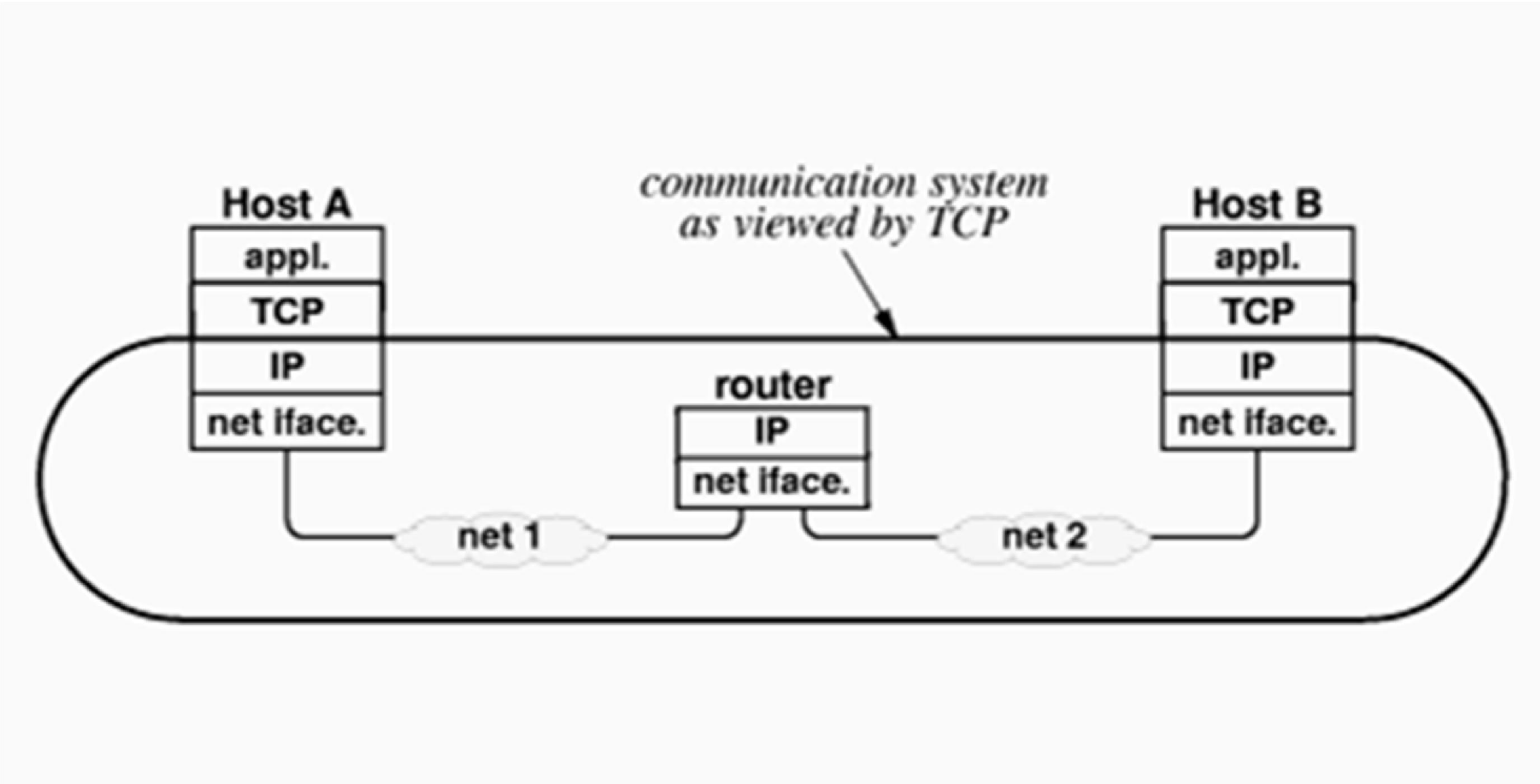
# Where TCP and IP Operate



# The complete TCP Transport Service offering

- Recall that the TCP Transport Service has the following characteristics:
  - ***Connection Orientation***: This aspect has been examined through a variety of lab exercises.
  - ***Point-To-Point Communication***: Each TCP connection has exactly <u>two</u> endpoints. Also examined through the lab exercises.
  - ***Complete Reliability***: TCP guarantees that the data will be delivered exactly as sent i.e. no data missing or out of sequence. To be examined later.
  - ***Full Duplex Communication***: A TCP connection allows data to flow in either direction
    - TCP buffers outgoing and incoming data (recall **RECVQ** and **SENDQ** buffers) allowing applications to continue executing other code whilst the data is being transferred.

