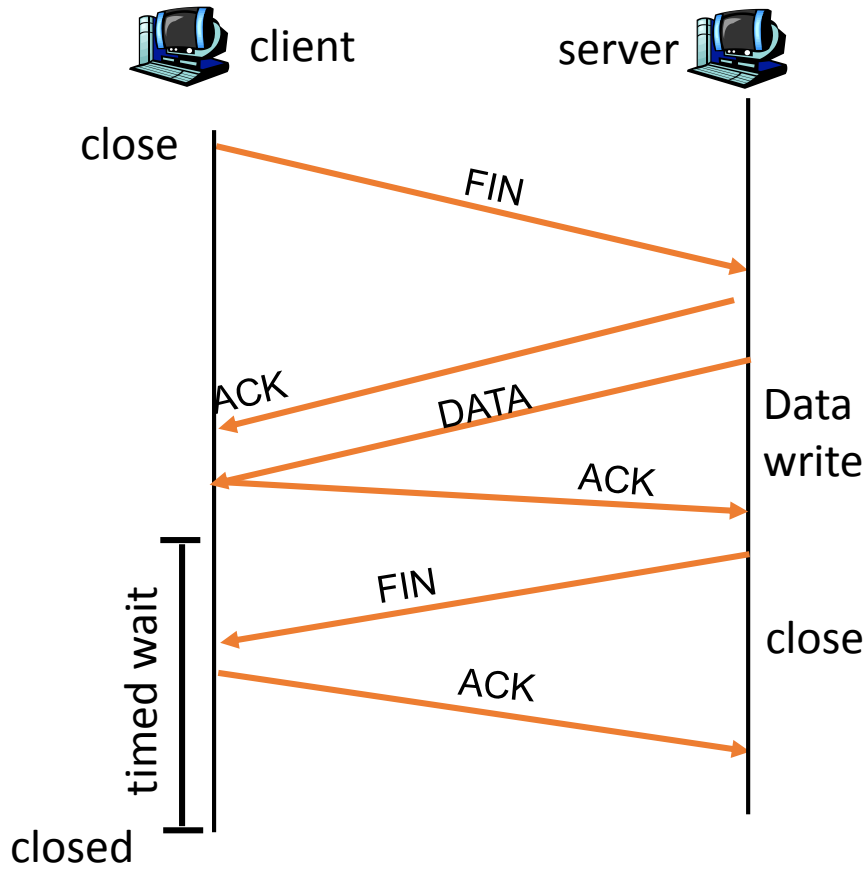
Closing a connection:

client closes socket:

```
clientSocket.close();
```

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Server might send some buffered but not sent data before closing the connection. Server then sends FIN and moves to Closing state.



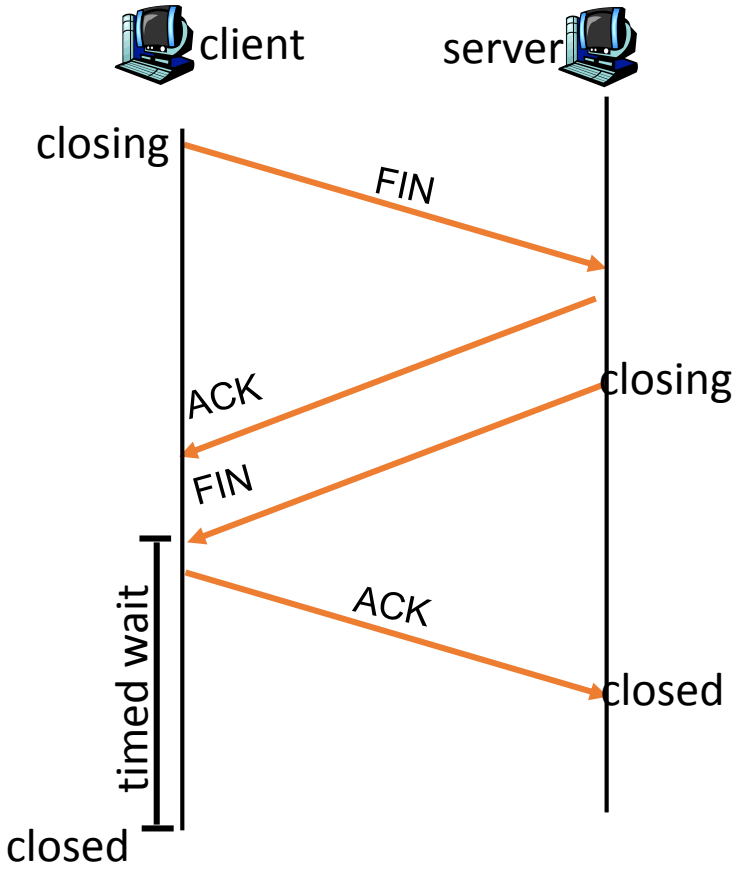
TCP Connection Termination

Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

- Why wait before closing the connection?
  - If the connection were allowed to move to CLOSED state, then another pair of application processes might come along and open the same connection (use the same port #s) and a delayed FIN from an earlier incarnation would terminate the connection.

