Computer Network I

Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale

(riccardo.caccavale@unina.it)

Università degli Studi di Napoli Federico II

DIETI UNINA

Transport Layer

TCP Problems in Connection Management

- Since TCP is connection-oriented **two hosts must agree** during both the **opening** and the **closing** procedures.

- Knowing that messages could be **lost or damaged** on the network, **it is quite difficult** for two hosts to find such agreement.

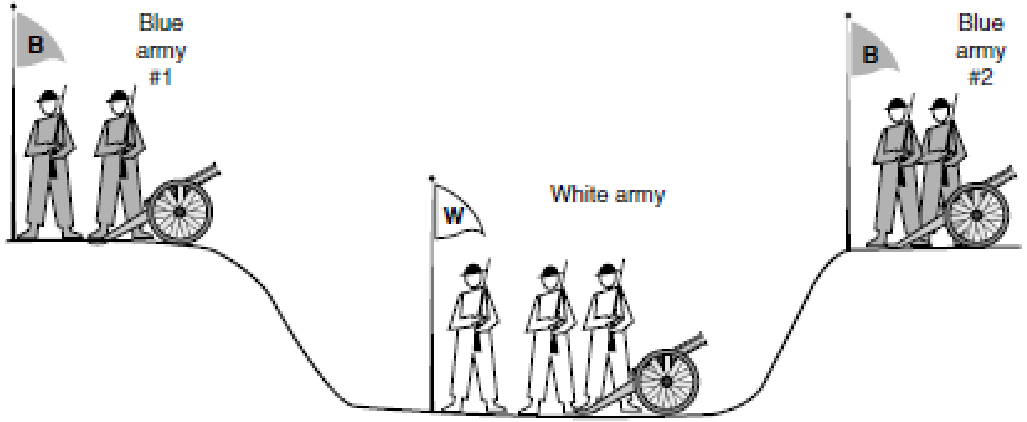
- If messages are lost during transmission **one end of the communication can be open or closed while the other is not**.

- To avoid (or better to mitigate) this issue TCP implements a procedure called **three-way handshake** in which open/close connection requests **have to be acknowledged** by hosts before a connection can be established/released.

Transport Layer

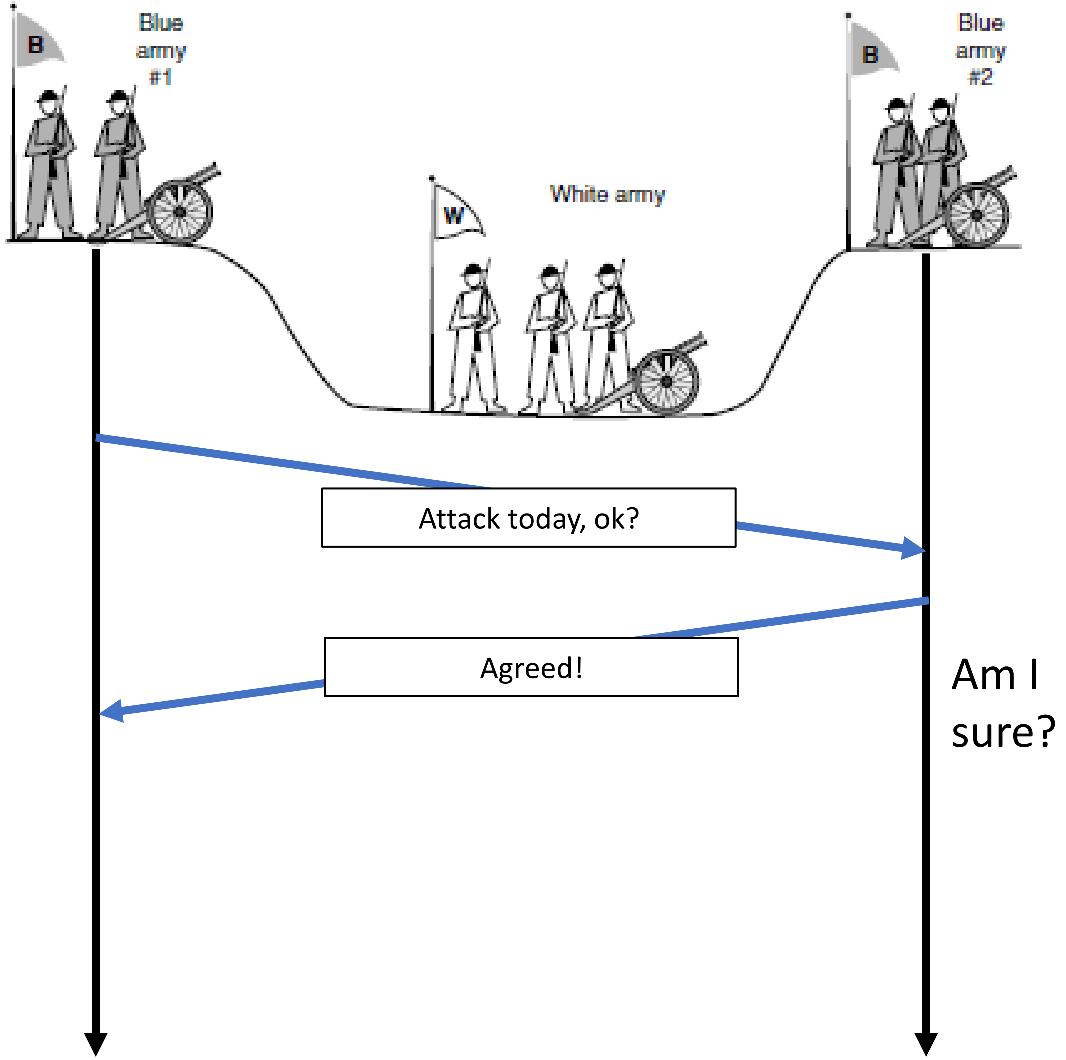
TCP Problems in Connection Management

- The two-army problem:
  - Imagine a **white army encamped in a valley** and, on both hillsides, there **are 2 enemy blue armies**.
  - The **white army is larger than either of the blue armies** alone, but together the blue armies are larger, they will be victorious only if the attack is simultaneous.
  - To synchronize their attacks, **blue armies must send messengers through the valley** where they might be captured (unreliable communication).
  - Does a protocol exist that allows the blue armies to win?



