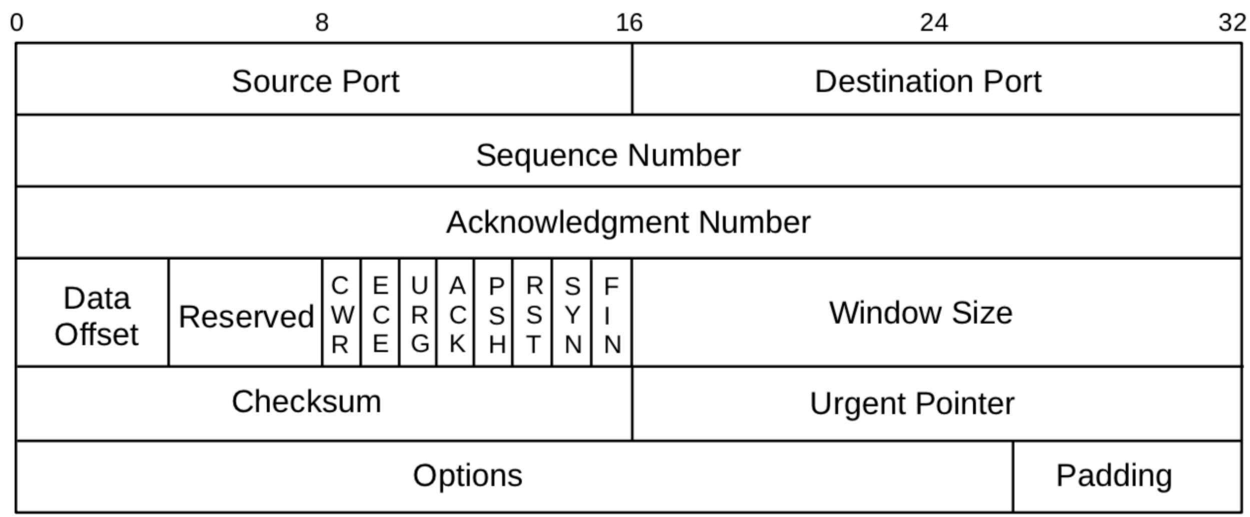
Computer Networking

COMP 177 | Fall 2020 | University of the Pacific | Jeff Shafer

TCP (3)

Transmission Control Protocol

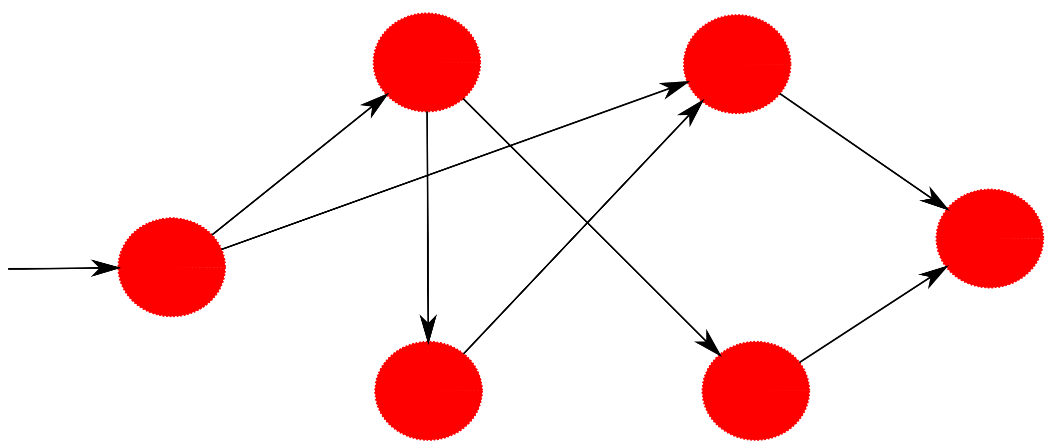
Transmission Control Protocol (TCP)



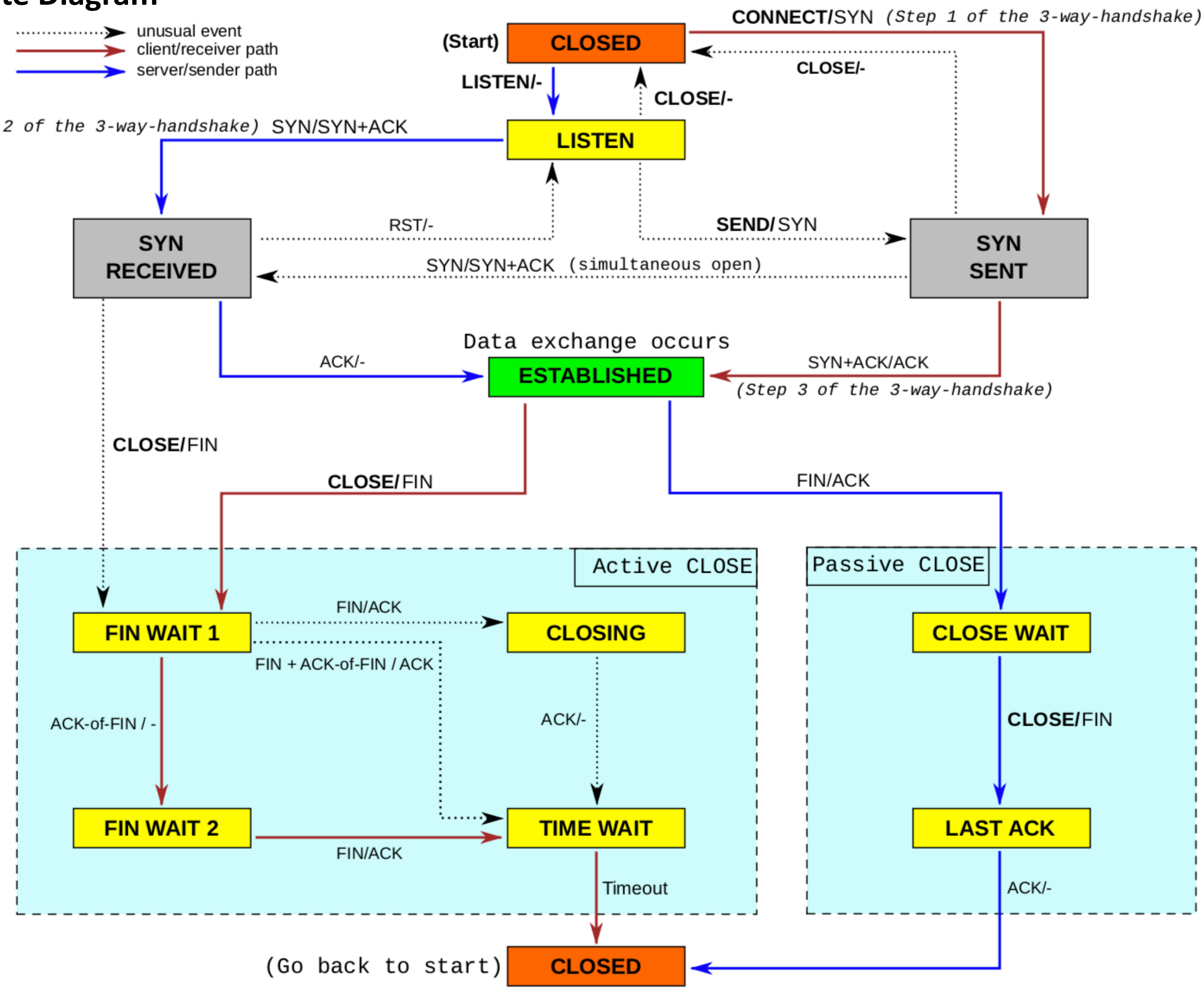
- Connection oriented
- Byte streaming
- Full duplex
- Reliable data transport
- Congestion control
- Flow control

State Diagrams

- Provide a concise and clear way to describe a protocol
- State diagrams describe a state machine consisting of
  - Finite set of states
  - Transition system from one state to another
  - Each state transition comes with a corresponding event and/or action
- State diagrams are appropriate for protocols with a lot of details



TCP State Diagram

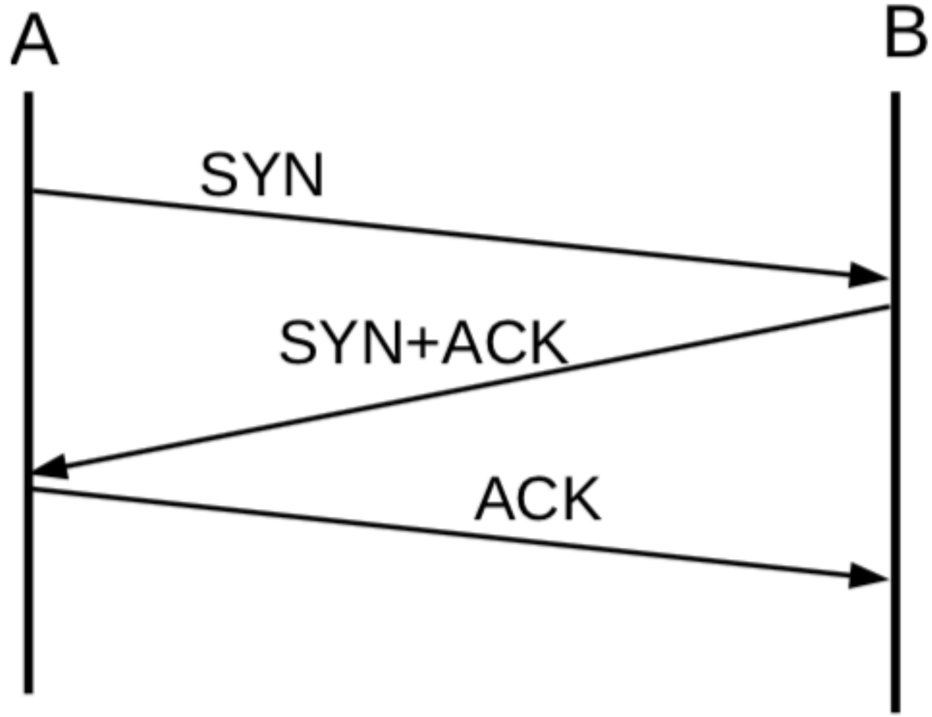


TCP State Diagram

- An **important thing to remember** is that **both the client and server <u>each</u> have their own state diagram and follow it independently!**