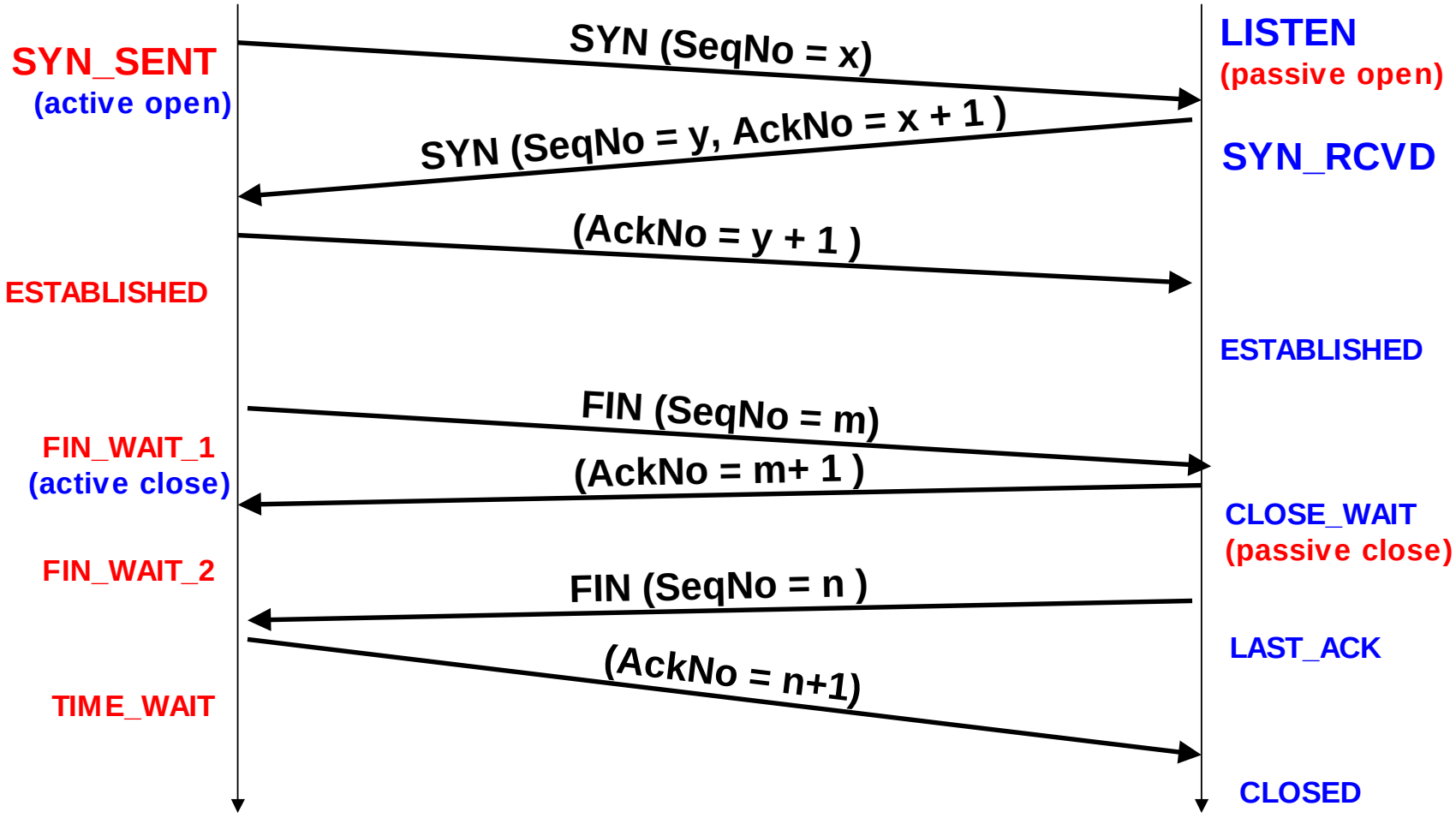


TCP/IP State Transition Diagram

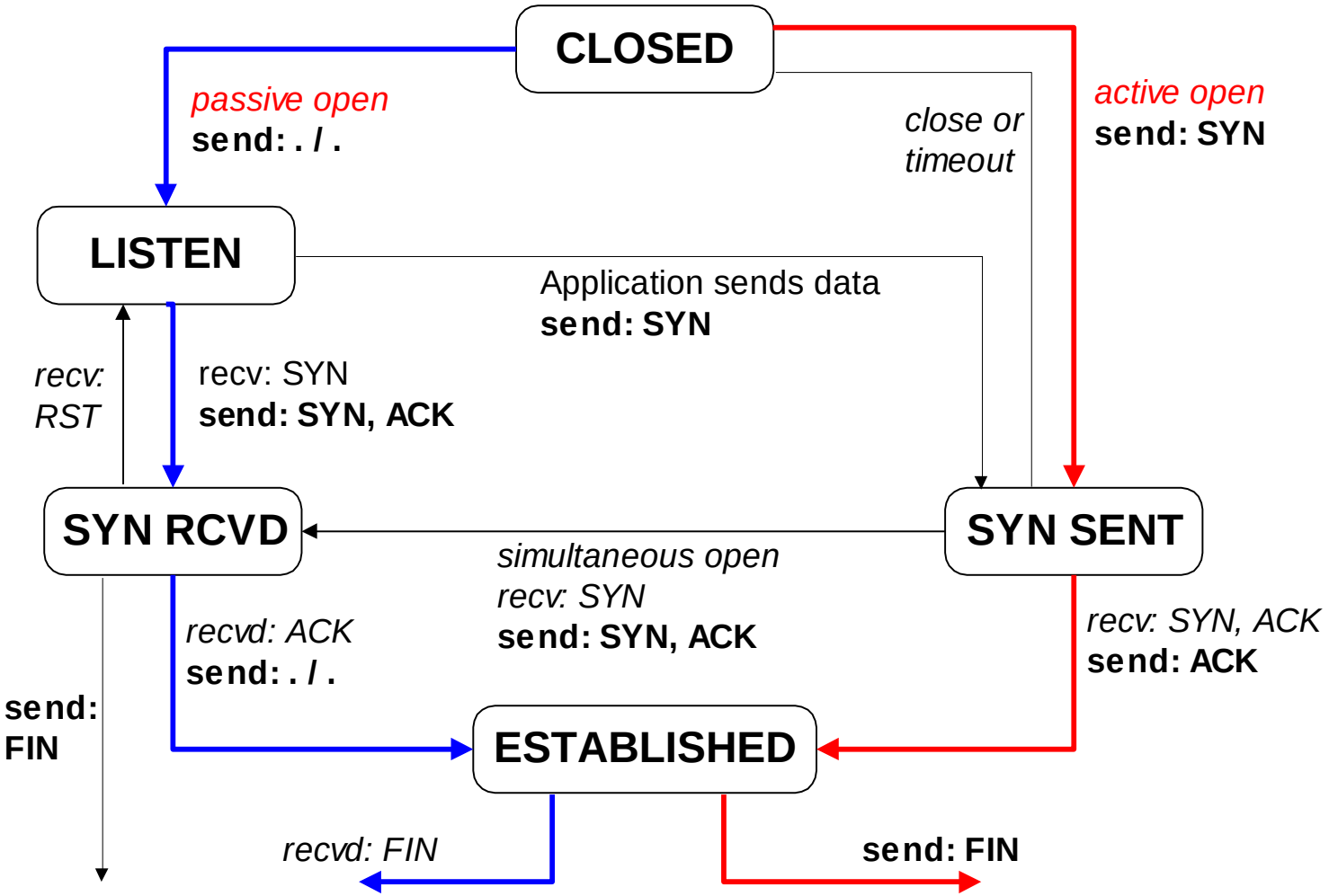
TCP States

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for Ack
SYN SENT	The client has started to open a connection
ESTABLISHED	Normal data transfer state
FIN WAIT 1	Client has said it is finished
FIN WAIT 2	Server has agreed to release
TIMED WAIT	Wait for pending packets (“2MSL wait state”)
CLOSING	Both Sides have tried to close simultanesously
CLOSE WAIT	Server has initiated a release
LAST ACK	Wait for pending packets