Transport Layer
TCP Problems in Connection Management

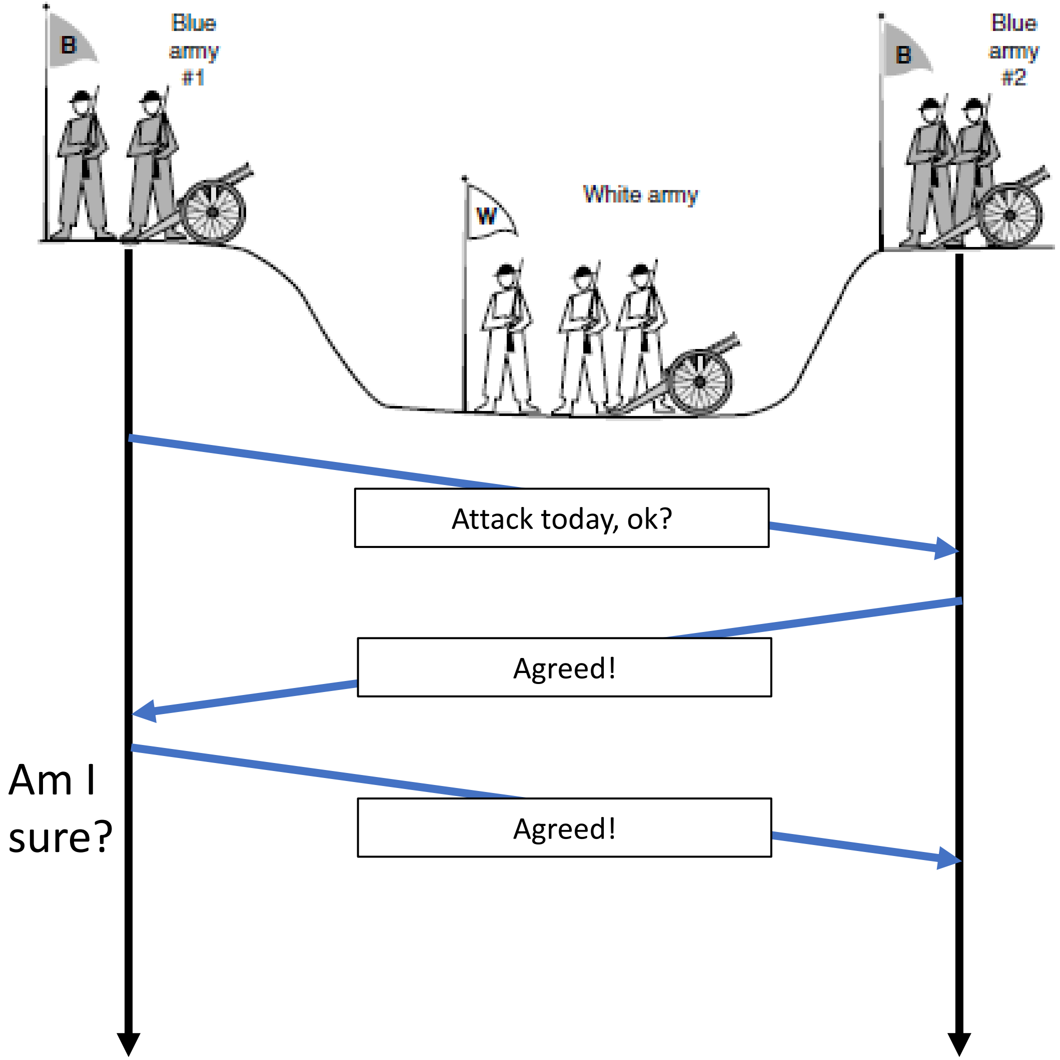
- Suppose that the commander of **blue army #1** sends a message reading: ''I propose we attack today, is it ok?''
- Now suppose that **the message arrives**, the commander of blue army #2 agrees, and his **reply gets safely back** to blue army #1.
- Will the attack happen? Probably not, because **commander #2 does not know if his reply got through**. If it did not, blue army #1 will not attack.