- They are each running a (Similar? Identical?) implementation of TCP in the operating system
  - Each implementation is moving through its own state diagram and making decisions about what to do next

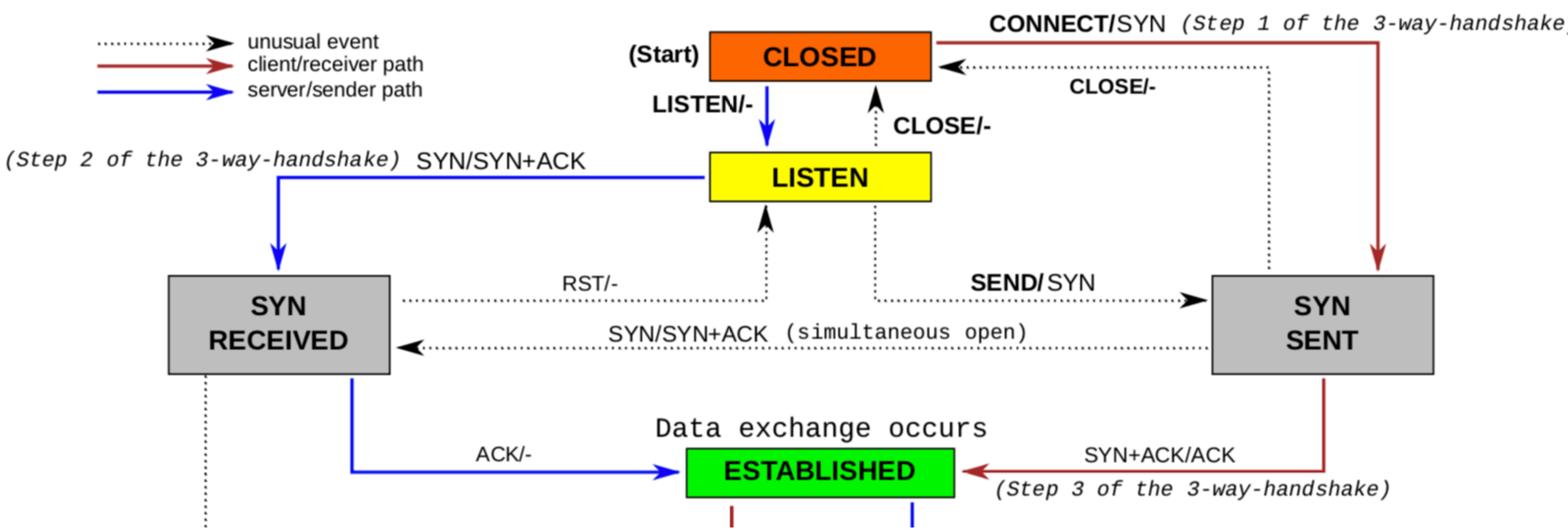
TCP Three-Way Handshake



TCP three-way handshake

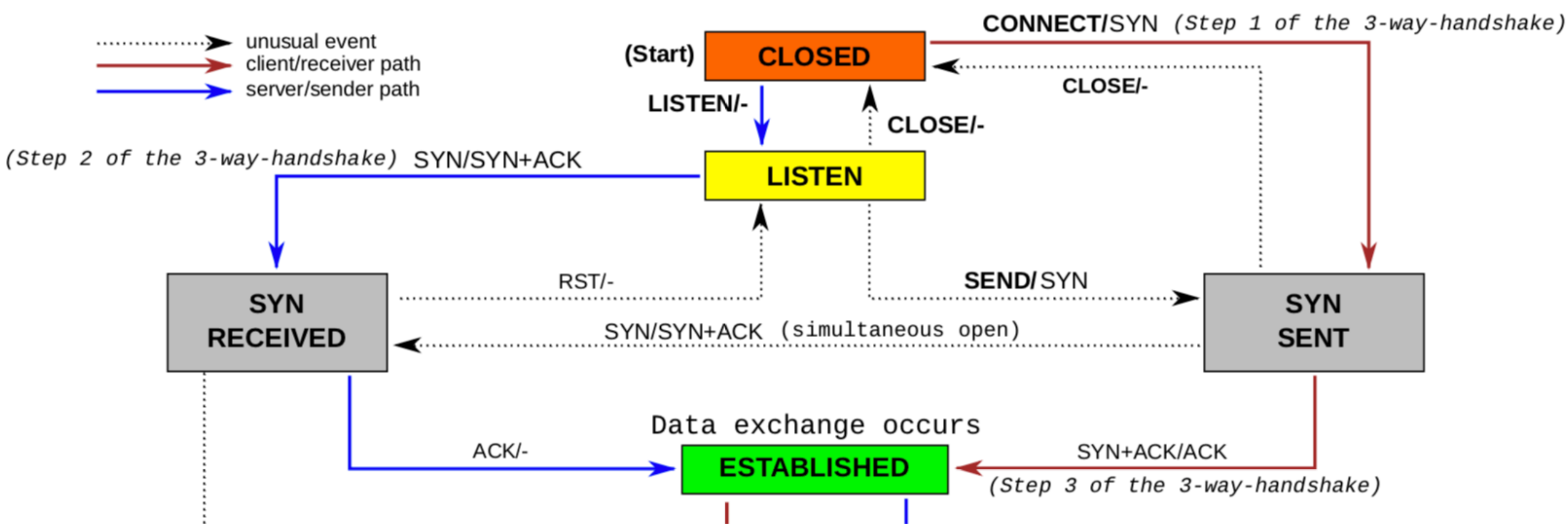
TCP State Diagram

- Initially, both the server and the client are in the **CLOSED** state
  - Both in the client and server, the socket is created (`socket()`) and address and port are bound (`bind()` on server)



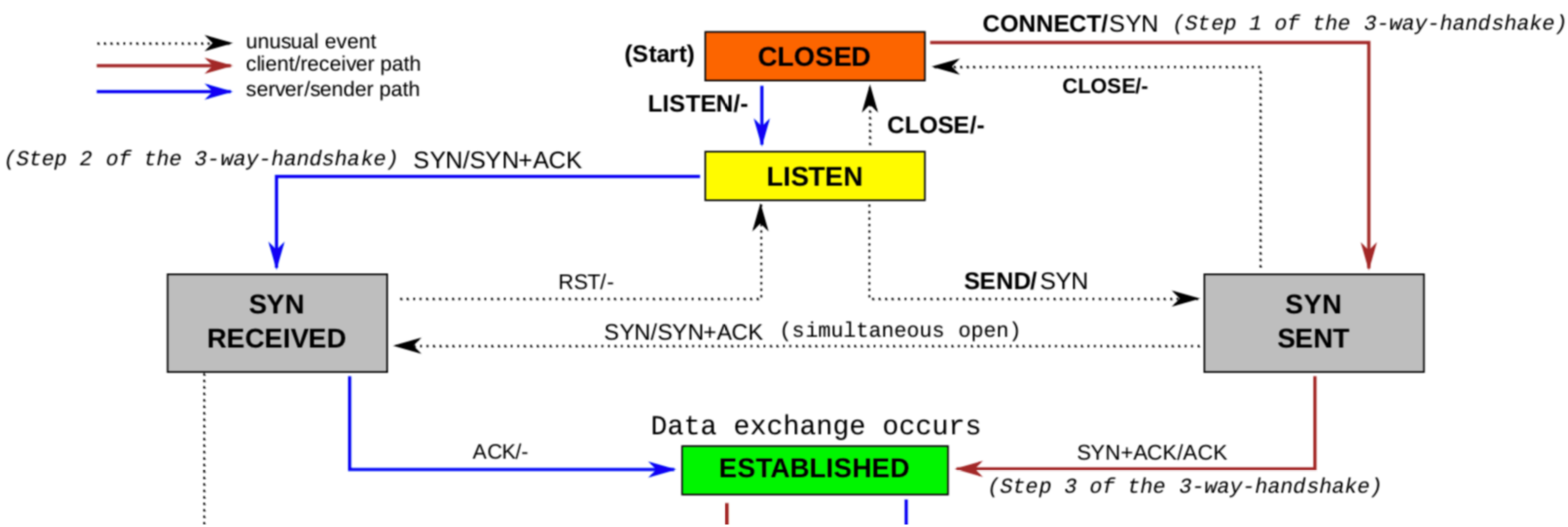
TCP State Diagram

- SERVER: After calling `listen()`, server goes to **LISTEN** state
- SERVER: After calling `accept()`, server waits to receive SYN packet from incoming client connection



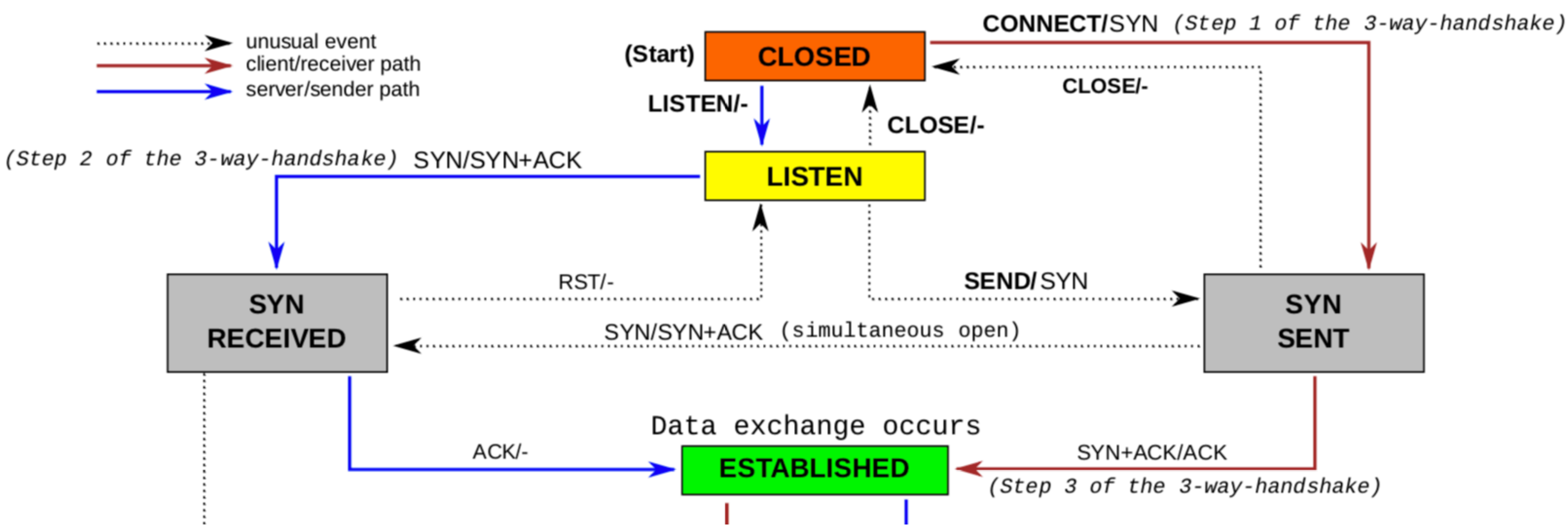
TCP State Diagram

- CLIENT: After calling `connect()` the client sends SYN packet and goes to **SYN SENT** state
  - In this state, the client is waiting for SYN-ACK packet from the server
- SERVER: While in **LISTEN** state, if the server receives the SYN packet, it responds with SYN-ACK and goes to **SYN RECEIVED** state
  - In this state, the server is waiting for ACK packet from the client