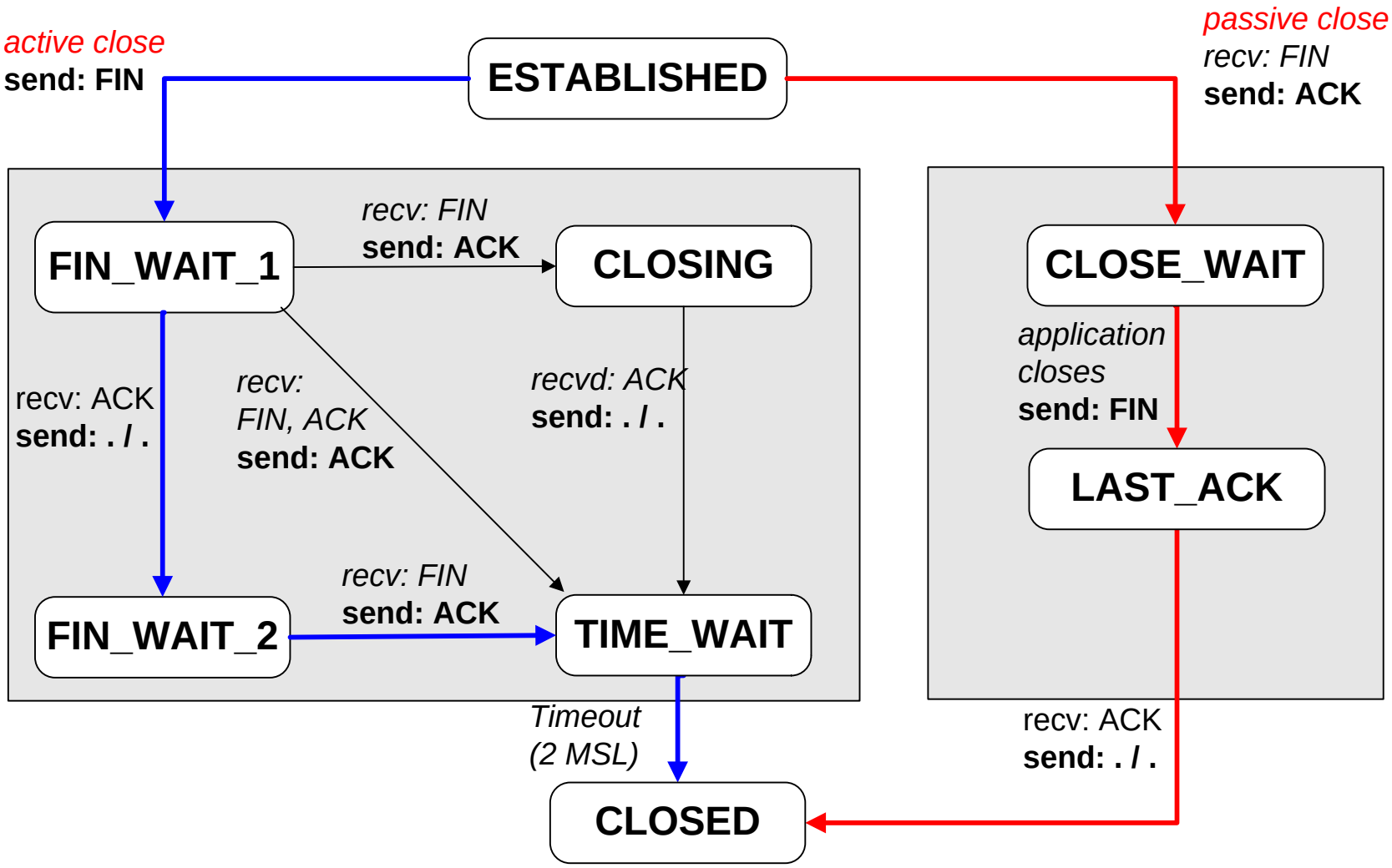
TCP States in “Normal” Connection Lifetime



TCP State Transition Diagram Opening A Connection



TCP State Transition Diagram Closing A Connection



2MSL Wait State

2MSL Wait State = TIME_WAIT

- When TCP does an active close, and sends the final ACK, the connection **must stay in in the TIME_WAIT state for twice the maximum segment lifetime**.