Transport Layer

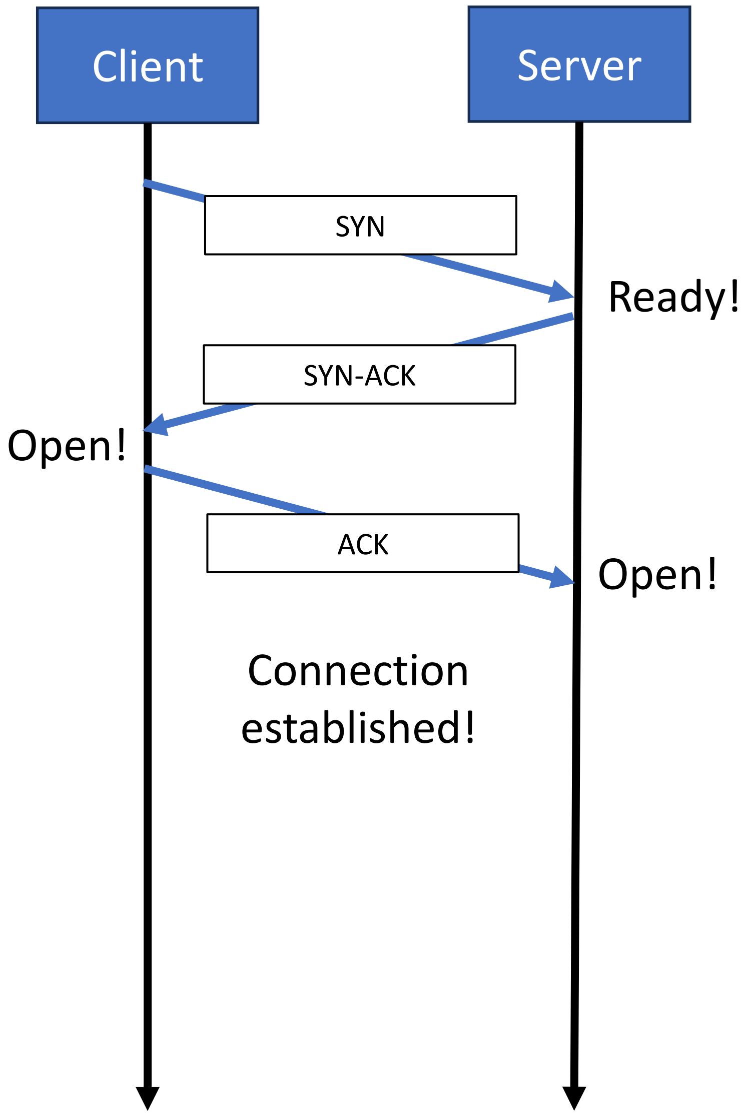
TCP Problems in Connection Management

- Let’s make it a **three-way handshake**: the first commander (of the original proposal) **must acknowledge** the response.
- Assuming **no messages are lost**, blue army #2 will get the acknowledgement, but the commander of blue army #1 will now hesitate (he does not know if his acknowledgement got through).
- Now **we could make it a four-way handshake**, but that does not help either. In fact, **it can be proven that no protocol exists that works**.
    - **Three-way handshake** is not perfect, but it is usually adequate!



Transport Layer
TCP Connection Establishment

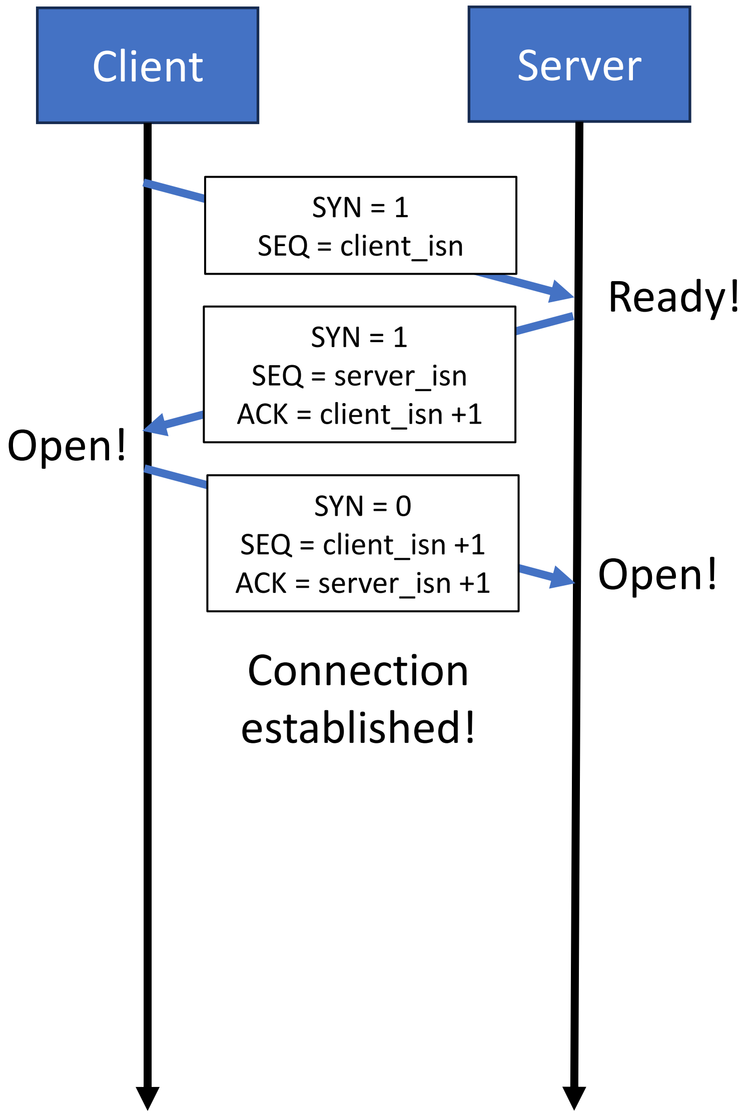
- In TCP **three-way handshake** is performed to establish connection:
  - Client send a **connection request** (SYN segment) to the server.
  - Server responds with a **special acknowledgment** (SYN-ACK segment).
  - Client sends back a **final acknowledgment** (ACK segment).
- TCP connection establishment is a **delicate procedure** that can also add significantly delays.
  - There is typically a **timeout** (30-60 secs) to complete the handshake, after that the procedure is aborted.
  - Some **attacks** (e.g., SYN flood) happen here.