TCP State Diagram

- CLIENT: While in **SYN SENT** state, if the client receives the SYN-ACK packet, the client responds with ACK and goes to **ESTABLISHED** state. 3-way handshake is completed!
- SERVER: While in **SYN RECEIVED** state, if the server receives the ACK packet (of SYN-ACK) then it goes to **ESTABLISHED** state. 3-way handshake is completed!
- In **ESTABLISHED** state, application layer messages can be communicated between the client and server ☺

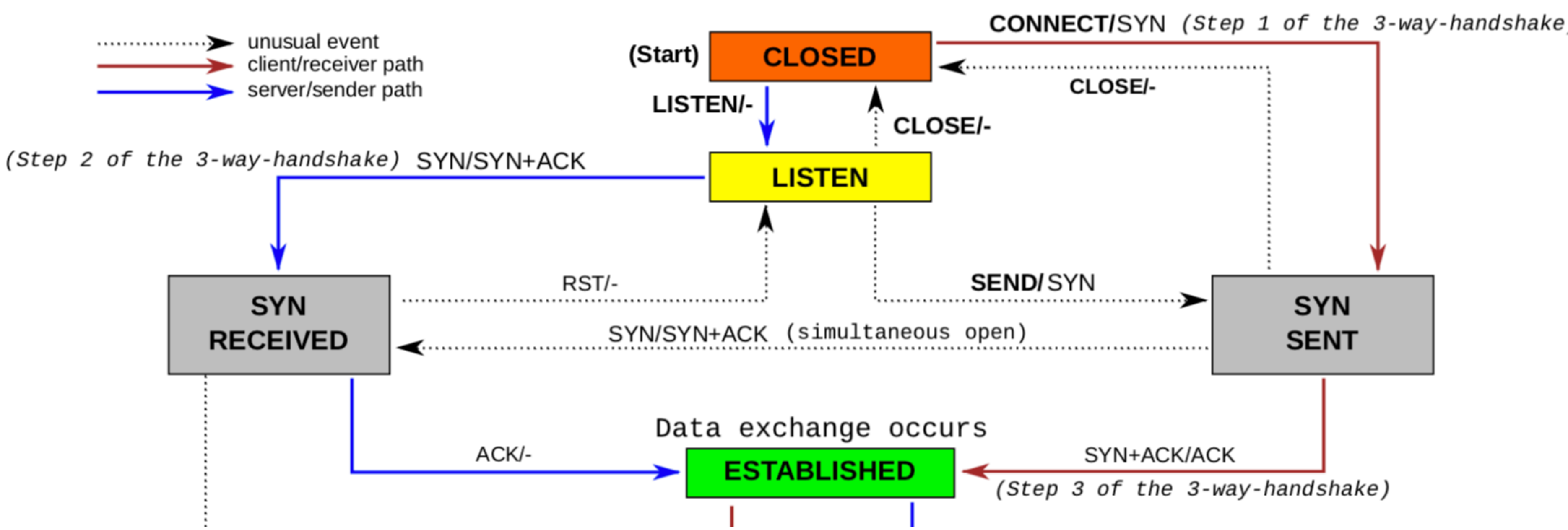


TCP State Diagram

- That was the *usual* state transitions for client and server
- **Applications can use sockets in *unusual* ways!**

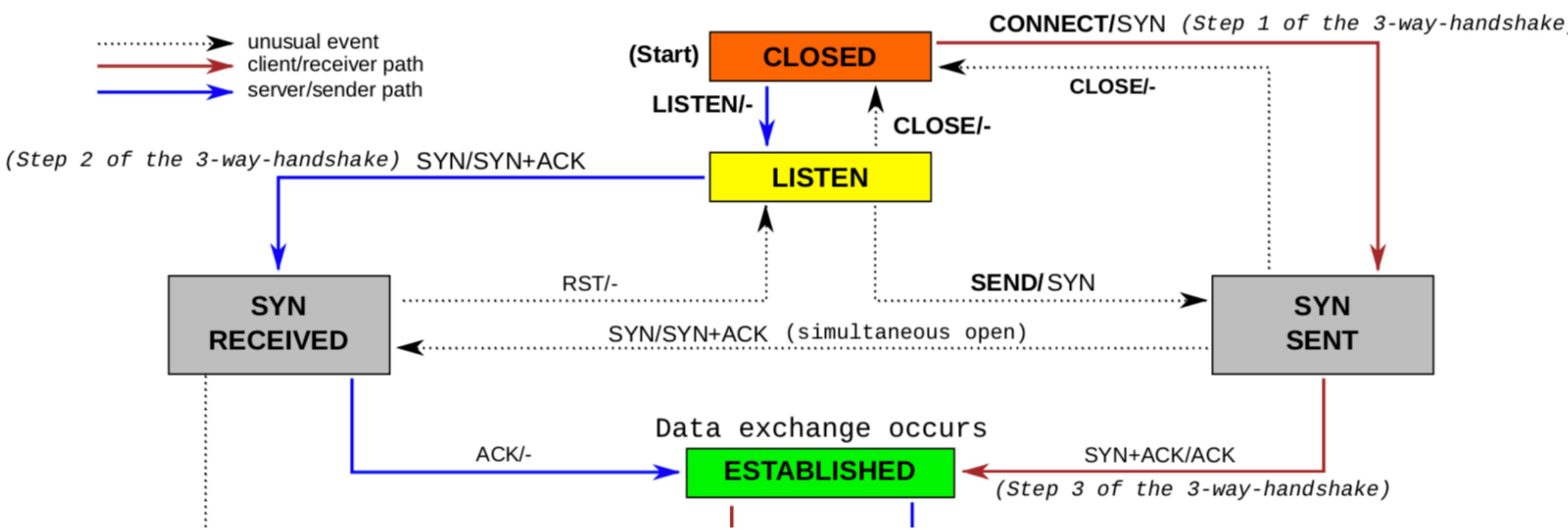
TCP State Diagram – *Unusual…*

- SERVER: The server may `close()` the connection while **LISTEN**ing.
  - The server goes back to **CLOSED** state
- CLIENT: While in **SYN SENT** and waiting for SYN-ACK, the client may `close()` the connection
  - The client goes back to CLOSED state



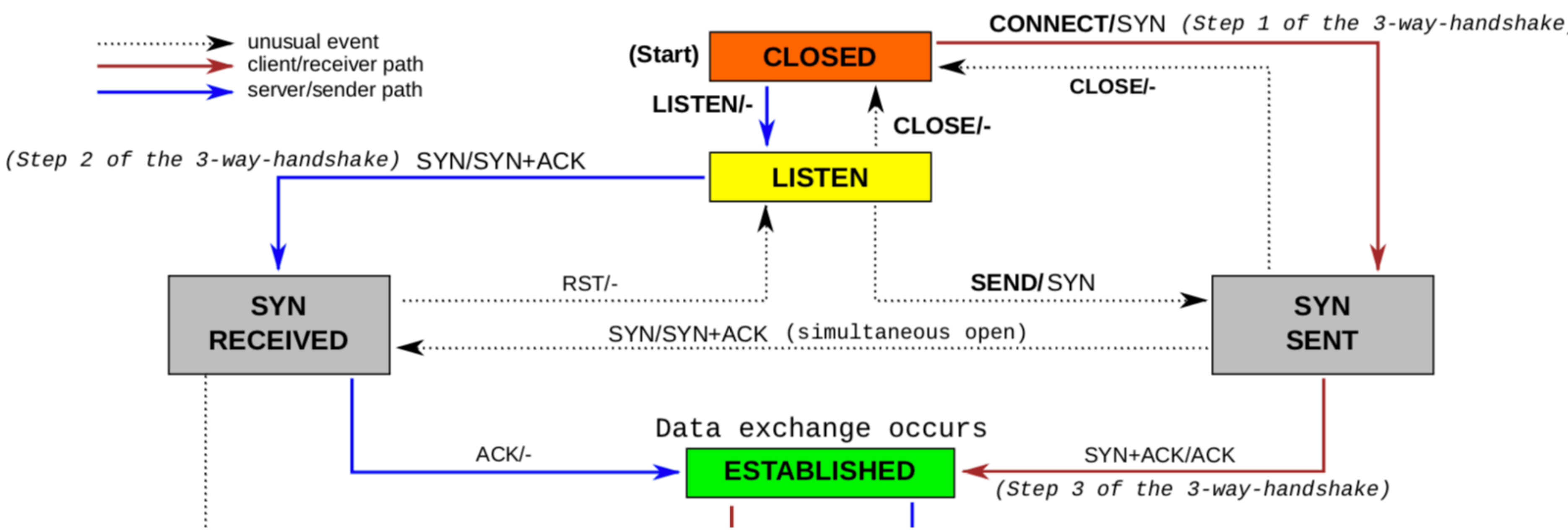
TCP State Diagram – *Unusual…*

- SERVER: While in **SYN RECEIVED** state and waiting for ACK, the server may receive RST packet from client
  - The server goes back to LISTEN state