# The complete TCP Transport Service offering

- ***Stream Interface***: The <u>source</u> application sends a *continuous* sequence of octets across a connection
  - The data is passed *en bloc* to TCP for delivery. Recall use of ***send()*** primitive containing a pointer to the local App-buffer.
  - TCP does not guarantee to deliver the data in the same size pieces that it was transferred by the source application. Recall use of ***recv()*** primitive which is always placed inside a ***while()*** structure.
- ***Reliable Connection Startup***: TCP both applications to <u>agree</u> to any new connection. To be examined.
- ***Graceful Connection Shutdown***: Either can request a connection to be shut down. To be examined.

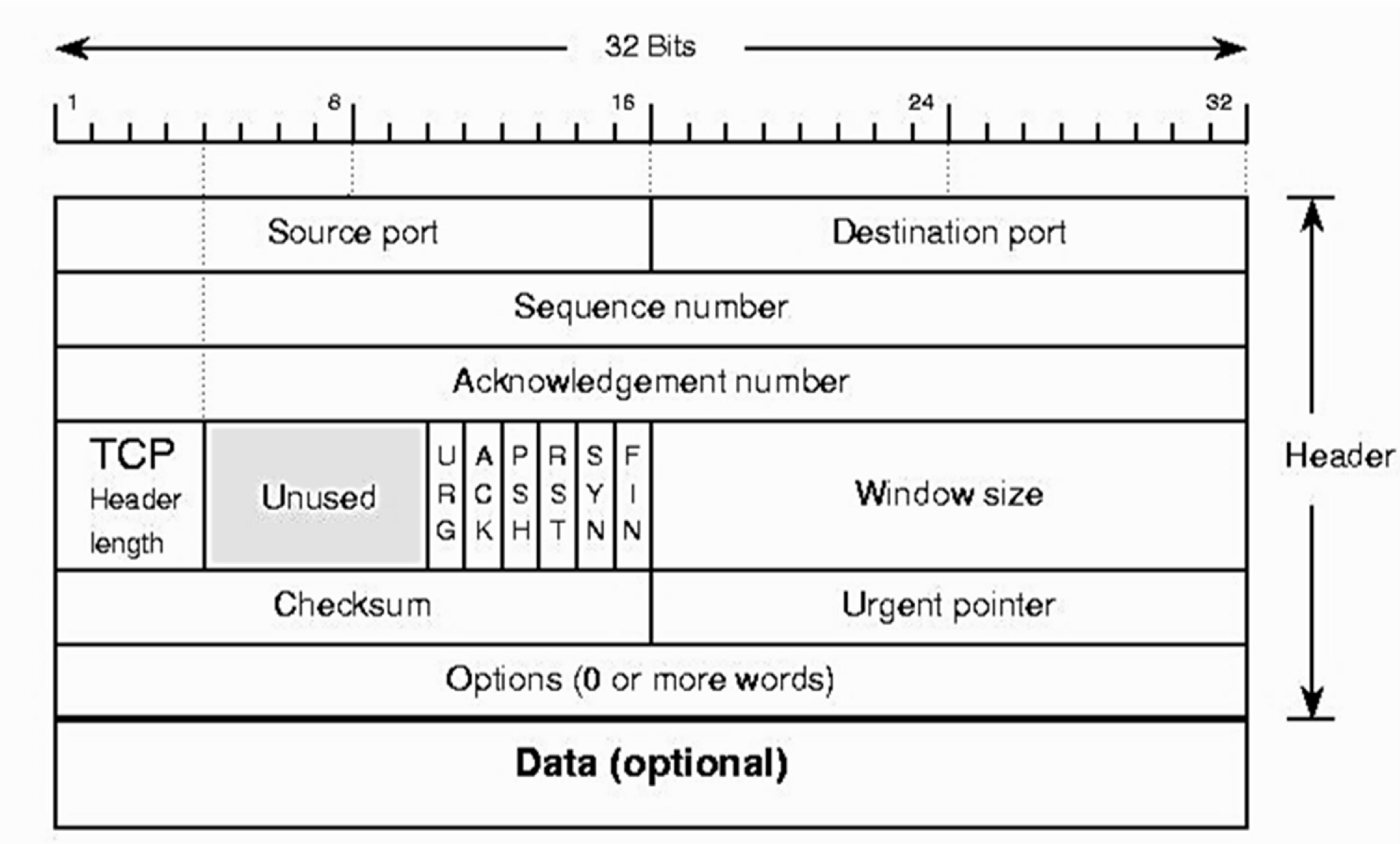
# TCP Connections

- Applications that need to exchange must have a connection with each other.