Transport Layer

TCP Connection Establishment

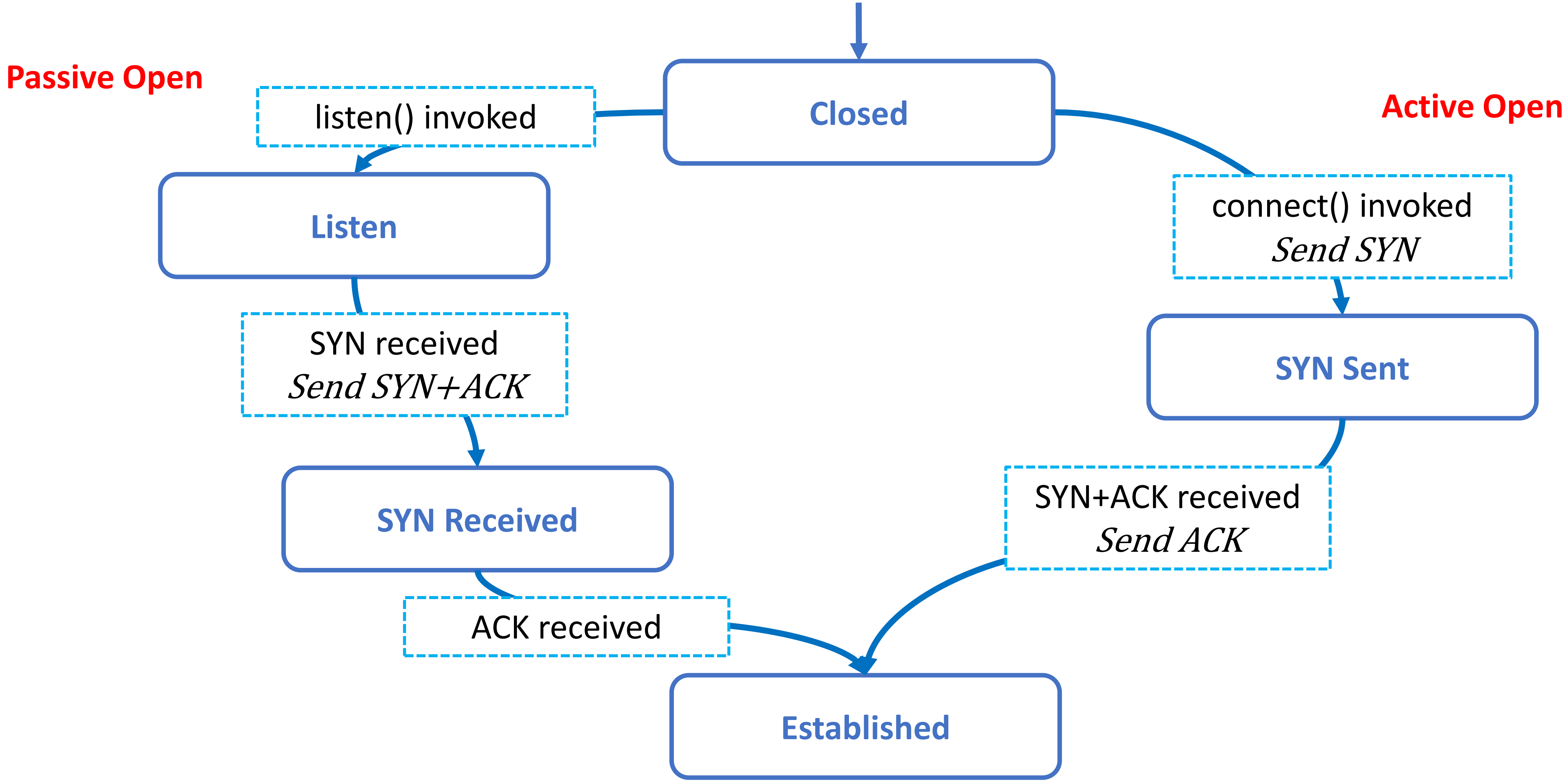
- Details of the **three-way handshake** procedure:
  1. The client sends a **SYN segment** having:
     - No application-layer data (payload).
     - The SYN bit set to 1.
     - A random initial sequence number (client_isn) as sequence number.
  2. When the above SYN segment (hopefully) arrives, the **server allocates TCP buffers and variables** and sends back a **SYNACK segment** having:
     - No application-layer data (payload).
     - The SYN bit set to 1.
     - The acknowledgment number set to client_isn+1.
     - A random initial sequence number (server_isn) as sequence number.
  3. When the SYNACK segment (hopefully) arrives, the **client also allocates buffers and variables** to the connection and sends to the server a final **ACK segment** having:
     - Possibly application-layer data.
     - The SYN bit set to 0 (connection is established).
     - The acknowledgment number set to server_isn+1.



Transport Layer

TCP Connection Establishment

- Simplified state diagram for TCP connection establishment.