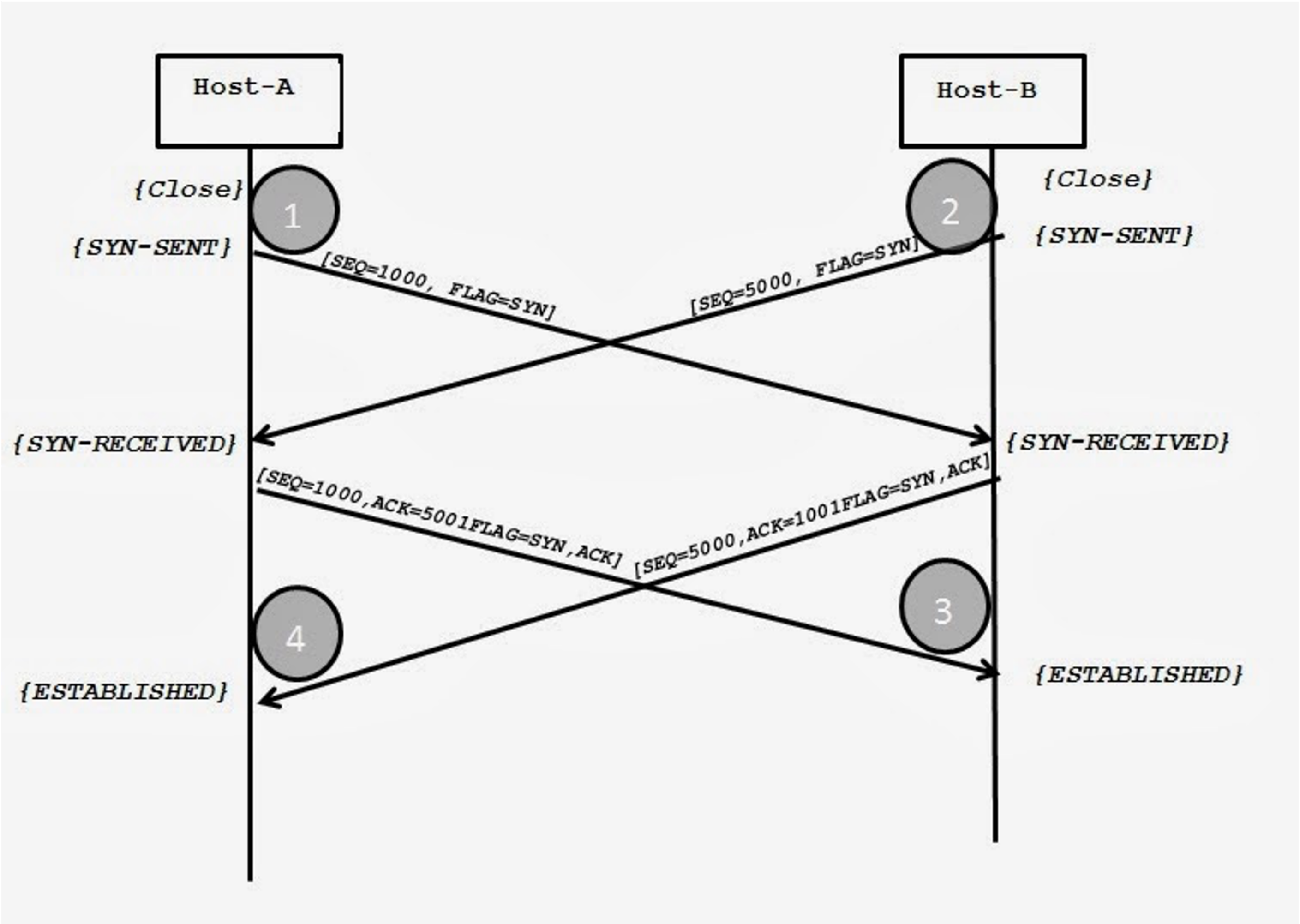


TCP State Diagram – *Unusual…*

- Simultaneous Open: It's possible for two applications to send a SYN to *each other* to start a connection
  - The possibility is small, because both sides must know which port on the other side to send to
  - While in **SYN SENT**, the instance receives SYN packet from the other side. Then, it sends a SYN-ACK and goes to **SYN RECEIVED** state.

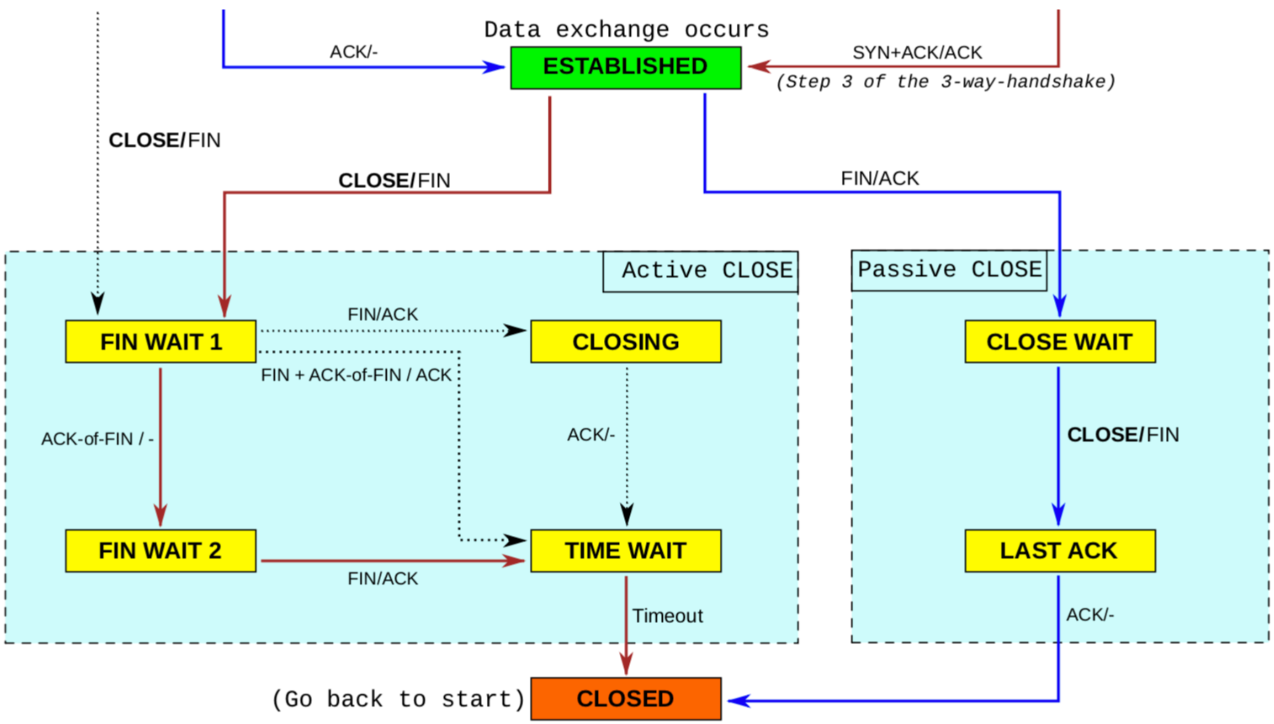


TCP Simultaneous Open



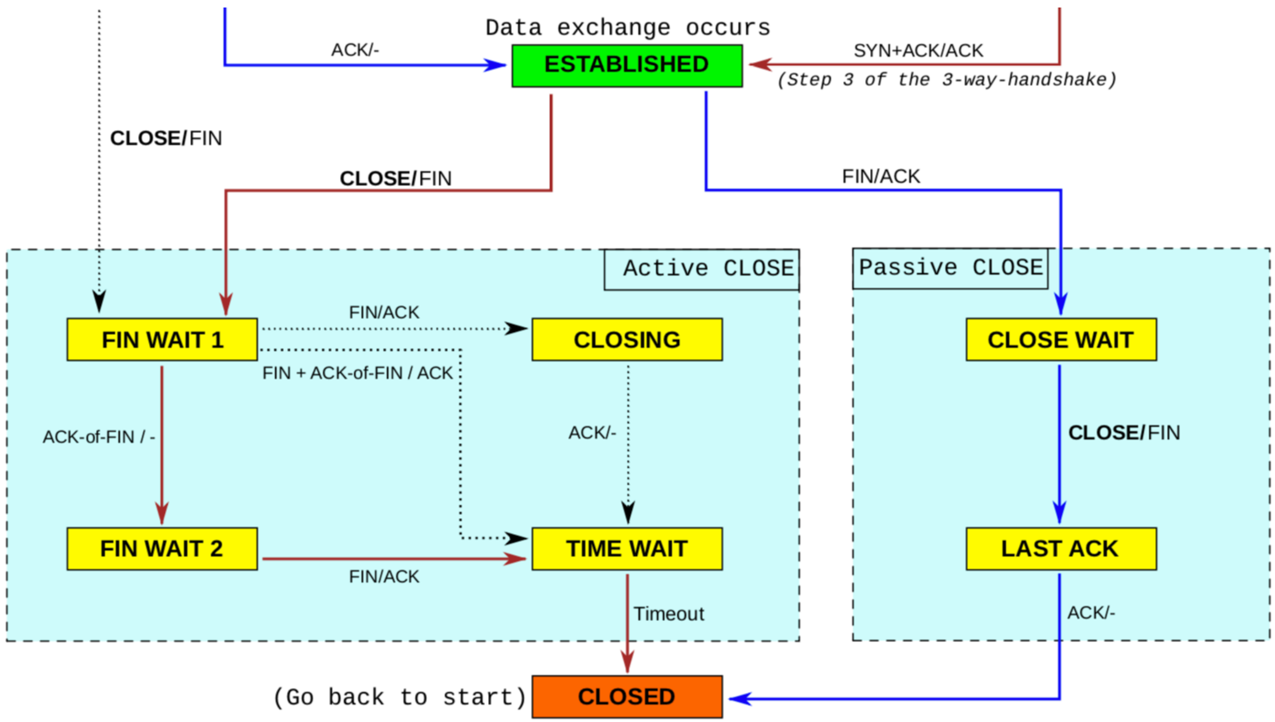
TCP State Diagram

- Both sides need to close the TCP connection
  - The "**active**" instance sends the first FIN packet
  - The "**passive**" instance sends the second FIN packet