- Each application interacts with its <u>local</u> TCP entity, via its <u>local</u> socket, to manage connection *Establishment* and *Termination* activities:
  - These activities relate to Phase 1 (Initialisation) and Phase 3 (Termination) of the communications model.
  - These activities are commonly referred to as *Opening* and *Closing* connections.
  - TCP plays a vital role in these activities.

# TCP Connections

- To facilitate an exploration of TCP’s role in *Opening* and *Closing* connections, it is important to explore the structure of a TCP PDU a.k.a. a ***Segment***:
  - The next slide shows the structure of a TCP segment.

# TCP Segment Header Format



# Opening a TCP connection: **The Three-Way Handshake**

- Opening a TCP connection:
  - A server calling *socket*, *bind*, and *listen* is performing a *passive* open
  - A client calling *connect* is performing an *active* open
- The call to *connect* causes the client TCP entity to send a SYN (*synchronize*) segment:
  - This segment contains the client's initial <u>sequence</u> number for its data
- The server TCP entity must acknowledge receiving a SYN segment and it must send its own SYN segment:
  - This contains the initial sequence number for its data (note: the full duplex operation)
  - In addition the server's SYN contains an ACK of the client's SYN message

# The Three-Way Handshake – Contd.

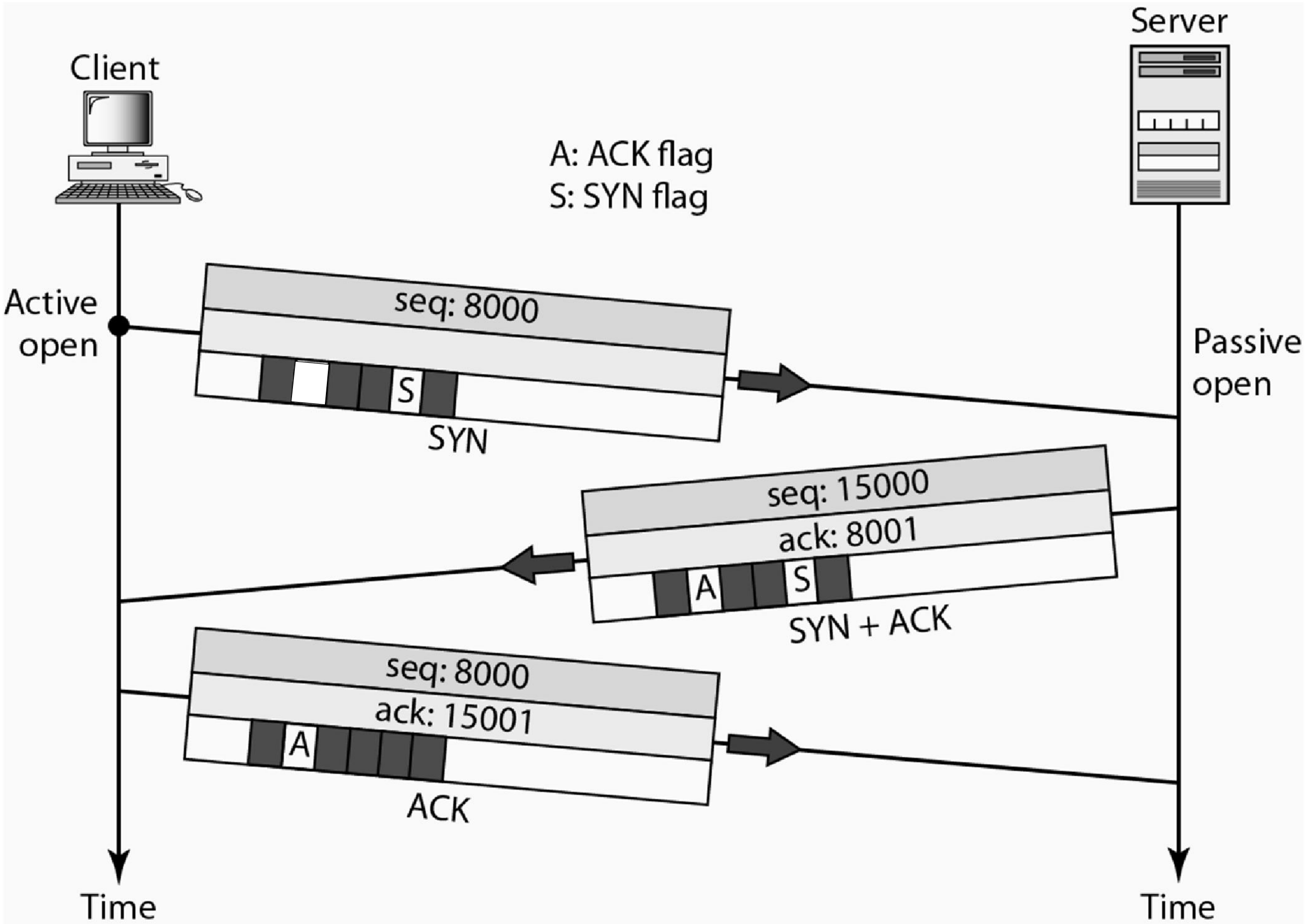
- The client TCP entity must also acknowledge receipt of the server's SYN segment:
  - As there are three segments used in the opening sequence this procedure is called a *three-way handshake*
- **Note on sequence numbers:**
  - Sequence numbers contained in a TCP acknowledgement segment (ACK) is the next expected sequence number
  - SYN messages occupy 1 byte of the sequence number space
  - ACKs on their own do not consume a sequence number
  - Refer to the example connection start-up sequence on the next slide