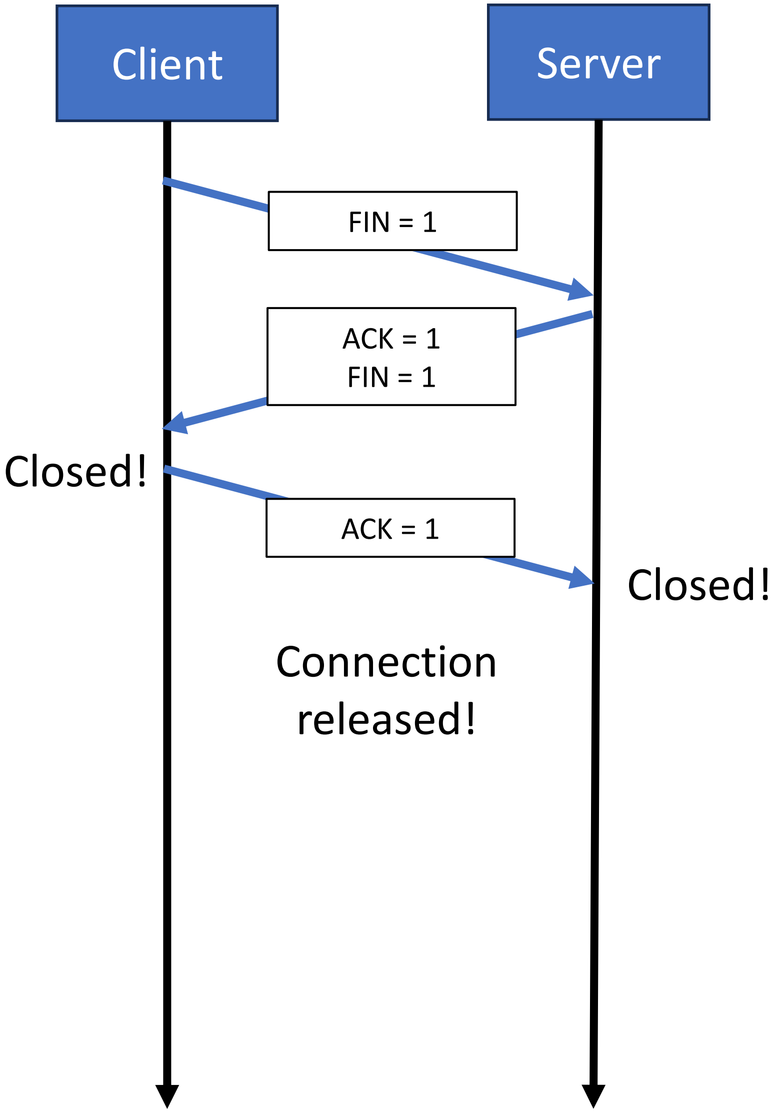


Transport Layer

TCP Connection Release

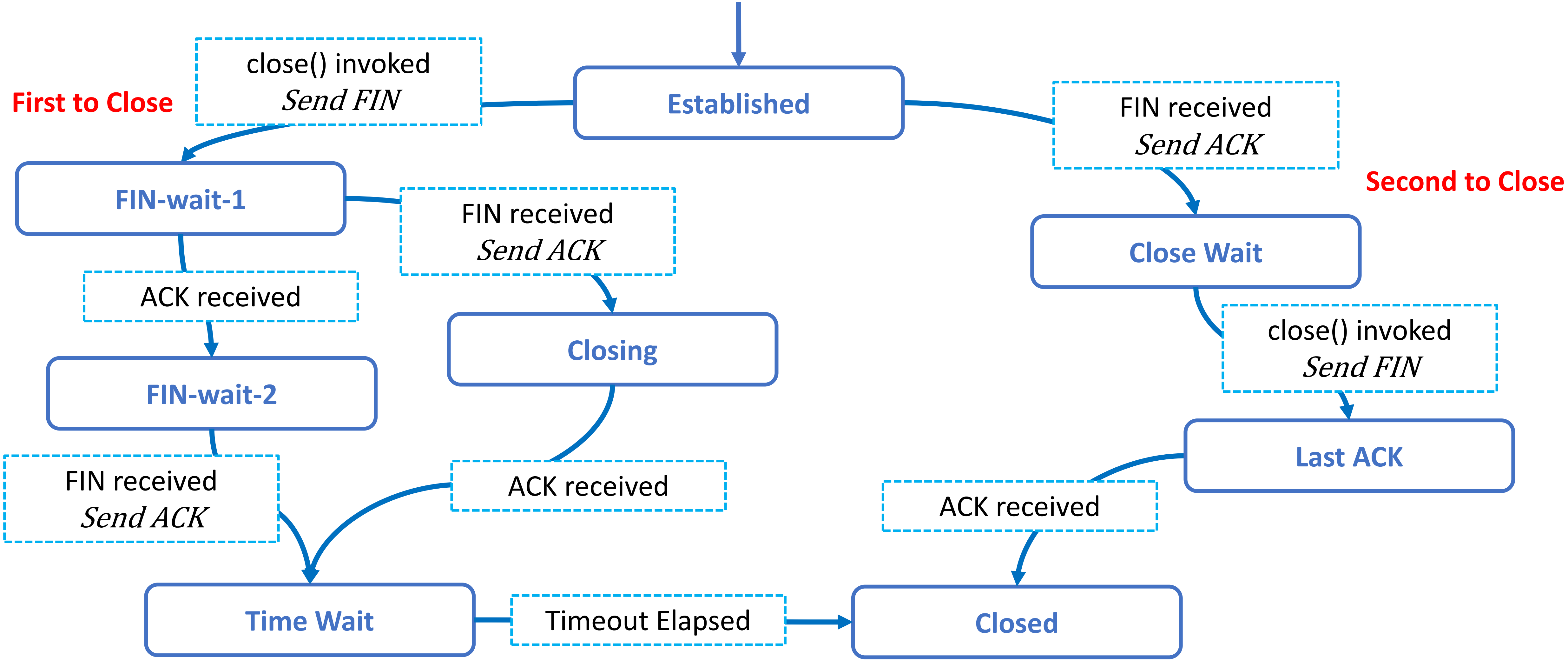
- TCP connection release (aka **teardown**) can be performed by either hosts.

- Let's assume client is closing the connection, the **three-way handshake** is performed as follow:
  1. The client sends a **special shutdown segment** (FIN segment) to the server having the FIN bit set to 1.
  2. When the server receives this segment, it **sends back an acknowledgment/shutdown segment**, having ACK to 1 and FIN bit set to 1.
     - Note: ACK and FIN can be sent in the same segment or in two separated ones.
  3. Finally, the client **acknowledges** the server's shutdown segment.



Transport Layer
TCP Connection Release

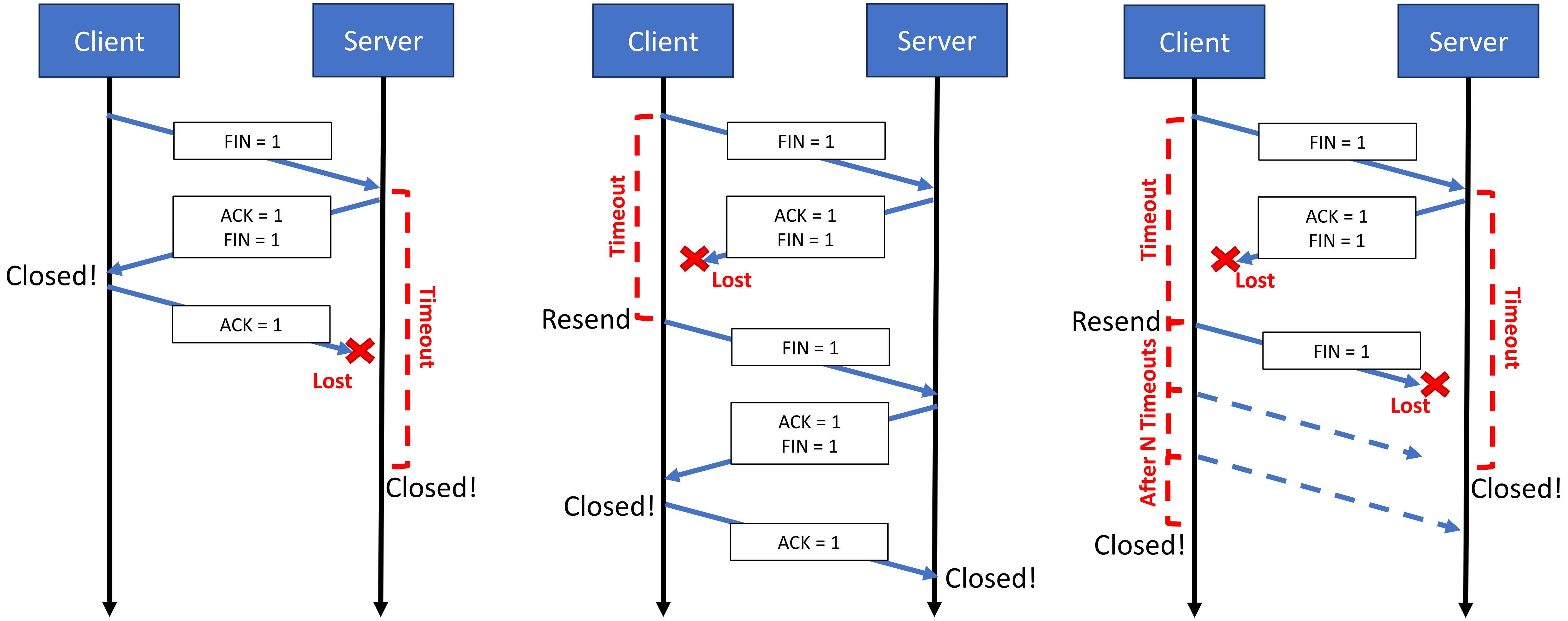
- Simplified state diagram for TCP connection release.



