TCP Congestion Control

Congestion Window

- Sender window size is controlled by the receiver by advertising *rwnd*
- It doesn’t guarantee that the intermediate routers doesn’t get congested
- A router may receive data from more than one sender
- There may be no congestion at the end, but the congestion may occur in middle
- More segment loss results in resending the same segment, which results in worsening the congestion, and finally the collapse of the communication
- TCP is an end-to-end protocol that uses the service of IP, and therefore; congestion happens in IP side and should be taken care by it
- IP is a simple protocol with no congestion control, and therefore; TCP must take care of the congestion

TCP Congestion Control

- **Congestion Window**
  - TCP cannot ignore the congestion in the network and should not aggressively sends the segments
  - TCP cannot be too conservation either, and start sending small number of segments in each time interval as this would results in under-utilizing the network resources
  - To control the number of segments to transmit, TCP uses another variable called **congestion window**, *cwnd*
  - A *rwnd* and *cwnd* together define the sender window
  - A *rwnd* is associated with the congestion at the end, and *cwnd* at the middle

Actual sender window size = minimum (*rwnd, cwnd*)

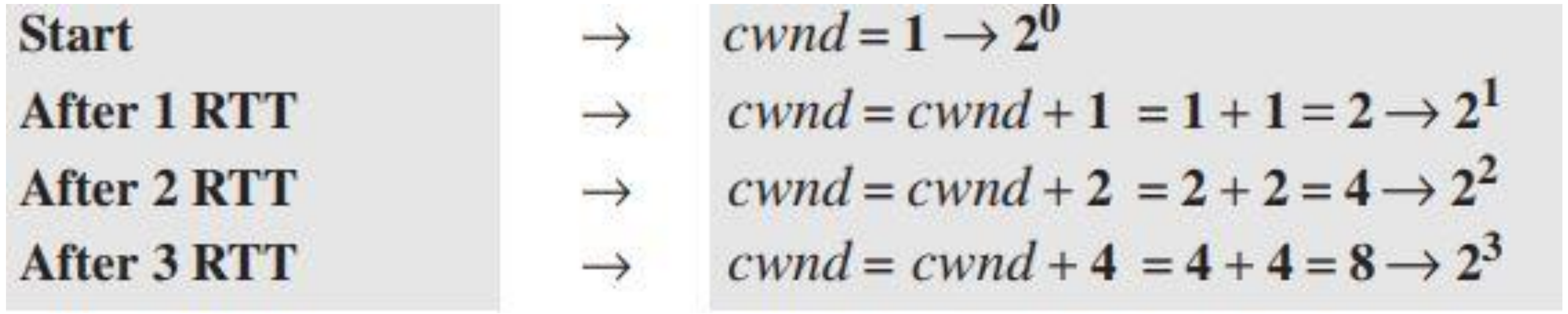
TCP Congestion Control

- **Congestion Detection**
  - **Time-out and three duplicate ACKs**
  - If a TCP sender does not receive an ACK for a segment or a group of segments before the *time-out* occurs, it assumes that the corresponding segment or segments are lost due to congestion
  - Sending *three duplicate ACKs* is sign of missing segments due to congestion
  - Congestion due to *three duplicate ACKs* is less severe than congestion due to *time-out*
  - TCP version named **Taho** treat *time-out* and *three duplicate ACKs* in a similar way
  - Later version of TCP named **Reno** treats them differently
  - TCP use only ACKs to detect congestion, *time-out* signifies the case of strong congestion, and *three duplicate ACKs* is the sign of weak congestion in the network

Congestion Policies

- **Slow-start, congestion avoidance, and congestion detection**

- **Slow-start: Exponential Increase**
  - Size of *cwnd* starts with 1 MSS and increases as acknowledgment arrives
  - Assumption: 1) Each segment is of same size, 2) Each segment is acknowledged individually