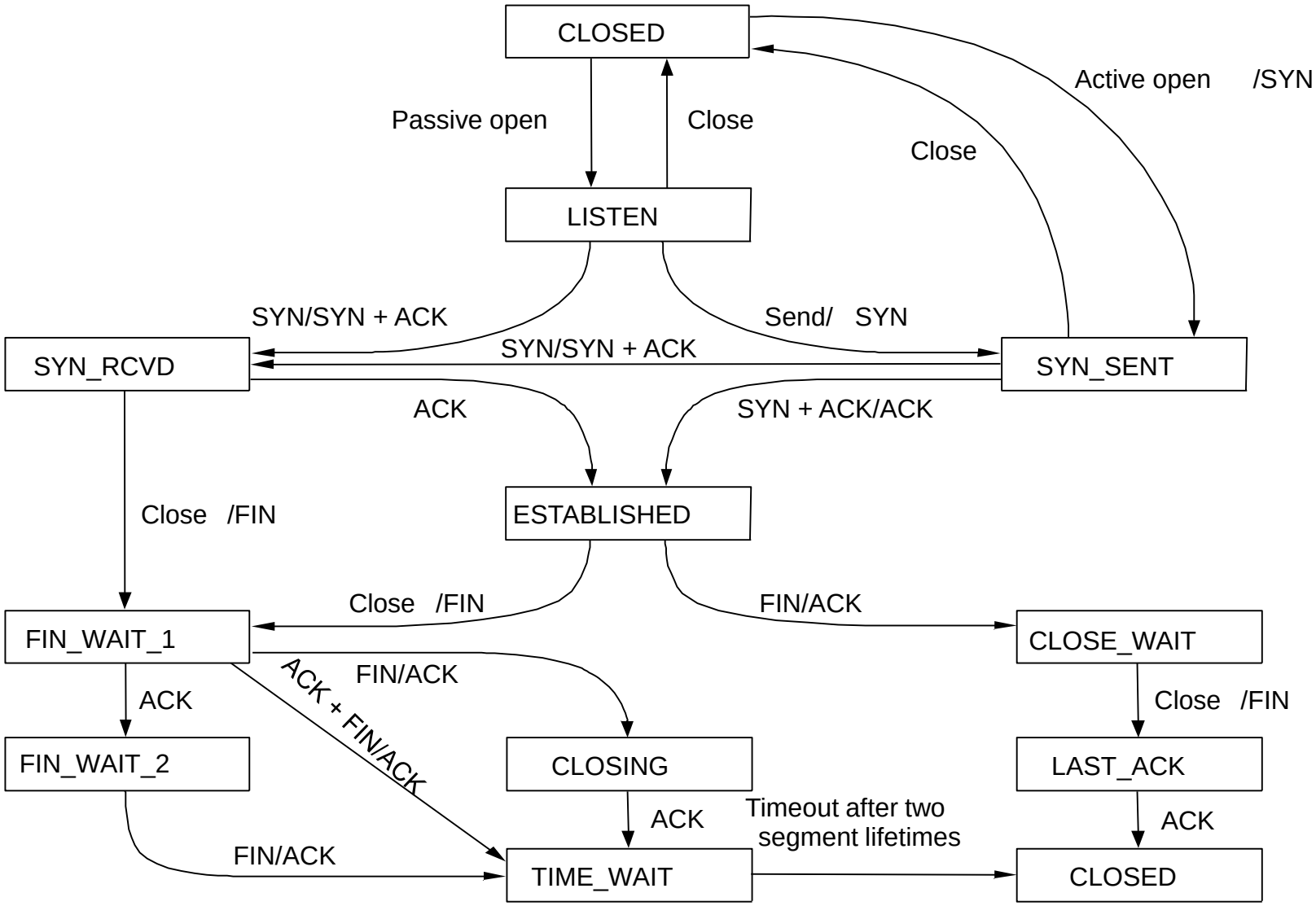
2MSL= 2 * Maximum Segment Lifetime

- Why?
  TCP is given a chance to resent the final ACK. (Server will timeout after sending the FIN segment and resend the FIN)
- The MSL is set to 2 minutes or 1 minute or 30 seconds.