Transport Layer

TCP Connection Release

- How do we save ourselves from packet loss?
- Since three-way handshake is not perfect, TCP rely on timers to eventually close connections or to send again requests. Here some examples during connection release:



Transport Layer

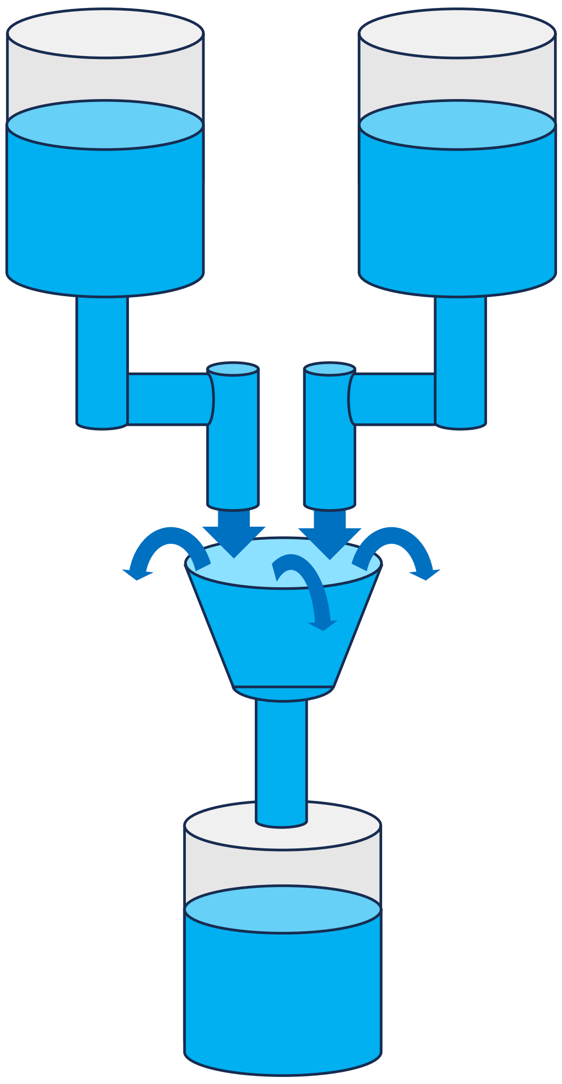
Congestion and Flow Control Problem

- Additional features of TCP with respect to UDP are the **congestion and flow control**.

- We mentioned that packet loss is **often a result of buffers' overflow**:
  - From receiver buffer, if **receiving application is not fast enough** in reading the data.
  - From network devices (e.g., routers), if **nodes are congested**.

- Following our previous water-flowing analogy, **we can see buffers as intermediate buckets** that overflows in case water exceeds.

- If packets are lost, **hosts are forced to rely on retransmission and timeouts** that drastically impair the network performance.



Transport Layer

Flow Control

- When the TCP connection receives bytes that are correct and in sequence, it places the data in the receive buffer. The receiving application **will read data from this buffer**, but **not necessarily at the instant the data arrives** (application may be busy).

- If the application is relatively **slow at reading the data**, the sender can very easily **overflow the receiver buffer** by sending too much data too quickly.

- TCP **flow control is a speed-matching service**: it attempts to match (reduce) the rate at which the sender is sending against the rate at which the receiving application is reading.
    - Remind: TCP segment **allow receiver to tell the sender** how much buffer has left through the **receive window field** (free buffer space).

Transport Layer
Flow Control

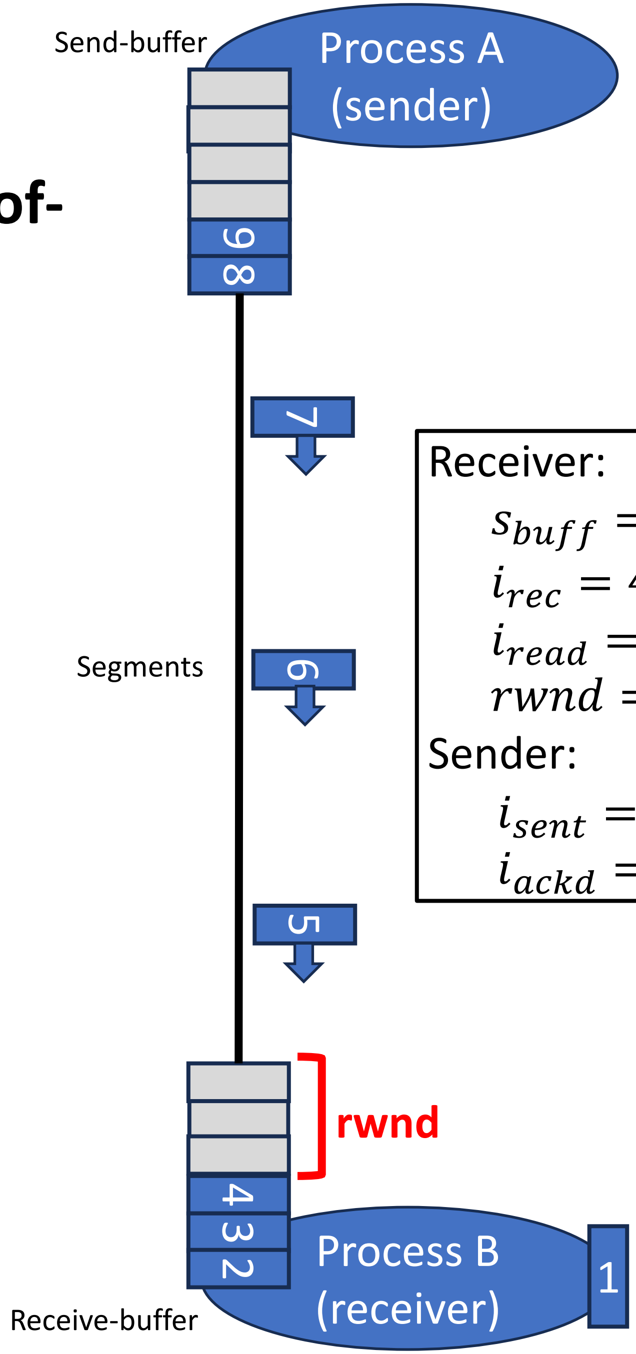
- Let's assume for simplicity that the **TCP receiver discards out-of-order segments**, so all segments in the buffer are ordered.