# Example *Opening* Three Way Handshake

socket()

connect()
(Blocks)

connect()
Returns




socket()
bind()
listen()
accept()
(Blocks)

accept()
Returns

# Closing a TCP connection

- Closing a TCP Connection:
  - An application calling *close* is performing an *active* close
  - The other end of connection is said to be performing a *passive* close
- The call to ***close()*** causes the local TCP entity to sends a FIN segment:
  - This implies that the application is finished sending data
  - Either application, the client or the server, can call ***close()***

# Closing a TCP connection

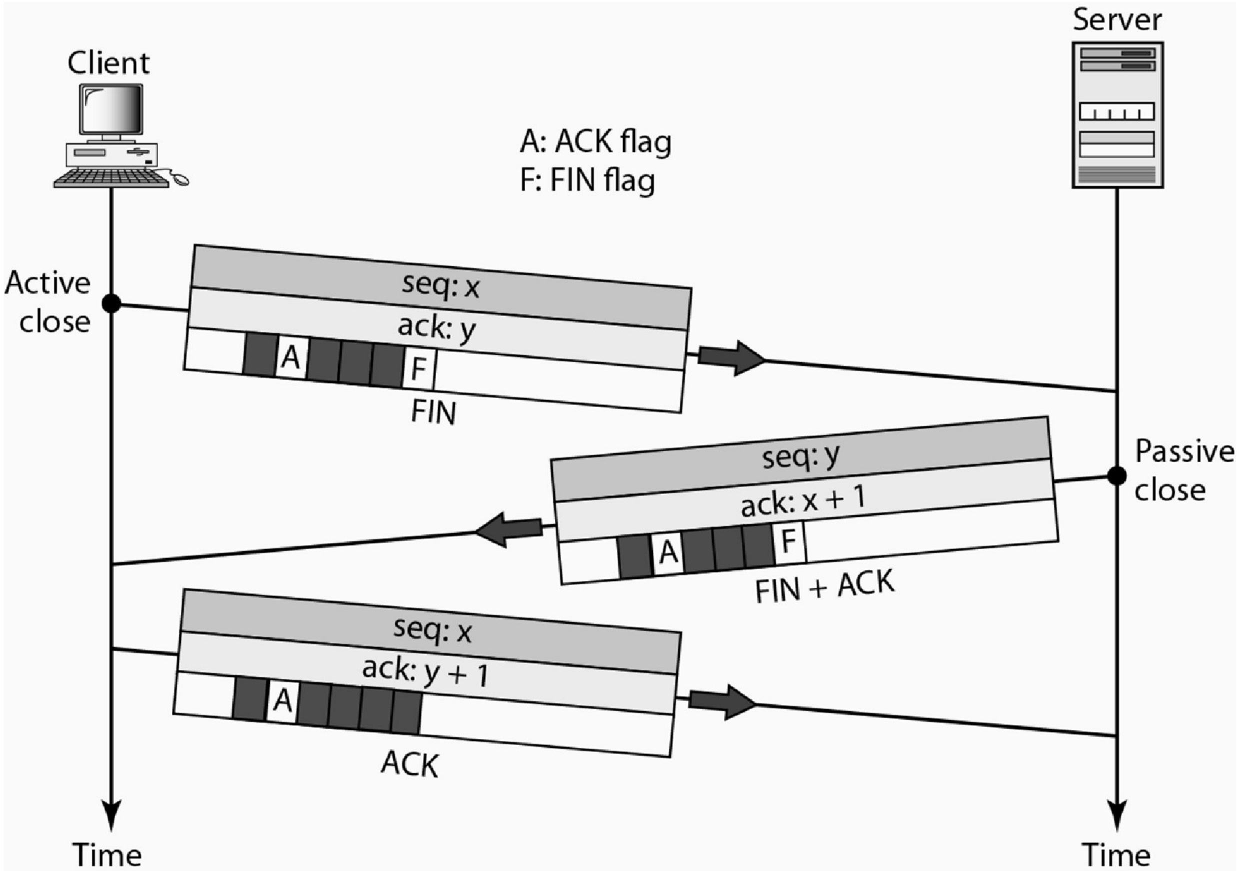
- The FIN segment is acknowledged by the receiving TCP entity:
  - This FIN message is passed to the application as an *end-of-file* and is queued after any remaining data
  - Receipt of a FIN means no more data will arrive on the connection
- An application that receives an *end-of-file* can choose to:
  - Leave the local socket open for longer in order to return data to the remote app. This is known as a *half-close,* **OR**,
  - Close its local socket by calling *close()* resulting in its local TCP entity sending a FIN. This FIN segment must be acknowledged by the remote TCP entity
- This choice results in either a *three-way* or *four-way* handshaking sequence for closing the connection

# Closing a TCP connection

- Closing  a connection requires a FIN and an ACK in each direction i.e. four segments are <u>normally</u> required:
  - However the *active* FIN is normally sent with data and the *passive* FIN can be combined with the ACK into a single segment i.e. a three-way handshake
- Connections can also be closed by terminating the application i.e. terminating the associated Unix process:
  - This causes all open socket descriptors to <u>close</u>
  - This results in a FIN segment being sent on any open TCP connections
- **Note on sequence numbers:**
  - Just like SYN segments, FIN segment also occupy 1 byte of the sequence number space
  - ACKs on their own do not consume a sequence number
  - Refer to the example connection termination sequence on the next slide