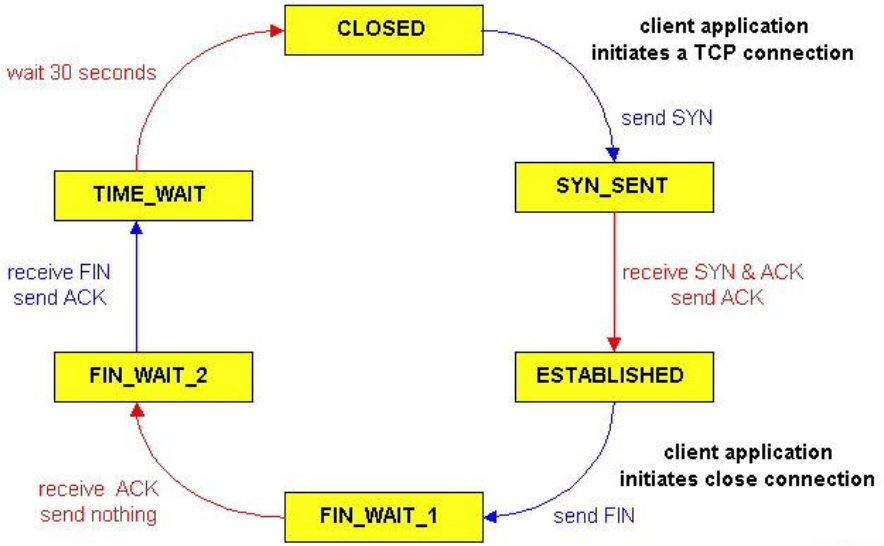
Resetting Connections

- Resetting connections is done by setting the RST flag
- **When is the RST flag set?**
  - Connection request arrives and no server process is waiting on the destination port
  - Abort (Terminate) a connection
    Causes the receiver to throw away buffered data. Receiver does not acknowledge the RST segment