Start	$\rightarrow$	$cwnd = 1 \rightarrow 2^0$
After 1 RTT	$\rightarrow$	$cwnd = cwnd + 1 \ = 1 + 1 = 2 \rightarrow 2^1$
After 2 RTT	$\rightarrow$	$cwnd = cwnd + 2 \ = 2 + 2 = 4 \rightarrow 2^2$
After 3 RTT	$\rightarrow$	$cwnd = cwnd + 4 \ = 4 + 4 = 8 \rightarrow 2^3$

Fig 1. RTT and cwnd

Slow start: Exponential Increase

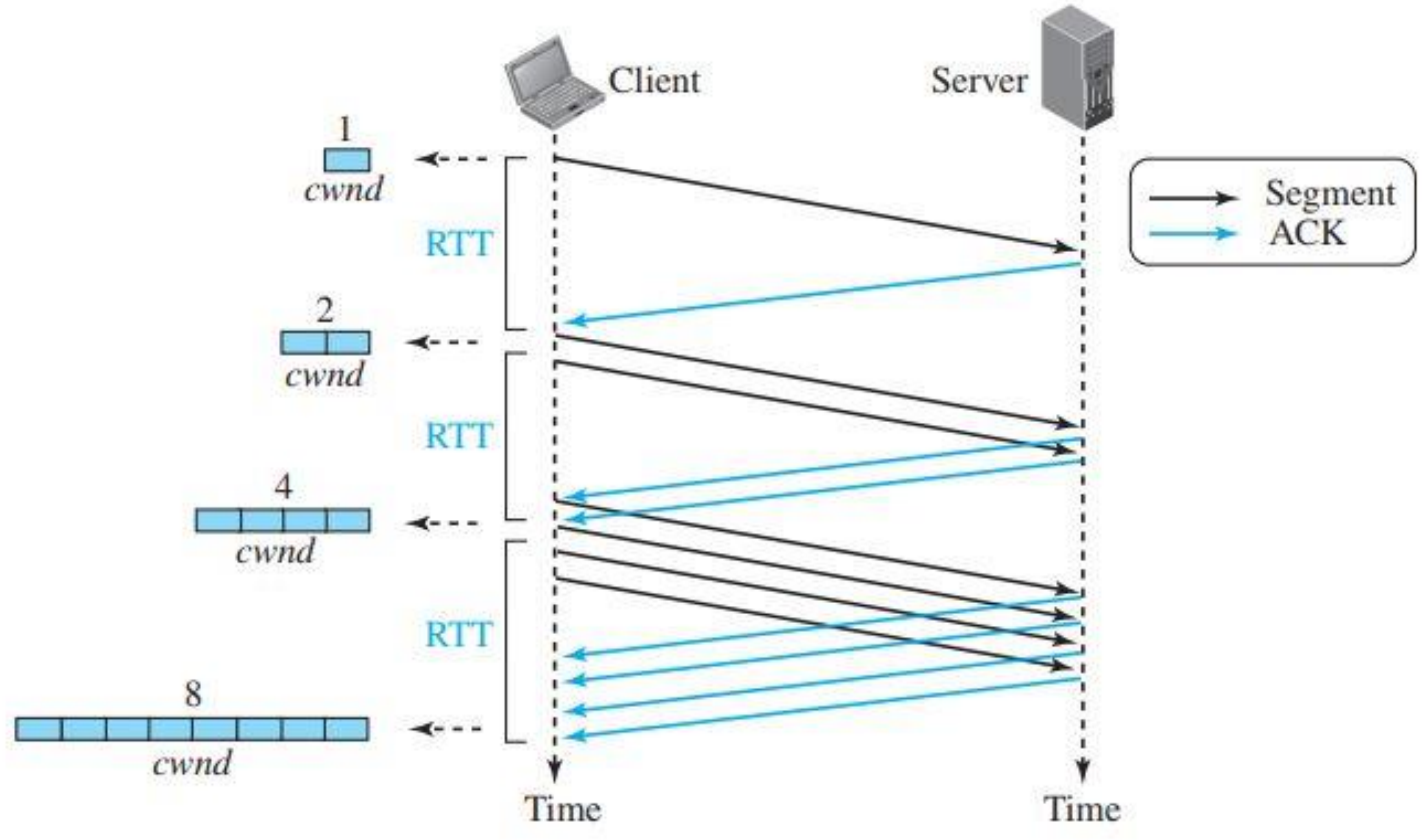


Fig 2. Slow start process

Slow start: Exponential Increase

- **Slow start process doesn’t continue indefinitely and there is threshold to stop it called *slow-start threshold (ssthresh = rwnd/2)***
- **In slow-start, size of congestion window increases exponentially**
- **When the size of the window in bytes reaches *ssthresh*, slow-start stops, and next phase starts**