# Example *Closing* using a *Three-way* Handshake

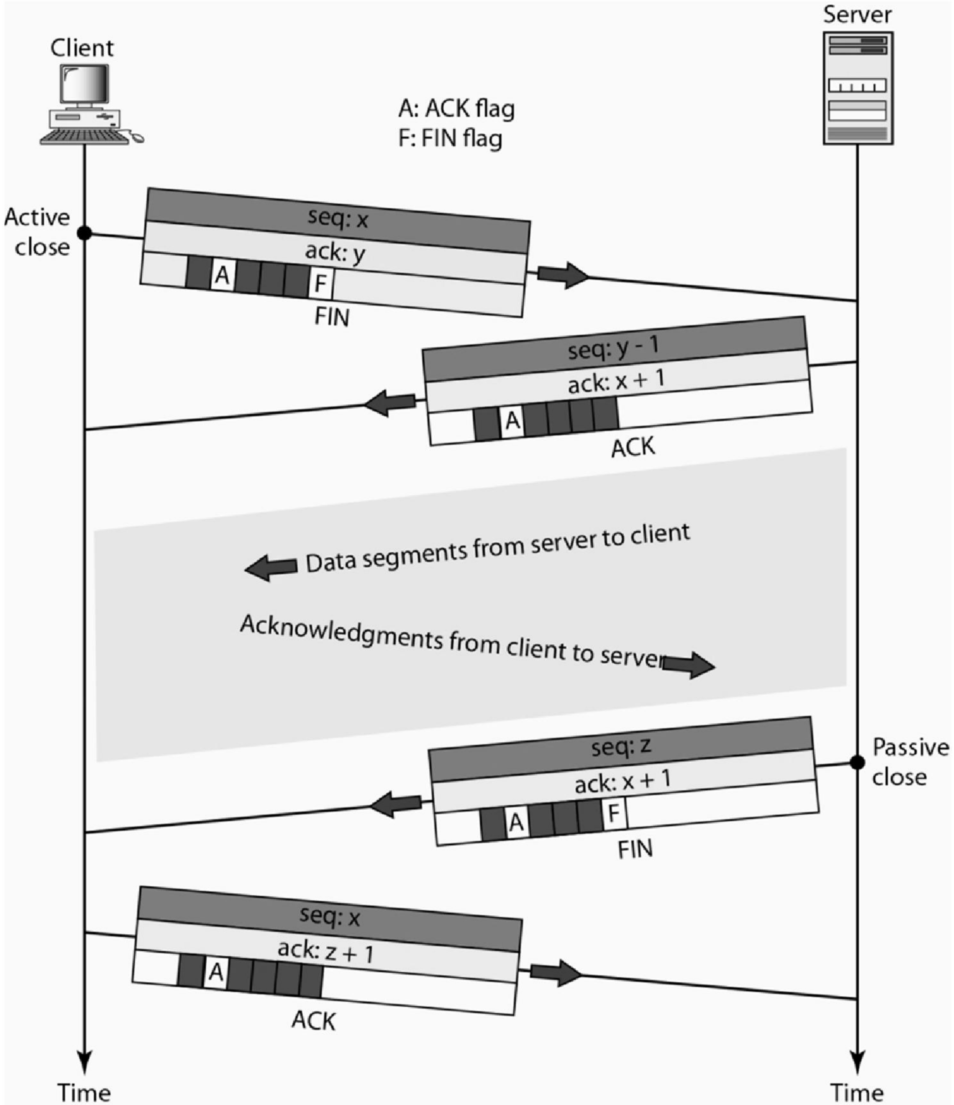
close()