TCP State-Transition Diagram

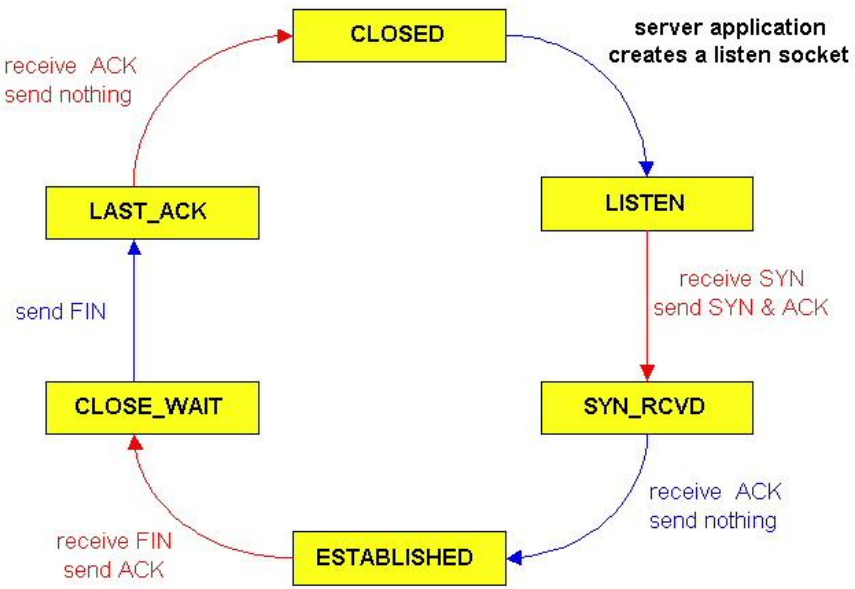


Typical TCP Client/Server Transitions



TCP client lifecycle

TCP server lifecycle

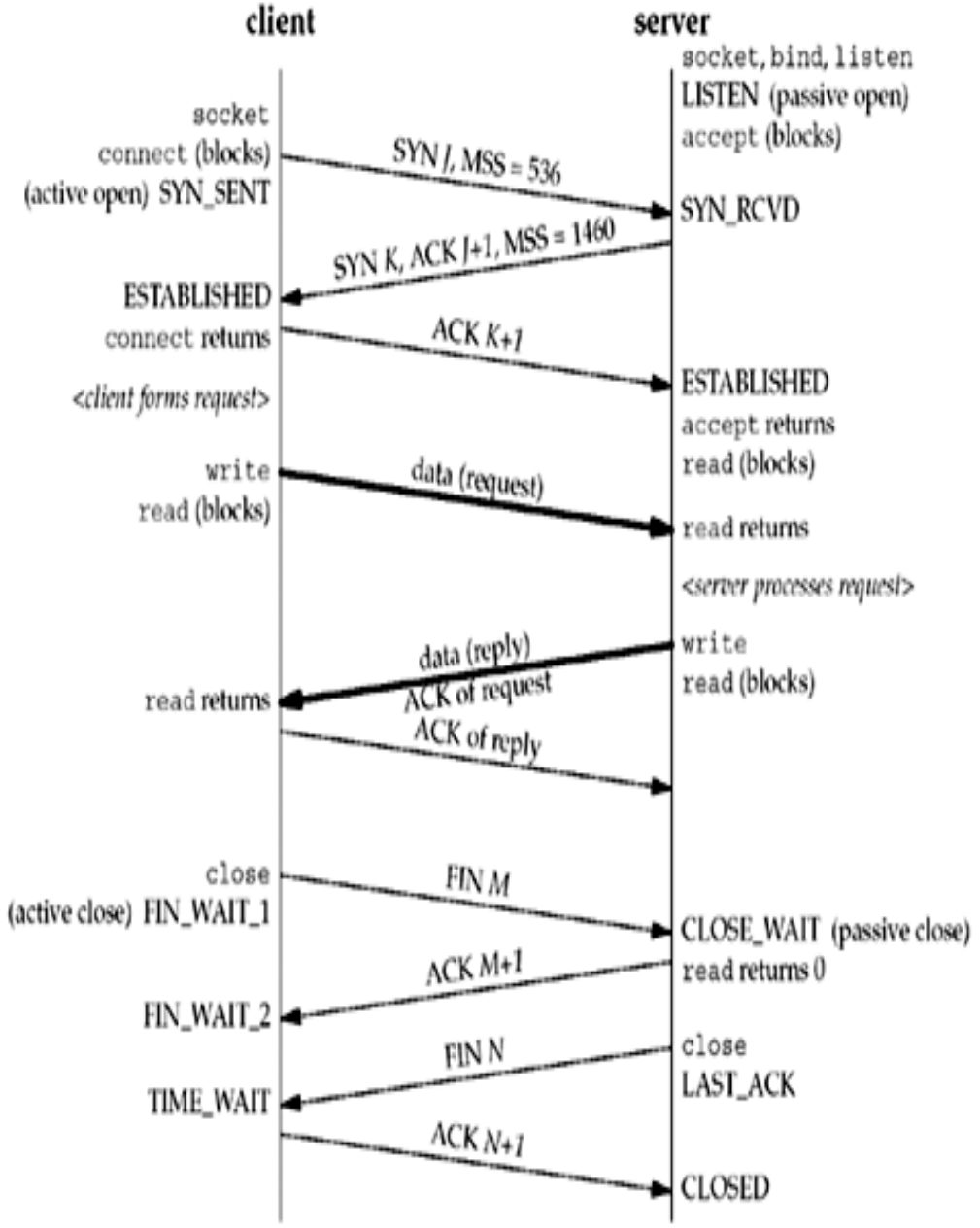


Watching the Packets

The client in this example announces an MSS of 536 (**minimum reassembly buffer size**) and the server announces an MSS of 1,460 (typical for IPv4 on an Ethernet). It is okay