- On the **receiver side** (host B) we have:
  - s_{buff} size of the receiver buffer (in bytes)
  - i_{read} the last byte of the stream retrieved by the application.
  - i_{rec} the last byte of the stream that is received.
  - We may define the size of the **receive window** ($rwnd$) as:

$$rwnd = s_{buff} - (i_{rec} - i_{read})$$

- On the **sender side** (host A) we have:
  - i_{sent} the last byte of the stream to be sent.
  - i_{ackd} the last byte of the stream that is acknowledged by the receiver.
  - Knowing the receive window, the **sender guarantees anytime** that:

$$i_{sent} - i_{ackd} \leq rwnd$$



Receiver:

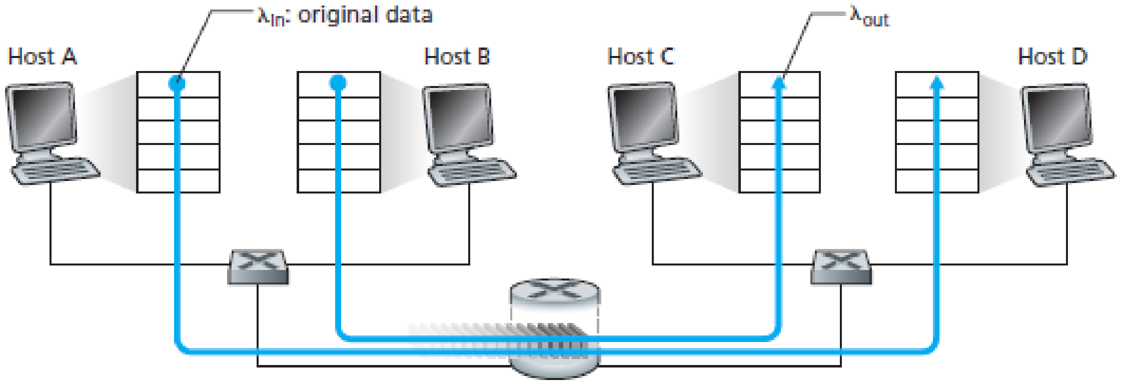
$s_{buff} = 6$
$i_{rec} = 4$
$i_{read} = 1$
$rwnd = 3$

Sender:

$i_{sent} = 7$
$i_{ackd} = 4$

Transport Layer
Congestion Problem

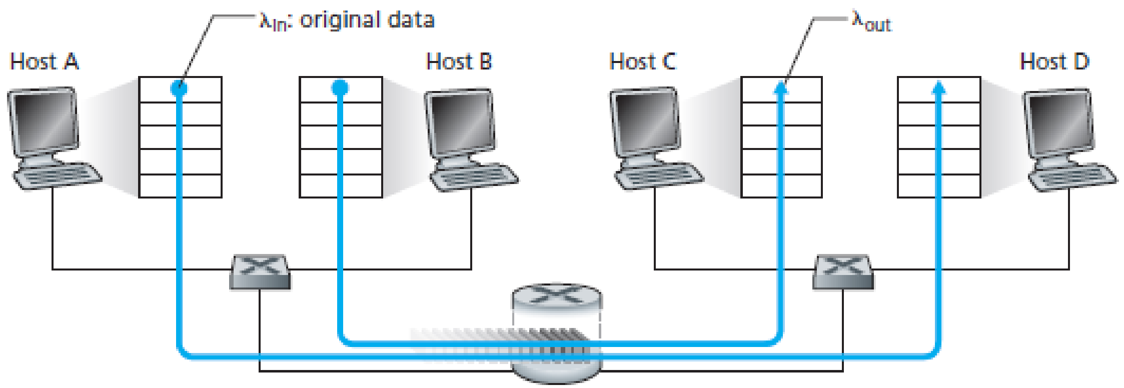
- The **congestion control problem** is similar to flow control, but it is related to the network infrastructure.



Router

- Example: two hosts (A and B) have a connection that **shares a single router** and a single outgoing link of capacity R between source and destination.
- The **router has buffers** that allow it to store incoming packets when the packet-arrival rate exceeds the outgoing link's capacity.
- Let's assume for simplicity that the both applications (in A and B) are sending data into the connection at **the same average rate** of λ_{in} bytes/sec.

- If $\lambda_{in} \leq R/\ 2$, everything sent by the sender is received by the receiver **with a finite delay**.
- If $\lambda_{in} > R/\ 2$, **the link reaches its full capacity** (R), and the exceeding packets are stored into the buffer.



Router

- As long as we exceed the maximum capacity the **packets will be accumulating into the router's buffer** waiting for their turn to be sent into the link.
- Since **buffer is finite**, accumulated **packets will eventually be discarded** and hosts will be forced to retransmit them (even more traffic).
- **Neither an infinite buffer works**, packets will not be lost but **the delay would constantly increase**: if we exceed the capacity forever, the delay will reach infinity (so packets are as good as lost).