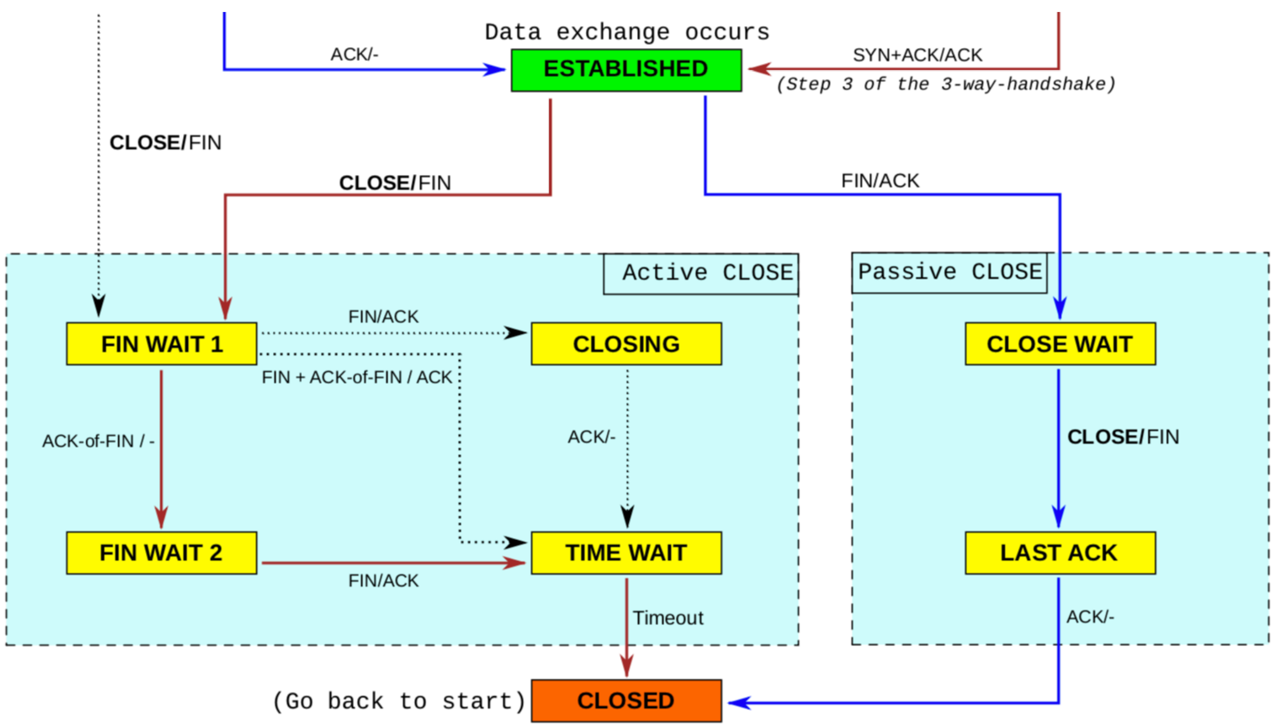
TCP State Diagram

- Passive close: While in **ESTABLISHED** state, the instance receives FIN packet, acknowledges it, and goes to **CLOSE WAIT**
  - In **CLOSE WAIT**, the instance can *still send data*



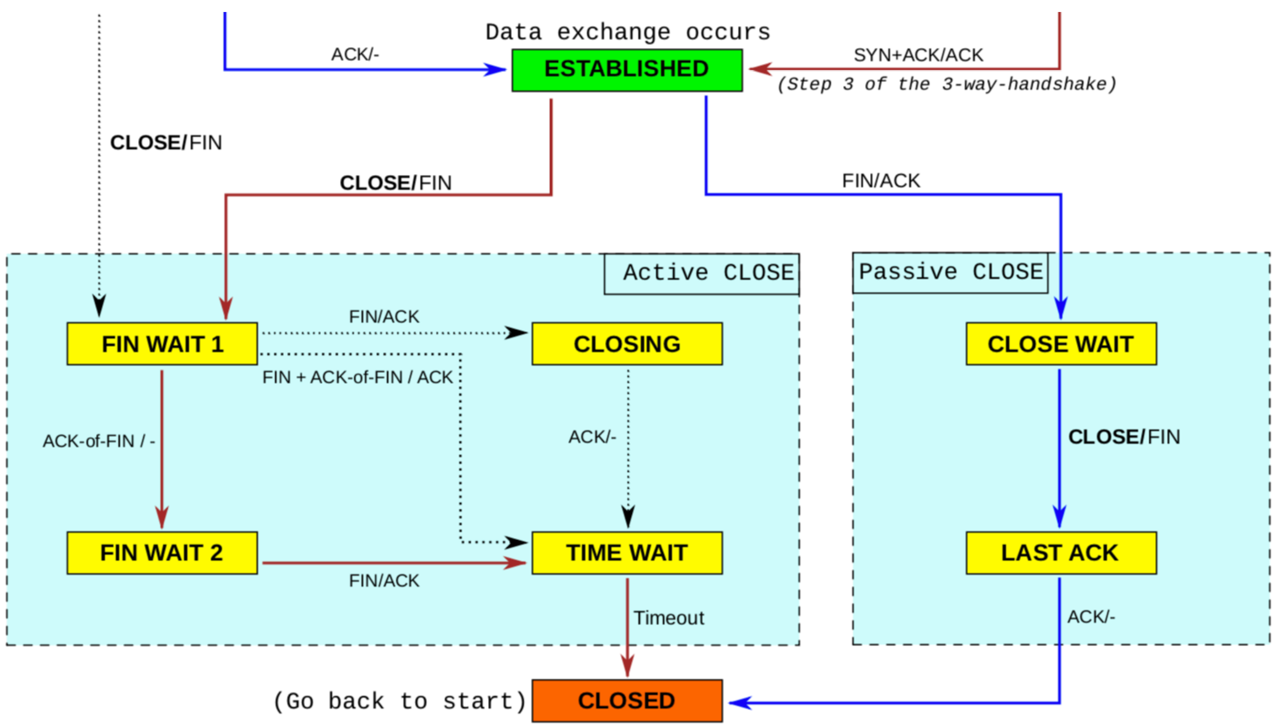
TCP State Diagram

- Passive close: While in **CLOSE WAIT**, if the instance calls `close()`, it sends a FIN packet and goes to **LAST ACK**.
  - In **LAST ACK**, the instance *cannot send data* anymore



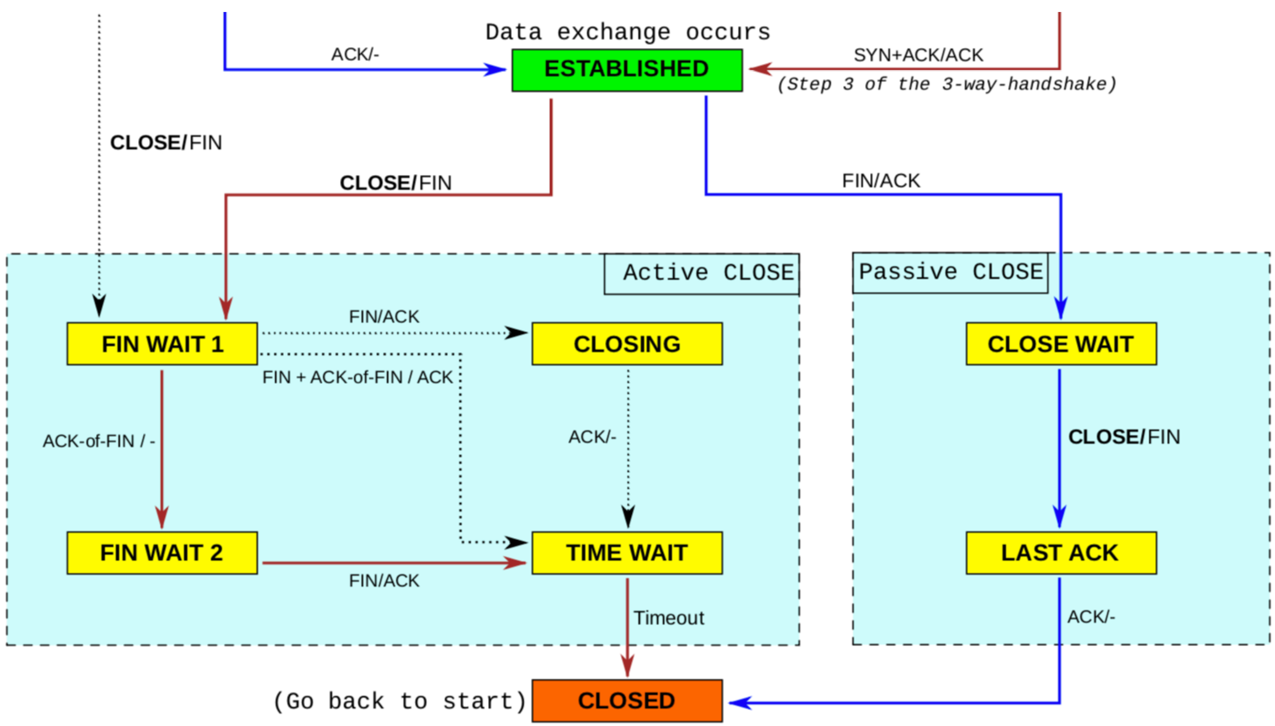
TCP State Diagram

- Active close: While in **ESTABLISHED**, if the instance calls `close()`, it sends a FIN packet and goes to **FIN WAIT 1**
  - In **FIN WAIT 1**, the instance is waiting to received ACK for sent FIN