for the MSS to be different in each direction. The acknowledgment of the client's request is sent with the server's reply. 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.

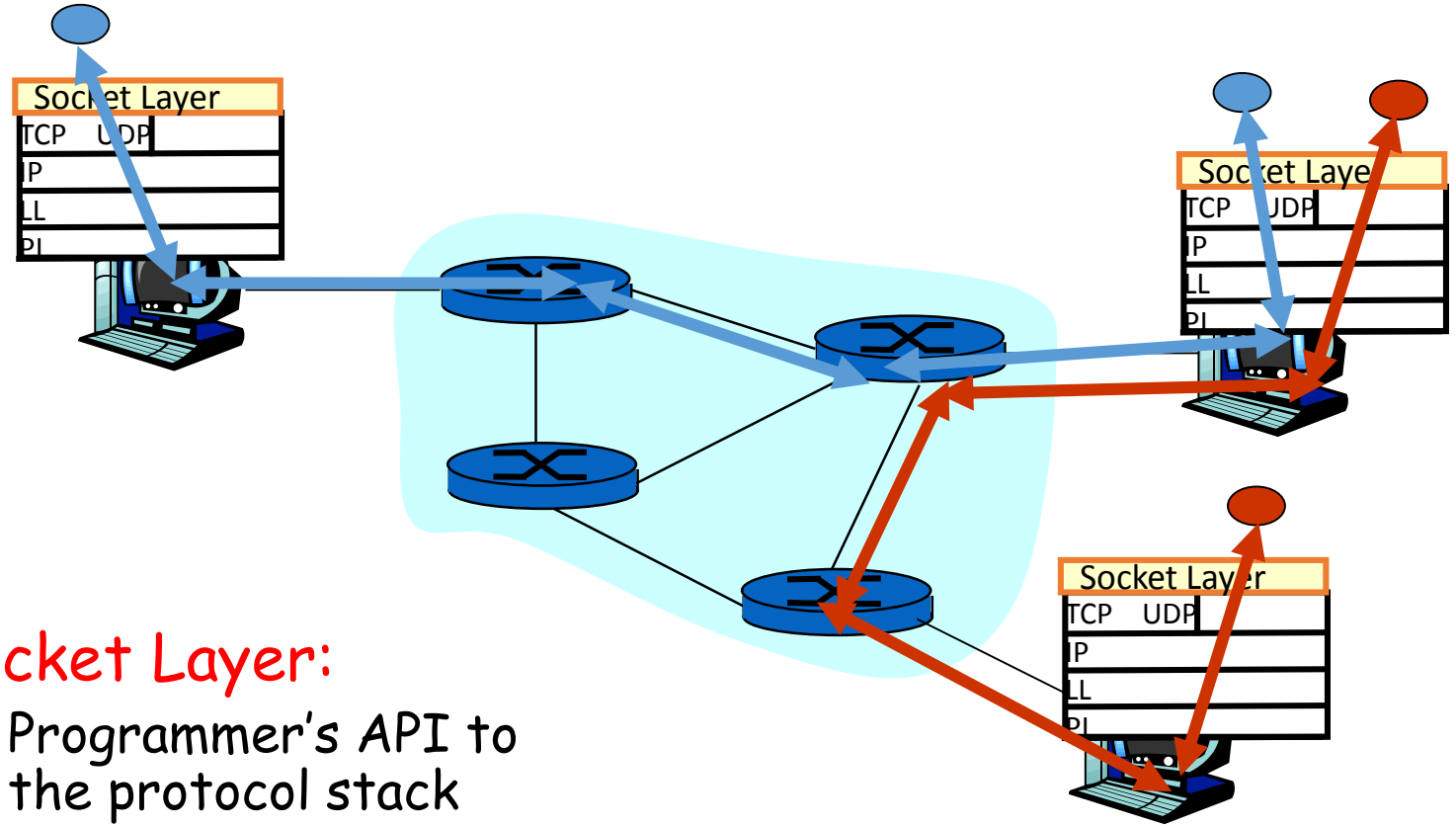
With TCP, there would be eight segments of overhead. If UDP was used, only two packets would be exchanged. UDP removes all the reliability that TCP provides to the application.