recv()
returns 0

# Example *Closing* using a Four-way Handshake (a Half Close)

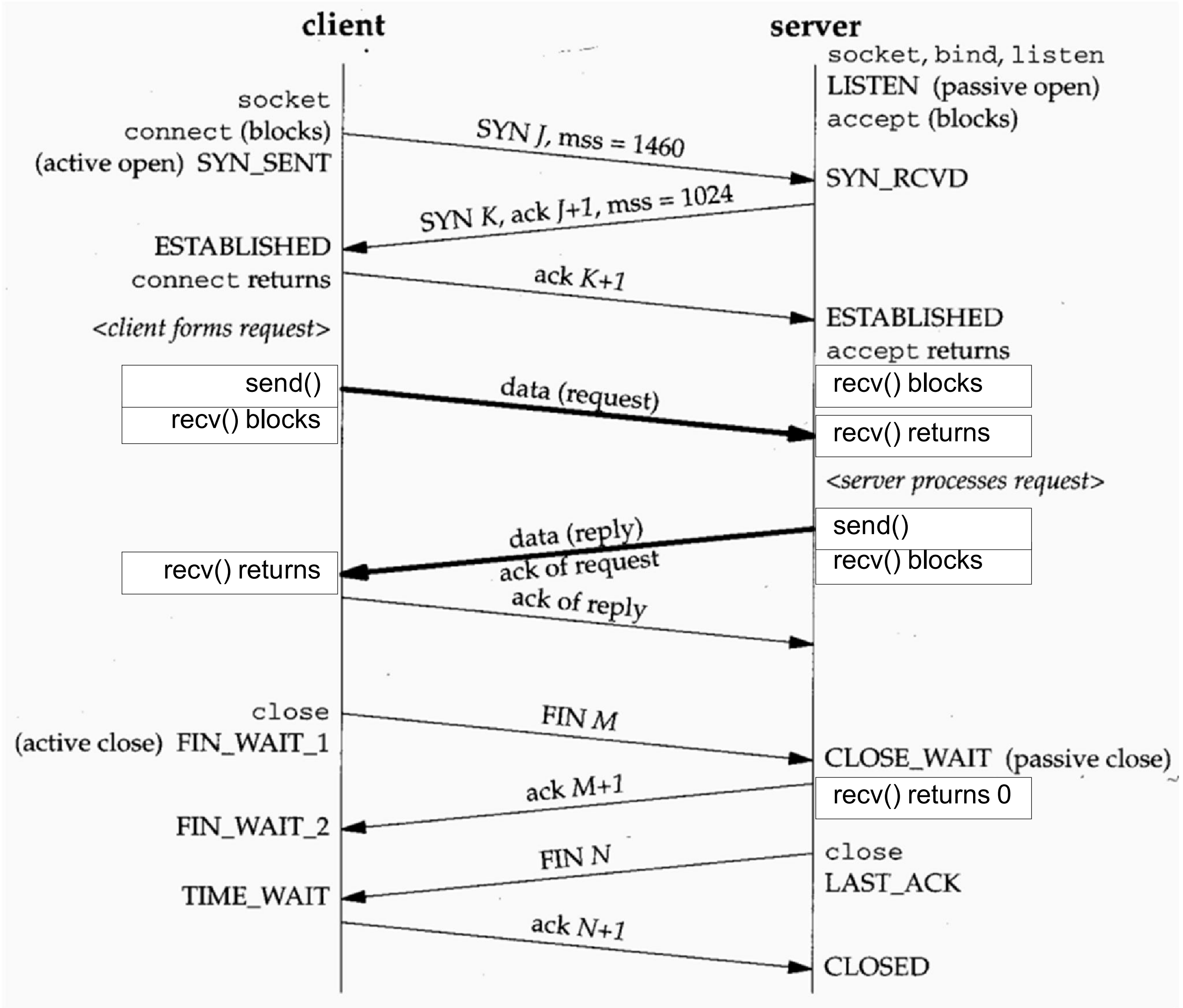
close()




recv() returns 0

close()

# Example of a complete connection sequence



# TCP State Transition Diagram - explained

- The TCP ***State Transition Diagram*** shows the operation of TCP's connection *establishment* and *termination* phases:
  - There are 11 different states defined for a connection
  - Client and server transitions are shown as dark solid and dark dashed lines respectively
  - The transition from one state to another depends on the segment received in that state
    - e.g. an application performing an *active* open in the CLOSED state moves to the SYN_SENT state and then to the ESTABLISHED state upon receipt of a SYN with an ACK
- The ESTABLISHED state is where most data transfer occurs

# TCP State Transition Diagram - explained

- Termination of a connection:
  - From the ESTABLISHED state there are two possible transitions:
    - An application calling *close* i.e. an *active* close moves to the FIN_WAIT_1 state
    - An application receiving a FIN (i.e. a *passive* close) moves to the CLOSE_WAIT state
- In rare circumstances it is possible for both ends to send SYN/FINs simultaneously (known as *simultaneous open/close):*
  - This scenario is not explored here