Slow start: Exponential Increase

- **Example:**
  - *rwnd* (W_r) = *64 KB, cwnd* (W_c) = *1 KB, sender window* (W_s) = *1 KB*
  - *segment = 1 KB*
  - *ssthresh = rwnd/2 = 32 KB*
  - $W_s = 1, 2, 4, 8, 16, 32 \rightarrow$ *Slow-start phase stops*

Congestion Avoidance: Additive Increase

- **To avoid exponential growth of the *cwnd* which may cause congestion, another phase starts called *congestion avoidance***
- **In congestion avoidance, size of *cwnd* increases additively**
- **Each time when whole window of segments is acknowledged, the size of window is increased by 1**
- **Example:**
  - *rwnd (W_r) = 64 KB, cwnd (W_c) = 1 KB, sender window (W_s) = 1 KB*
  - *segment = 1 KB*
  - *ssthresh = rwnd/2 = 32 KB*
  - $W_s = 1, 2, 4, 8, 16, 32$ → *Slow-start phase*
  - $W_s = 33, 34, 35, \ldots\ldots, 64$ → *Congestion avoidance phase*

Congestion Avoidance: Additive Increase

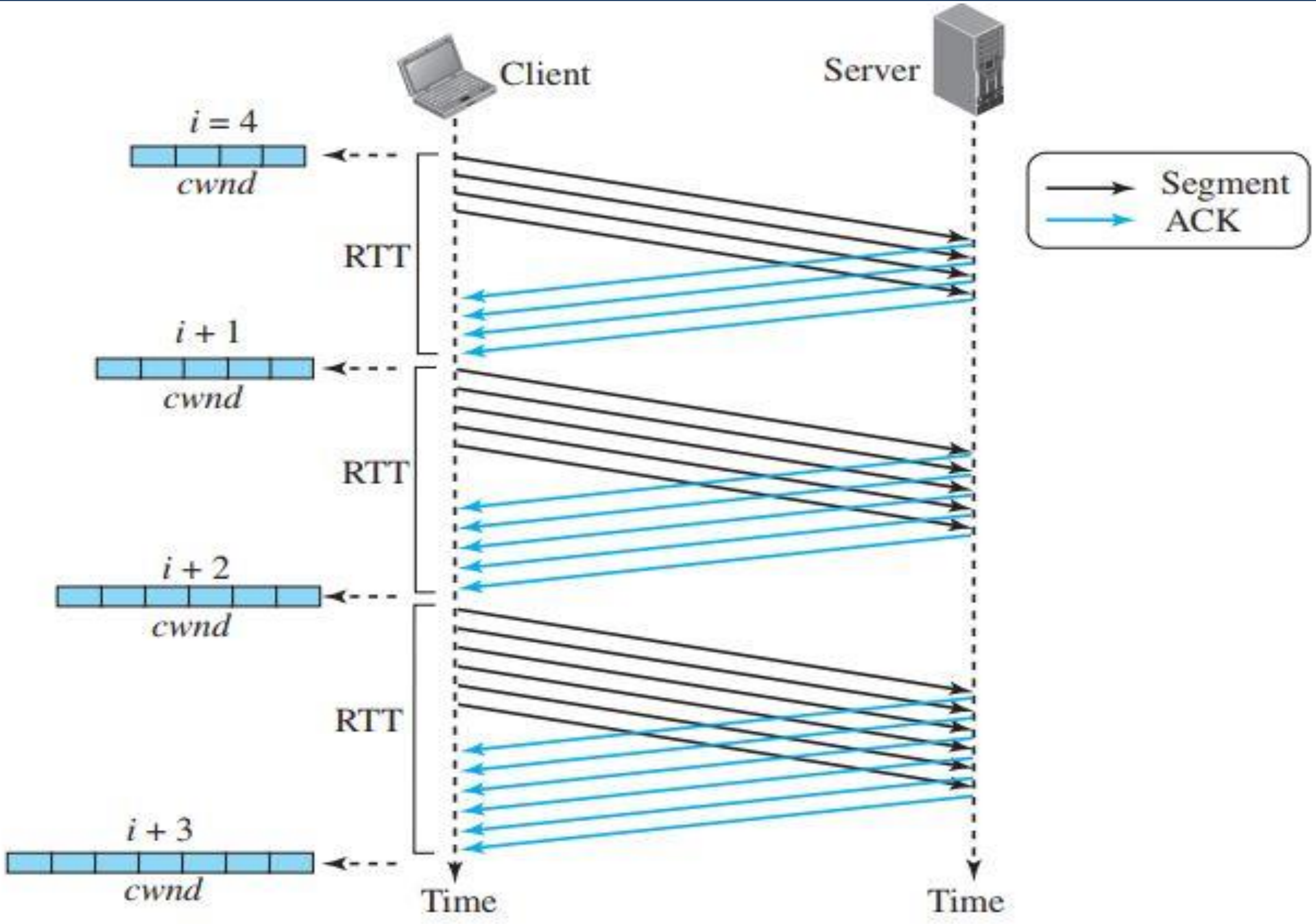
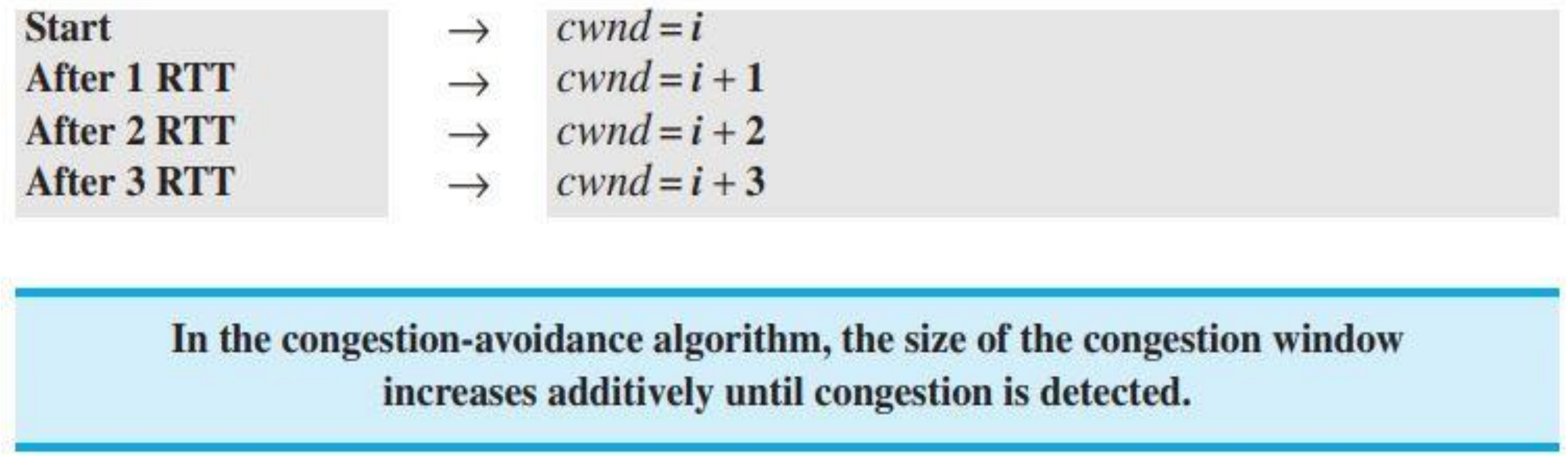


Fig 3. Congestion avoidance

Congestion Avoidance: Additive Increase



Start	$\rightarrow$	$cwnd = i$
After 1 RTT	$\rightarrow$	$cwnd = i + 1$
After 2 RTT	$\rightarrow$	$cwnd = i + 2$
After 3 RTT	$\rightarrow$	$cwnd = i + 3$

In the congestion-avoidance algorithm, the size of the congestion window increases additively until congestion is detected.