UDP avoids the overhead of TCP connection establishment and connection termination.



Socket

How to program using the TCP?



- **Socket Layer:**
  - Programmer's API to the protocol stack
- Typical network app has two pieces: *client* and *server*
- **Server:** Passive entity. Provides service to clients
  - e.g., Web server responds with the requested Web page
- **Client:** initiates contact with server ("speaks first")
  - typically requests service from server, e.g., Web Browser

Socket Creation

- mySock = socket(family, type, protocol);
- UDP/TCP/IP-specific sockets

	Family	Type	Protocol
TCP	PF_INET	SOCK_STREAM	IPPROTO_TCP
UDP		SOCK_DGRAM	IPPROTO_UDP

- Socket reference
  - File (socket) descriptor in UNIX
  - Socket handle in WinSock

TCP Client/Server Interaction

Server starts by getting ready to receive client connections…

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Create socket for incoming connections */
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
    a. Accept new connection
    b. Communicate
    c. Close the connection

TCP Client/Server Interaction

```
 echoServAddr.sin_family = AF_INET;                 /* Internet address family */
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY);/* Any incoming interface */
echoServAddr.sin_port = htons(echoServPort);         /* Local port */

if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
   DieWithError("bind() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
 /* Mark the socket so it will listen for incoming connections */
if (listen(servSock, MAXPENDING) < 0)
   DieWithError("listen() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
for (;;) /* Run forever */
{
   clntLen = sizeof(echoClntAddr);

   if ((clntSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen)) < 0)
      DieWithError("accept() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

Server is now blocked waiting for connection from a client

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

Later, a client decides to talk to the server...

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Create a reliable, stream socket using TCP */
if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
    a. Accept new connection
    b. Communicate
    c. Close the connection

TCP Client/Server Interaction

```
 echoServAddr.sin_family      = AF_INET;                /* Internet address family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP);   /* Server IP address */
echoServAddr.sin_port        = htons(echoServPort);    /* Server port */

if (connect(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
   DieWithError("connect() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
echoStringLen = strlen(echoString);          /* Determine input length */

/* Send the string to the server */
  if (send(sock, echoString, echoStringLen, 0) != echoStringLen)
     DieWithError("send() sent a different number of bytes than expected");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Receive message from client */
if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
   DieWithError("recv() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

close(sock);

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

close(clntSocket)

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Tidbits

- Client knows server address and port
- No correlation between `send()` and `recv()`

Client	Server
send(“Hello Bob”)	
	recv() -> “Hello ”
	recv() -> “Bob”
	send(“Hi ”)
	send(“Jane”)
recv() -> “Hi Jane”	