# The TCP State Transition Diagram

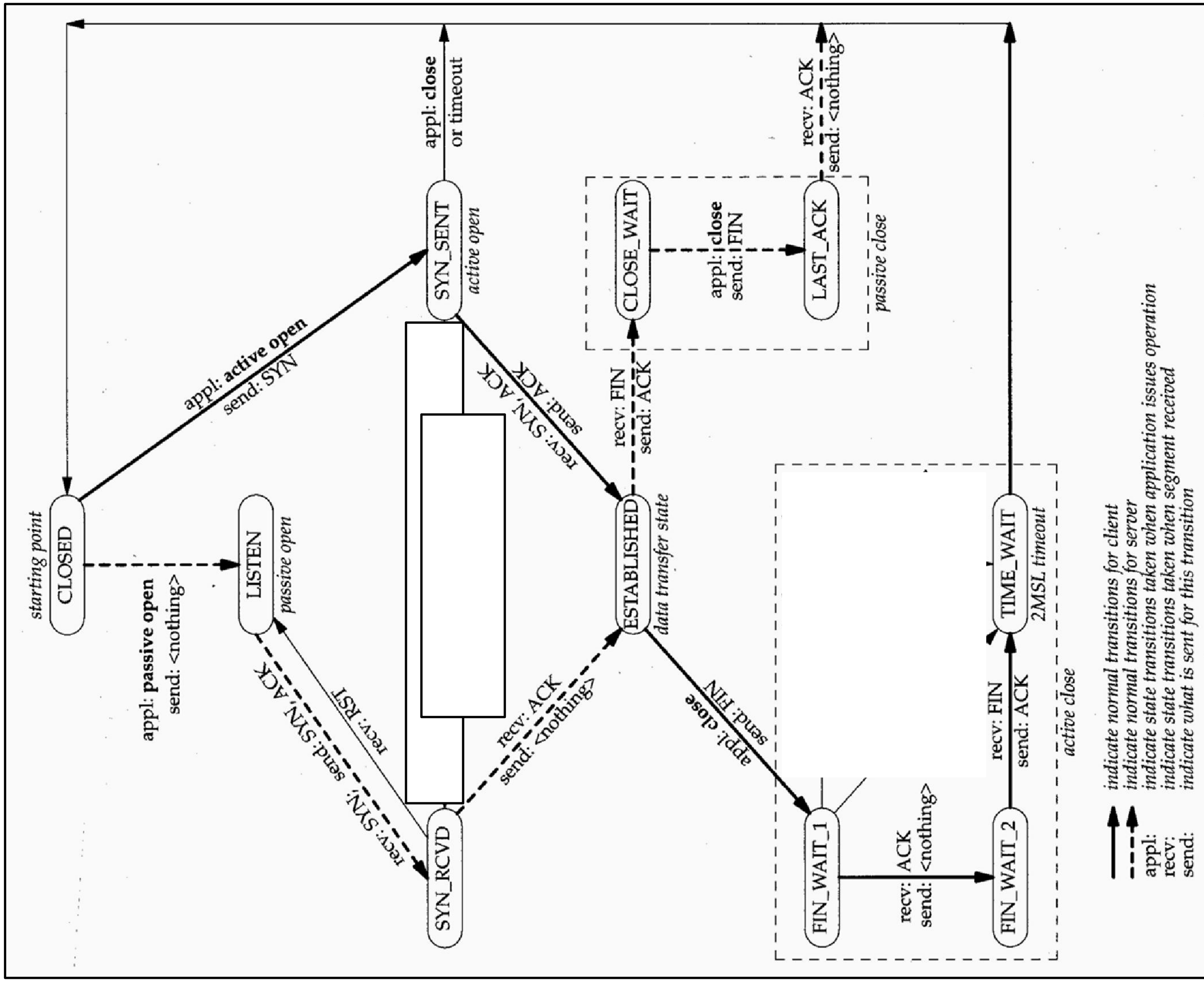



**Figure 2.4** TCP state transition diagram.

# TCP's TIME_WAIT State - explained

- The end that performs the *active* close goes through the TIME_WAIT state for a period of <u>twice</u> the MSL (maximum segment lifetime) a.k.a. *2MSL*
  – MSL is the maximum amount of time that an IP *datagram* can live in an internet. This is linked to the TTL field (max value is 255) – to be covered later
  – TCP chooses a value for MSL of between 1 min. and 4 minutes

# TCP's TIME_WAIT State - explained

- There are <u>two</u> reasons for the TIME_WAIT state:
  - To implement TCP's full-duplex connection *termination* <u>reliably</u>
    - Recall two of TCP's service offerings is *full-duplex communication* and *reliable termination*
    - The end that performs an *active* close remains in the TIME_WAIT state as it might have to retransmit the final ACK
  - To allow old *duplicate* segments to expire in the network:
    - Datagrams containing TCP segments can get caught in *routing loops* within an internet due to *routing errors.* These are known as *lost* or *wandering duplicates*
    - TCP must handle these duplicates for connections that have been *reincarnated*
    - TCP will not initiate an *incarnation* of a connection that is currently in the TIME_WAIT state
    - This guarantees that all old *duplicates* from previous incarnations have expired