Fig 4. Congestion avoidance

Congestion Detection: Multiplicative Decrease

- **If time-out occurs, there is strong possibility of congestion, and segment has probably been dropped in the network**
  - *ssthresh = cwnd/2*
  - *cwnd = 1*
  - *Starts slow-start phase again*

- **If three ACKs are received, there is weaker chance of congestion as a segment may have been dropped but some segments after that may have been received. This is called fast retransmission and fast recovery**
  - *ssthresh = cwnd/2*
  - *cwnd = ssthresh*
  - *It starts the congestion avoidance phase*

Congestion Example

- We assume that the maximum window size *(rwnd)* is 32 segments. The threshold is set to 16 segments (one-half of the maximum window size). In the *slow-start* phase, the window size starts from 1 and grows exponentially until it reaches the threshold. After it reaches the threshold, the *congestion avoidance (additive increase)* procedure allows the window size to increase linearly until the congestion is detected or the maximum window size is reached. In Figure 5, the time-out occurs when the window size is 20. At this moment, the *multiplicative decrease* procedure takes over and reduces the threshold to one-half of the previous window size. The previous window size was 20 when the time-out happened so the new threshold is now 10.

- TCP moves to slow start again and starts with a window size of 1, and TCP moves to additive increase when the new threshold is reached. When the window size is 12, a three-ACKs event happens. The multiplicative decrease procedure takes over again. The threshold is set to 6 and TCP goes to the additive increase phase this time. It remains in this phase until another time-out or another three ACKs happen