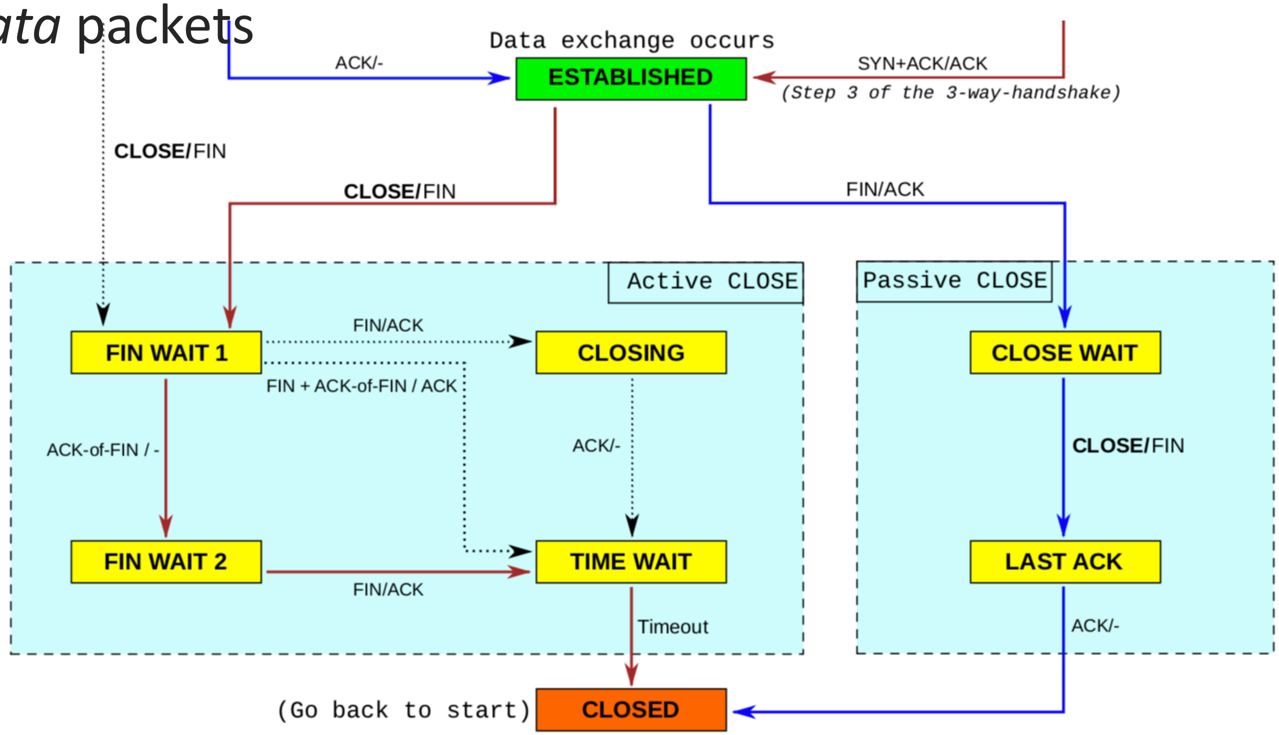
TCP State Diagram

- Active close: While in **FIN WAIT 1**, if the instance receives the ACK of already sent FIN, it goes to **FIN WAIT 2**
  - In **FIN WAIT 2** state, instance can *still receive data*



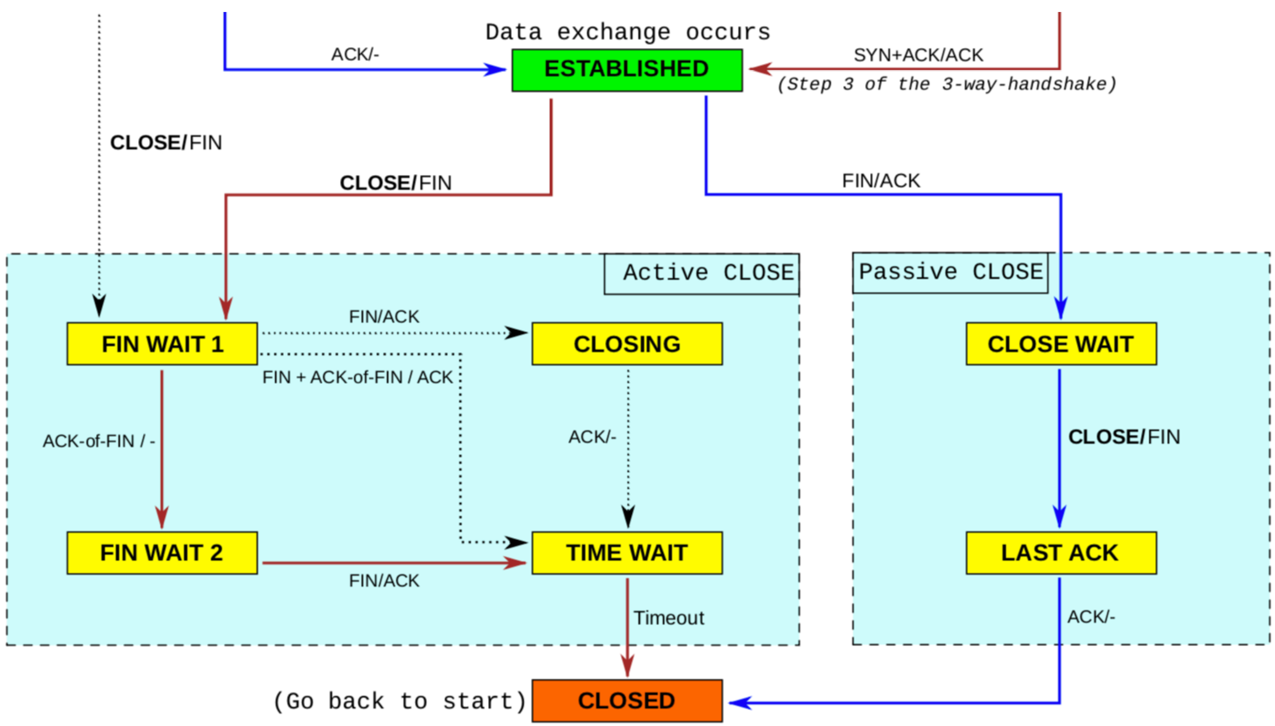
TCP State Diagram

- Active close: While in **FIN WAIT 2**, if the instance receives FIN packet, it sends the ACK, starts a timer, and goes to **TIME WAIT**
  - In **TIME WAIT** state, instance can still receive *potentially delayed data* packets



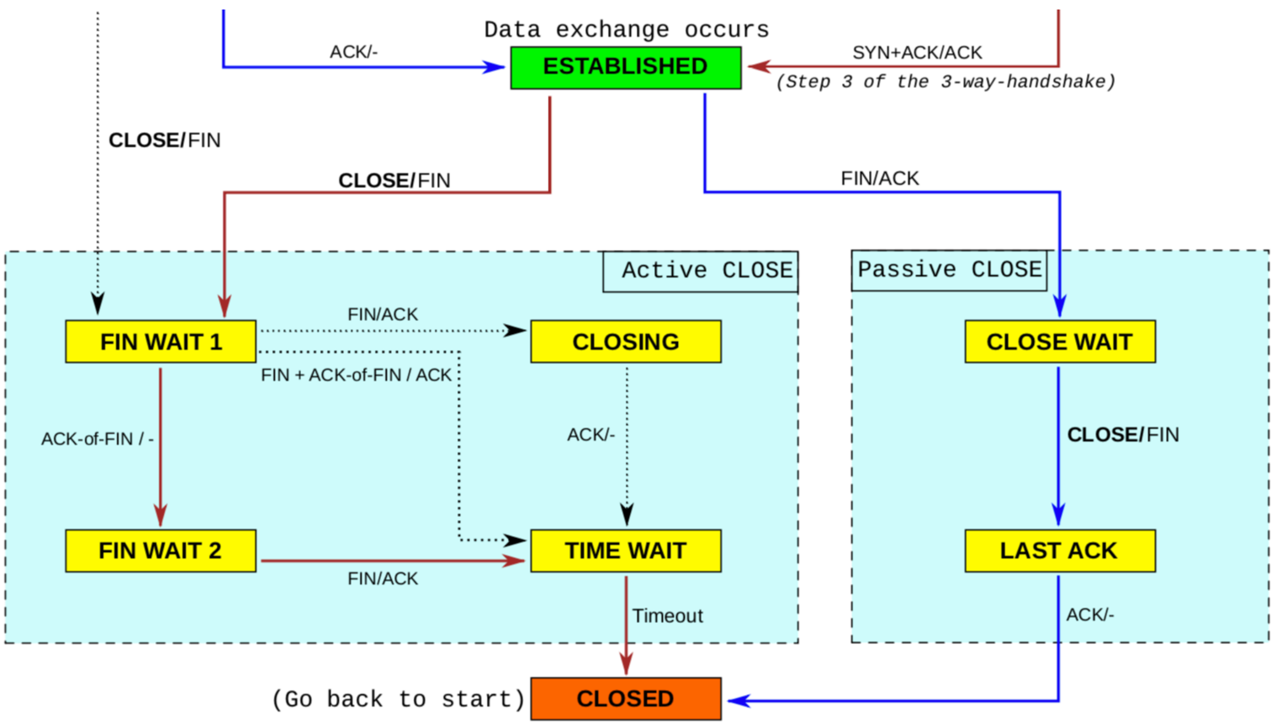
TCP State Diagram

- Active close: While in **TIME WAIT**, when the timer expires, it deallocates the socket resources and goes to **CLOSED**
  - The timeout value is usually 1-2 minutes



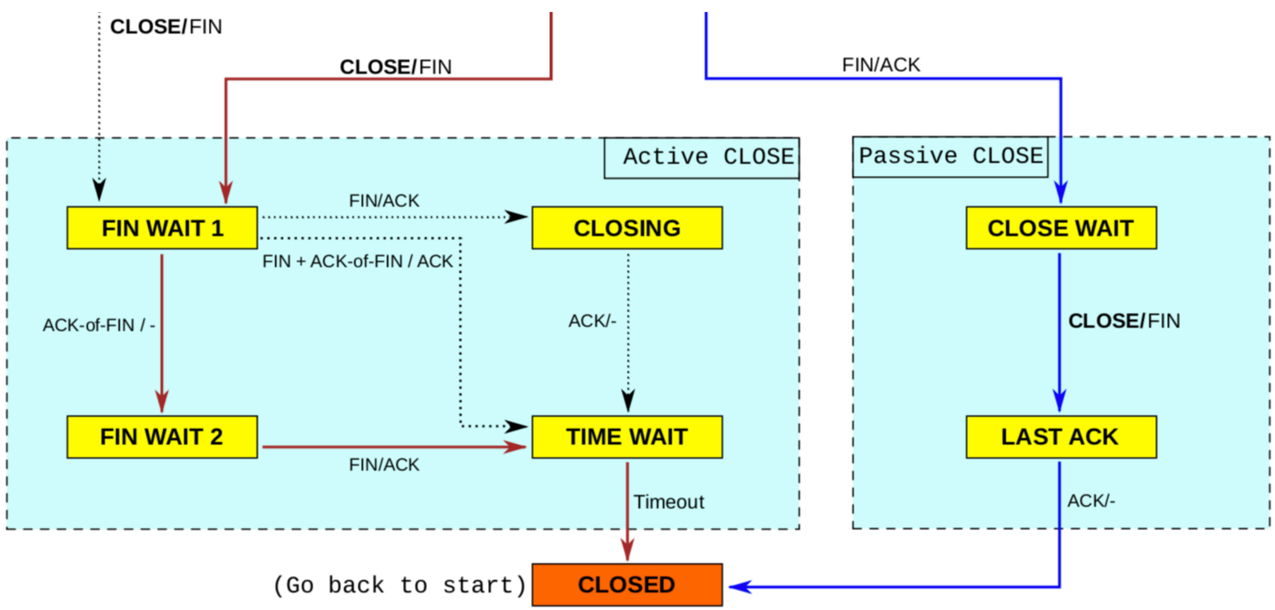
TCP State Diagram

- Active close: While in **FIN WAIT 1** and expecting ACK, if it receives FIN-ACK packet, it sends the ACK of FIN and directly goes to **TIME WAIT**