Transport Layer

Congestion Control

- There are **mainly two approaches** to congestion control:
  1. **End-to-end congestion control**: this is the standard TCP approach where **congestion is inferred by the end systems** based only on observed network behavior (for example, packet loss and delay).
     - TCP segment loss (due to timeouts or the receipt of 3 duplicate ACKs) is taken as an indication of network congestion, and TCP decreases sending rate accordingly.
  2. **Network-assisted congestion control**: this is a recent (optional) approach where transport-layer works in synergy with network-layer. **Routers provide explicit feedback** to the sender and/or receiver regarding the congestion state of the network.
     - The feedback may be as simple as a **single bit indicating congestion** at a link, but more sophisticated feedback is also possible.

- **TCP mostly relies on end-to-end congestion control**.

- The approach is to have **each sender to adapt their sending rate** depending on the **perceived network congestion**.

- Here there are 3 main problems to be considered:
  - **Rate regulation**: how does a TCP sender regulate the rate at which it sends traffic into its connection?
  - **Congestion detection**: how does a TCP sender perceive the congestion on the path between itself and the destination?
  - **Rate adjustment**: what algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?

Transport Layer

TCP Congestion Control – Rate Regulation

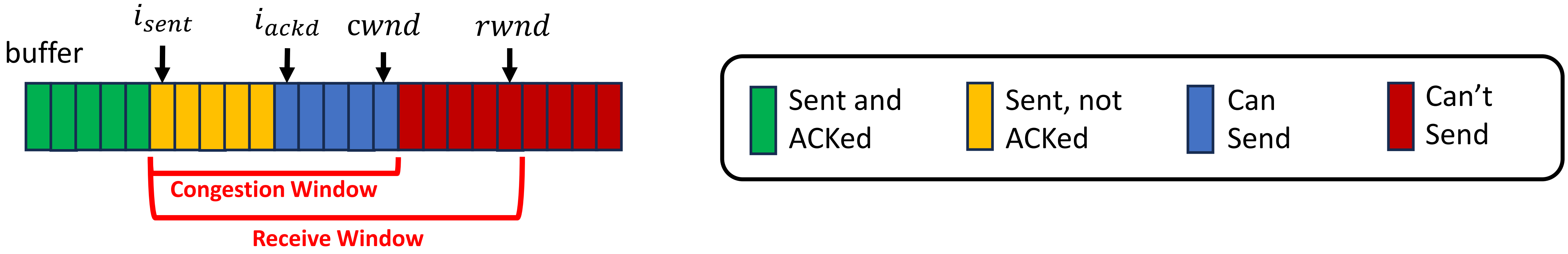
- Reminder: in TCP flow control **the sending rate is regulated by considering the buffer space** on the receiver side (receiver window):

$$i_{sent} - i_{ackd} \leq rwnd$$

- In TCP congestion-control the sender **also keeps track of a congestion window** ($cwnd$):

$$i_{sent} - i_{ackd} \leq \min(rwnd, cwnd)$$

- So, the rate is regulated by increasing/decreasing $cwnd$.



Transport Layer

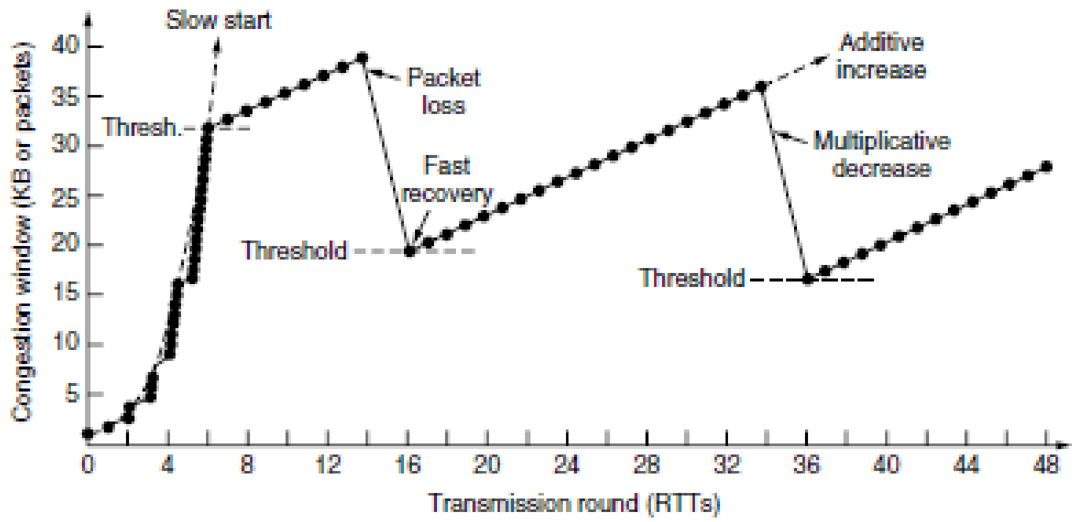
TCP Congestion Control – Congestion Detection

- A TCP sender perceives that there is congestion on the path between itself and the destination in two ways by checking ***loss events***.
- A *loss event* happens **when either a timeout is reached or 3 duplicate ACKs are received**.
  - Both events mean that a previous packet **is not arrived at the destination** probably because of network devices overflow.
- If **loss events do not occur**, i.e., ACKs of segments arrive as expected, TCP will assume that network is not congested, so the **congestion window can be increased**.
- If, on the other hand, **loss events occur**, the **congestion window must be decreased**.

Transport Layer

TCP Congestion Control – Rate Adjustment

- TCP rate adjustment is then performed through **bandwidth probing**: rate is increased as long as ACKs arrive correctly (probing the network), when congestion is detected (loss events) the rate is decreased. This process is continuously repeated.
- TCP uses **Jacobsen congestion-control algorithm [Jacobson 1988] to regulate the rate**, it includes 3 phases:
  1. **Slow start**: start from 1 MSS/RRT increase rate exponentially.
  2. **Additive Increase** (or congestion avoidance): increase rate linearly.
  3. **Fast Recovery** (optional): halves the rate instead of slow-starting again and proceed with additive increment.



Typical sawtooth behavior of the Jacobsen algorithm

- Simplified state diagram representing the Jacobson algorithm.