Congestion Example

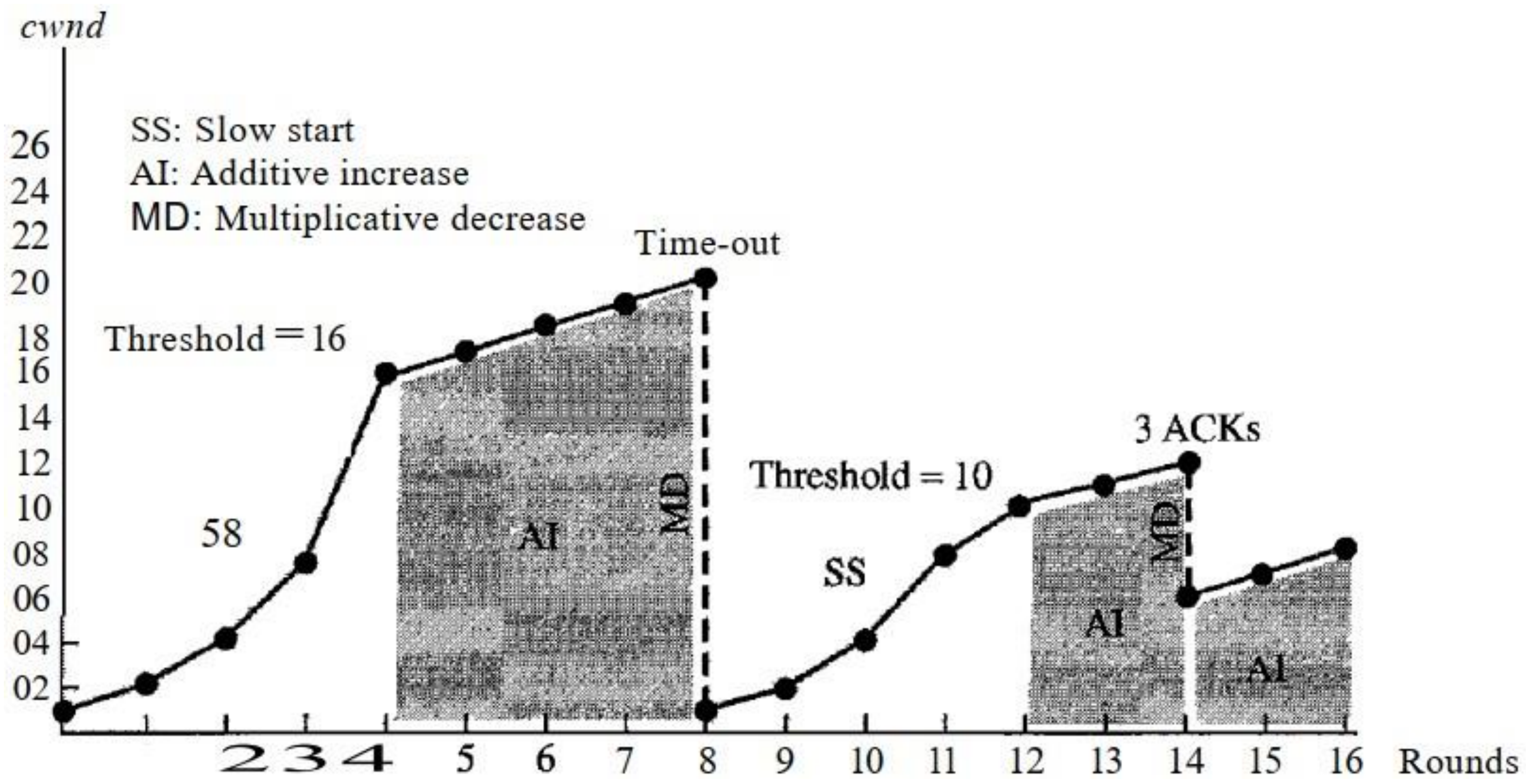


Fig 5. Congestion Example

Quality of Service (QoS)

- **QoS is the ability to provide different priority to different applications, users, or data flows, to guarantee a certain level of performance**

- **Flow characteristics**
  - ***Reliability***
  - ***Delay***
  - ***Jitter***
  - ***Bandwidth***

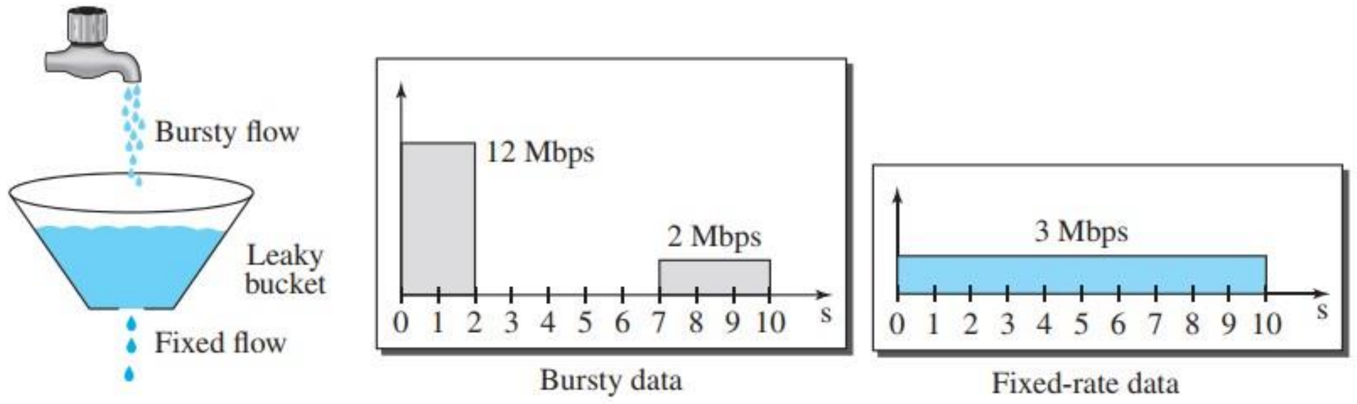
- **Techniques to improve QoS**

- **Scheduling:** ***FIFO Queueing, Priority Queueing (suffers from starvation), Weighted Fair Queueing***

Quality of Service (QoS)