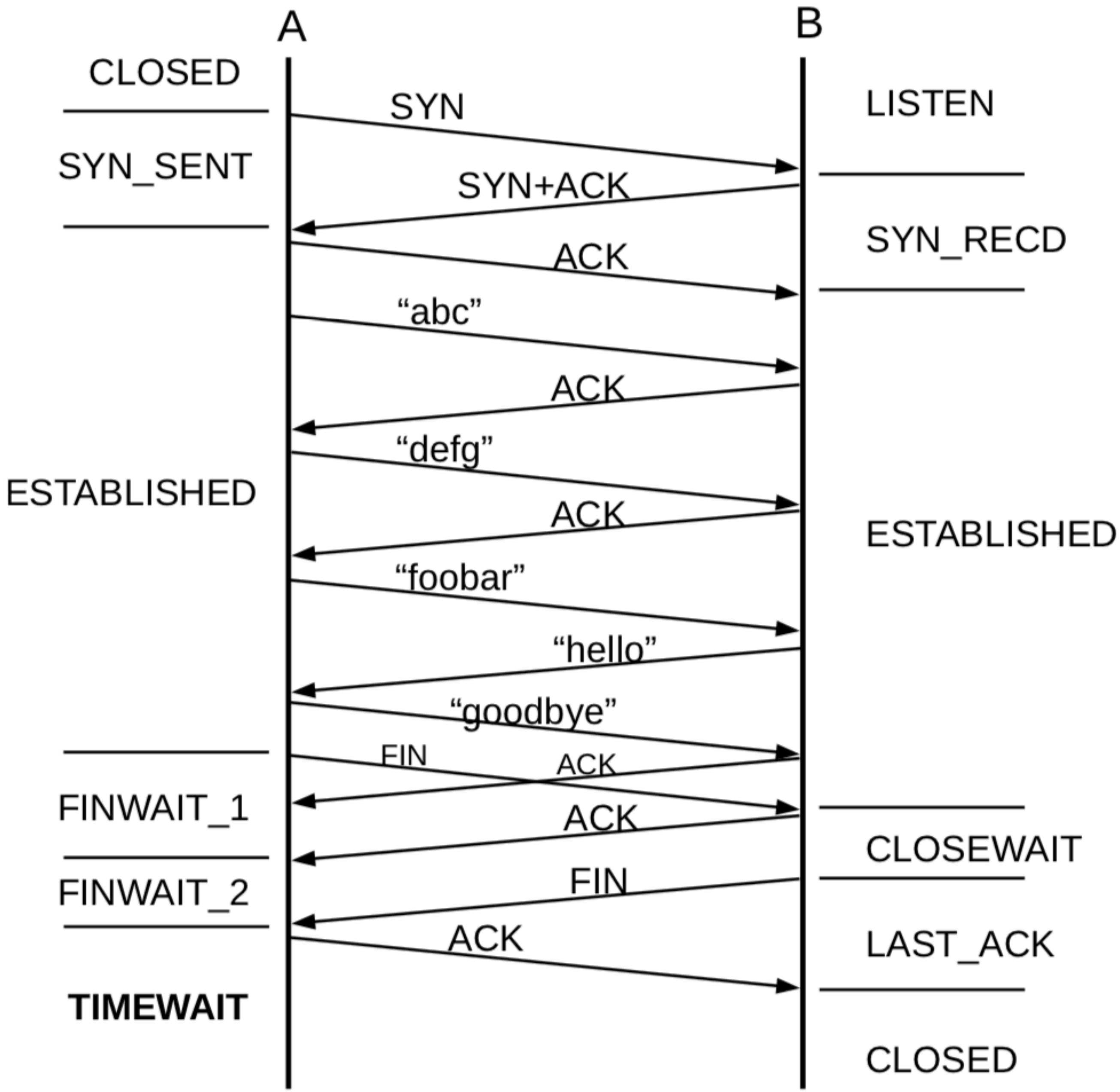


TCP State Diagram

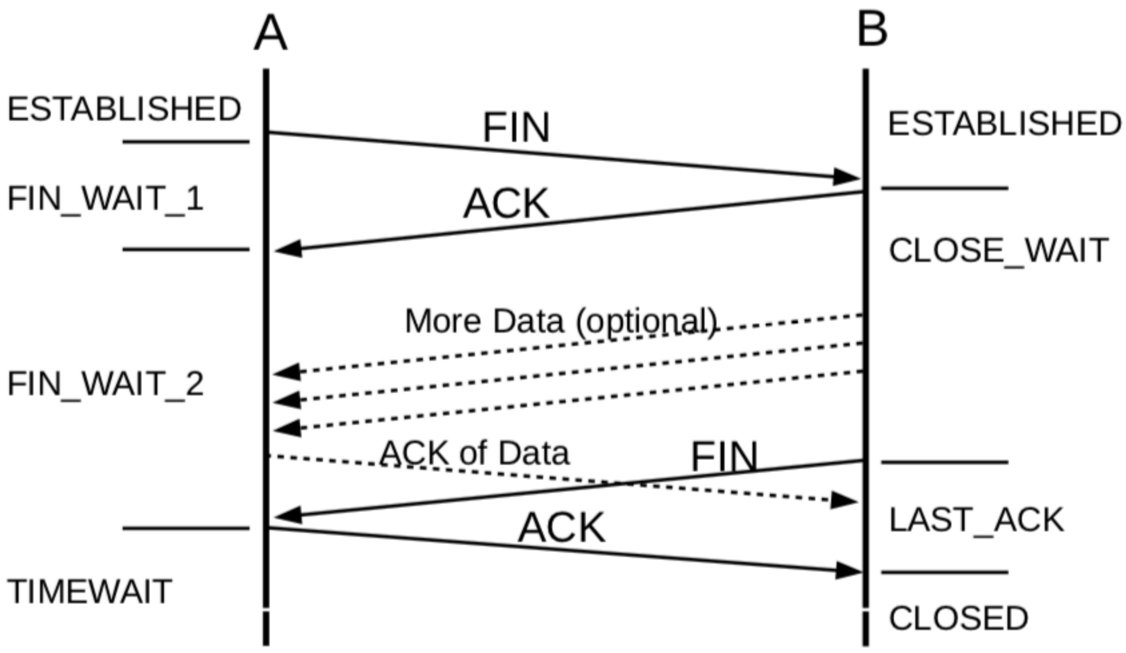
- Active close: While in **FIN WAIT 1** and expecting ACK, if it receives FIN, it sends the ACK of FIN and goes to **CLOSING**
  - In CLOSING state, it waits to receive ACK of sent FIN
  - If so, it goes to **TIME WAIT**



State Changes



State Changes in TCP Closing



Normal close

Checking TCP States

- You can use `netstat -a` to check the status of all TCP connections, in your machine.
- Most TCP states are ephemeral. The exceptions are
  - ESTABLISHED: Both sides sending application messages
  - LISTEN: The server is listening on its welcoming sockets
  - TIME WAIT: Waiting a few minutes before closing the connection fully
  - CLOSE WAIT: The connection is half-open, after closing one way
  - FIN WAIT 2: The connection is half-open, after closing one way

Path MTU Discovery

- TCP is a byte stream protocol that needs to divide the application layer message into *smaller segments*
- The maximum application layer data as a payload of TCP is called maximum segment size (MSS)
- MSS is determined by the *maximum transmission unit* (MTU) in the path between the two ends
- TCP service may need to *discover* the path MTU in order to maximize MSS
  - Different approach is used by TCP over IPv4 versus IPv6

Path MTU Discovery in IPv4

- To discover path MTU with TCP on IPv4
  - An IPv4 packet with DF=1 is sent with a certain size *x*
  - If ICMP message "Fragmentation required, but DF set" is received, or the packet times out, then a packet with smaller size is sent with DF=1.
  - If the acknowledgement is received for the sent packet, then a packet with size bigger than *x* is sent, where DF=1
- Process repeats to experimentally find the MTU
  - Typical sizes of 512-1500 bytes is covered by this process by a few discrete values

Path MTU Discovery in IPv6

- IPv6 does not have DF flag
- When TCP uses IPv6, in order to discover path MTU:
  - TCP sends IPv6 packets with gradually increasing size
  - This process continues until ICMPv6 “Packet too Big” is received.
  - ICMPv6 “Packet too Big” message can be sent by any intermediary node
  - Note that IPv6 routers do not fragment the packets. They drop larger than MTU packets and send back ICMPv6 message, reporting the case
  - If ICMPv6 error received, an IPv6 packet with smaller size is tried. If successful, MTU is discovered

Reliable Data Transport

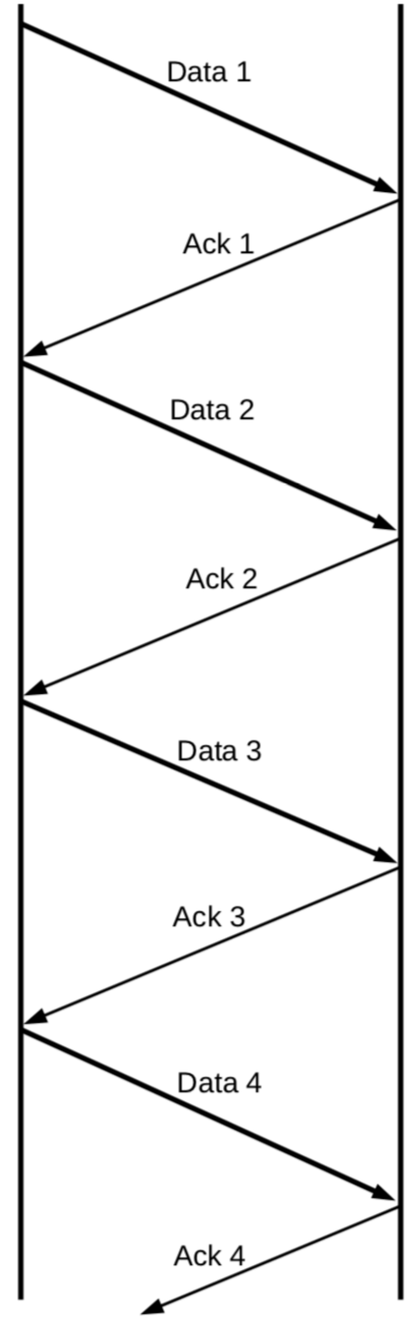
- **Problem**: How to build a reliable data transport service on top of an unreliable service?
  - *An abstract discussion, independent of (reliable) TCP and (unreliable) IP.*