- **Traffic Shaping**

- **Leaky bucket: Input rate varies, output rate remain constant**



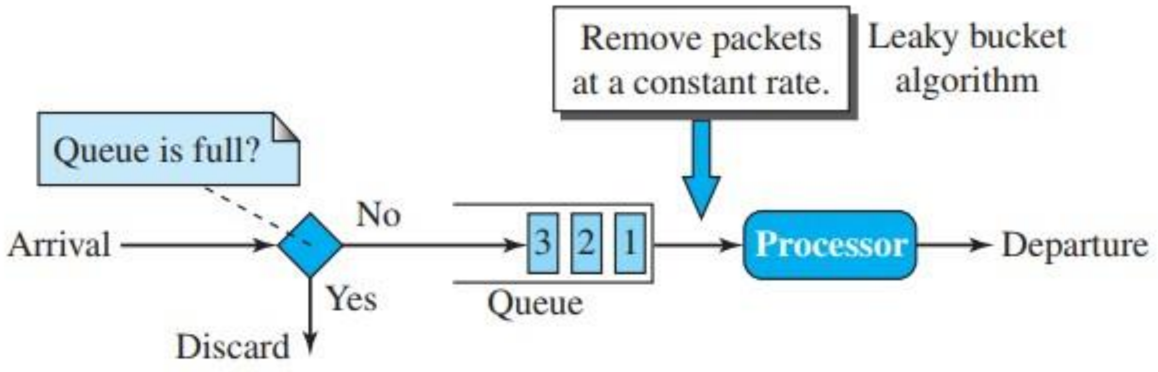
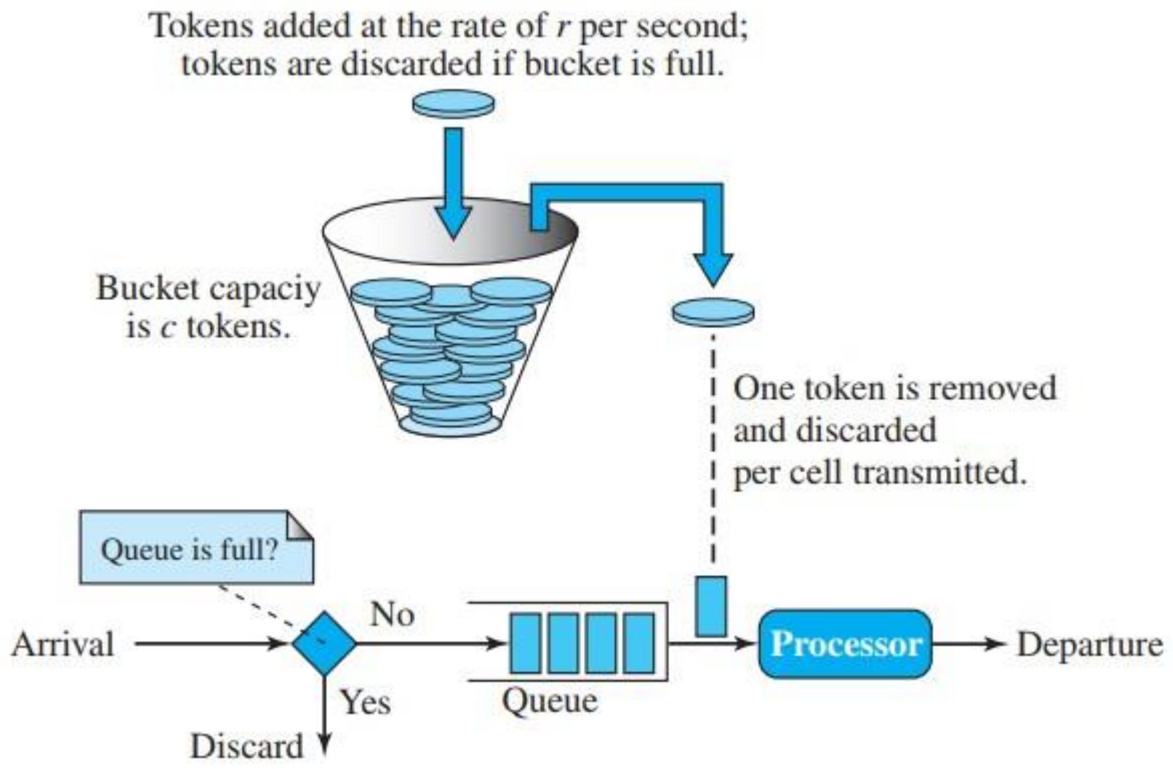


Fig 6. Leaky Bucket

Quality of Service (QoS)

- **Leaky bucket is restrictive and doesn't give credit to an idle host**
- **Token bucket allows idle host to accumulate traffic for the future in the form of tokens**



Tokens added at the rate of *r* per second; tokens are discarded if bucket is full.
Bucket capaciy is *c* tokens.
One token is removed and discarded per cell transmitted.
Queue is full?
No
Yes
Arrival
Discard
Queue
Processor
Departure

Fig 7. Token Bucket

- **Resource Reservation and Admission Control**

Advantage of token bucket over leaky bucket

- If a bucket is full in tokens bucket, tokens are discarded not packets. While in leaky bucket, packets are discarded.
- Token Bucket can send large bursts at a faster rate while leaky bucket always sends packets at constant rate.
- **Predictable Traffic Shaping:** Token Bucket offers more predictable traffic shaping compared to leaky bucket. With token bucket, the network administrator can set the rate at which tokens are added to the bucket, and the maximum number of tokens that the bucket can hold. This allows for better control over the network traffic and can help prevent congestion.
- **Better Quality of Service (QoS):** Token Bucket provides better QoS compared to leaky bucket. This is because token bucket can prioritize certain types of traffic by assigning different token arrival rates to different classes of packets. This ensures that important packets are sent first, while less important packets are sent later, helping to ensure that the network runs smoothly.
- **More efficient use of network bandwidth:** Token Bucket allows for more efficient use of network bandwidth as it allows for larger bursts of data to be sent at once. This can be useful for applications that require high-speed data transfer or for streaming video content.
- **More granular control:** Token Bucket provides more granular control over network traffic compared to leaky bucket. This is because it allows the network administrator to set the token arrival rate and the maximum token count, which can be adjusted according to the specific needs of the network.
- **Easier to implement:** Token Bucket is generally considered easier to implement compared to leaky bucket. This is because token bucket only requires the addition and removal of tokens from a bucket, while leaky bucket requires the use of timers and counters to determine when to release packets.

Disadvantage of token bucket over leaky bucket

- **Tokens may be wasted:** In Token Bucket, tokens are generated at a fixed rate, even if there is no traffic on the network. This means that if no packets are sent, tokens will accumulate in the bucket, which could result in wasted resources. In contrast, with leaky bucket, the network only generates packets when there is traffic, which helps to conserve resources.
- **Delay in packet delivery:** Token Bucket may introduce delay in packet delivery due to the accumulation of tokens. If the token bucket is empty, packets may need to wait for the arrival of new tokens, which can lead to increased latency and packet loss.
- **Lack of flexibility:** Token Bucket is less flexible compared to leaky bucket in terms of shaping network traffic. This is because the token generation rate is fixed and cannot be changed easily to meet the changing needs of the network. In contrast, leaky bucket can be adjusted more easily to adapt to changes in network traffic.
- **Complexity:** Token Bucket can be more complex to implement compared to leaky bucket, especially when different token generation rates are used for different types of traffic. This can make it more difficult for network administrators to configure and manage the network.
- **Inefficient use of bandwidth:** In some cases, Token Bucket may lead to inefficient use of bandwidth. This is because Token Bucket allows for large bursts of data to be sent at once, which can cause congestion and lead to packet loss. In contrast, leaky bucket helps to prevent congestion by limiting the amount of data that can be sent at any given time.

Application Layer

- **HTTP**
- **DNS**
- **FTP**
- **SMTP**
- **SNMP**

❖ ***Please go through the Forouzan Book (4th and 5th edition). Go through the notes that are shared in the classroom portal***