- **Short answer**: achieved by *retransmission-on-timeout* policy
  - If a packet is sent, and no acknowledgment is received within the timeout interval, then the packet is resent.
  - Protocols that implement this policy are called ARQ (Automatic Repeat reQuest).
  - To improve throughput in ARQs, *sliding windows* are used.
  - Retransmission-on-timeouts require *sequence numbers* for the packets, in order to identify them.
  - Notation: Let's denote
    - Data[N]: Nth data packet
    - Ack[N]: Acknowledgement of Nth data packet *cumulatively*, i.e., acknowledging every packet up to Nth

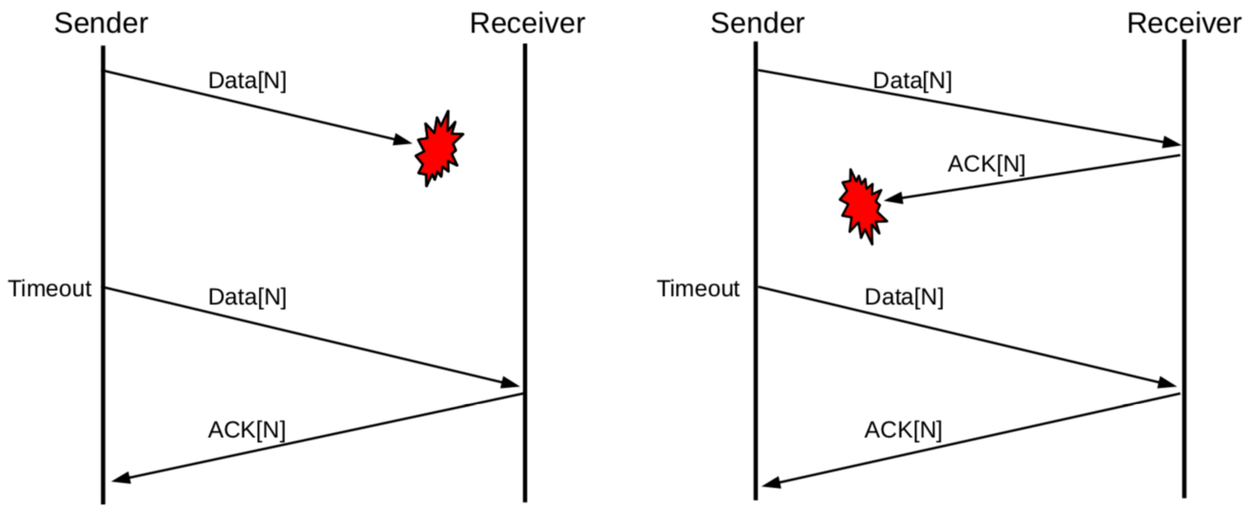
Stop-and-Wait

- The simplest ARQ protocol is a *stop-and-wait* protocol.
  - The sender only sends one outstanding packet in a time
  - The sender starts a timer upon sending the packet
  - If that outstanding packet is acknowledged within the timeout interval, then the sender sends the next packet in sequence, and resets the timer
  - Otherwise, if the sender does not receive the acknowledgement before timing out, it retransmits the outstanding packet, and resets the timer

Stop-and-Wait



Stop-and-Wait



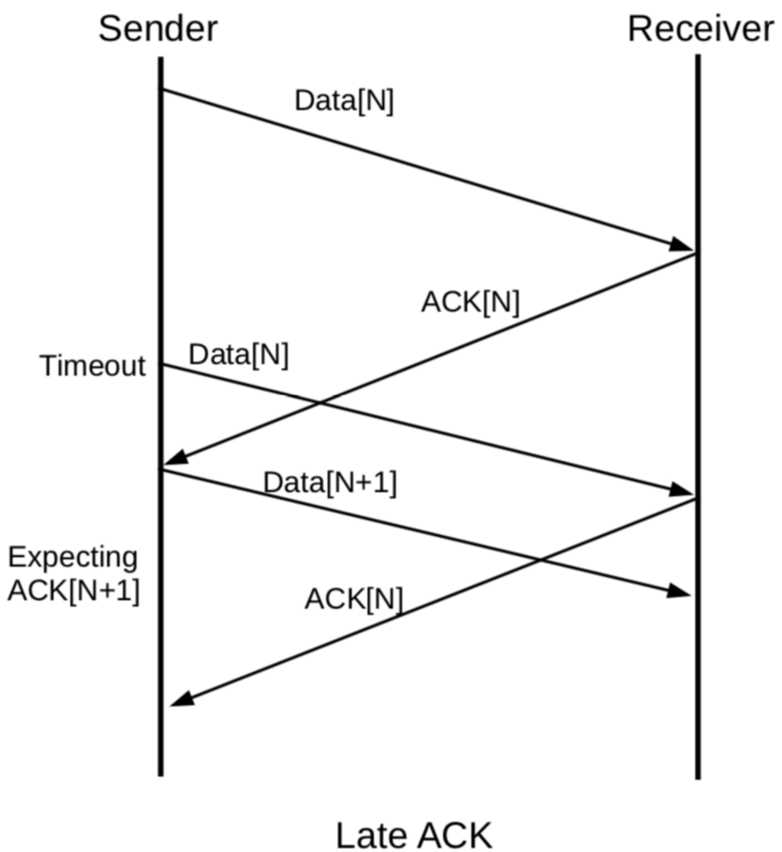
Lost Data

Lost ACK

- Sender cannot differentiate these two scenarios
  - Is the packet is lost?
  - Is the acknowledgement of the packet lost?

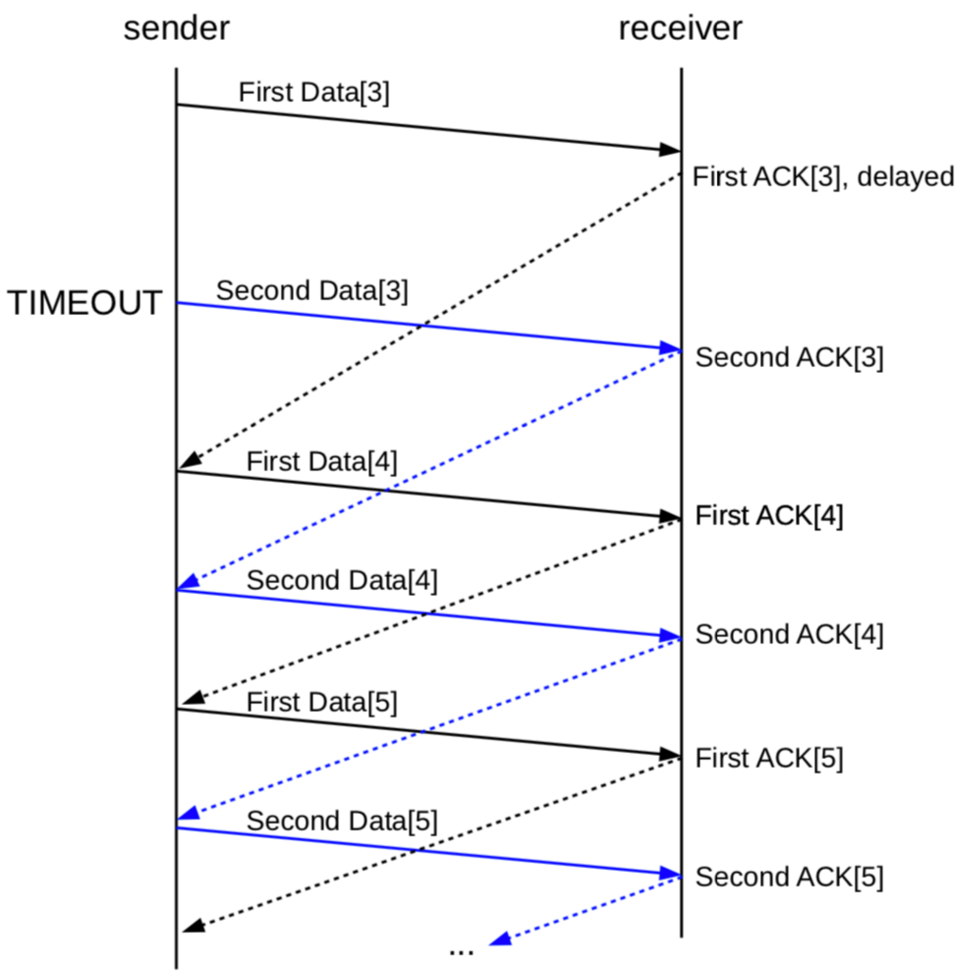
- If the acknowledgement is lost, the receiver would receive the same packet *twice*.
  - The receiver implements retransmission-on-duplicate strategy, i.e., it re-acknowledges the duplicated packet.

Stop-and-Wait: Packet Duplication



- Receiving a duplicate packet may have different reasons:
  - Acknowledgement of that packet was lost
  - Acknowledgement was delayed, and so the sender had timed out before receiving the acknowledgement.
  - The sender had prematurely timed out before receiving the on-time acknowledgement

Stop-and-Wait: Packet Duplication



- If both sender and receiver follow *retransmission-on-duplicate* strategy, upon receiving a delayed acknowledgement, every single packet would end up being transmitted multiple times
  - Significantly decreasing throughput

Closing Thoughts

Recap

- Today we discussed
  - TCP state system
  - TCP path MTU discovery
  - Reliable data transport
  - Stop-and-wait protocols

Next Class

- More TCP

Project 4

Due Nov 18th

Presentation

Due Nov 23rd