POLICY TRANSITION

- We discussed three congestion policies in TCP. Now the question is when each of these policies are used and when TCP moves from one policy to another.
- To answer these questions, we need to refer to three versions of TCP:
  - Taho TCP
  - Reno TCP
  - New Reno TCP.

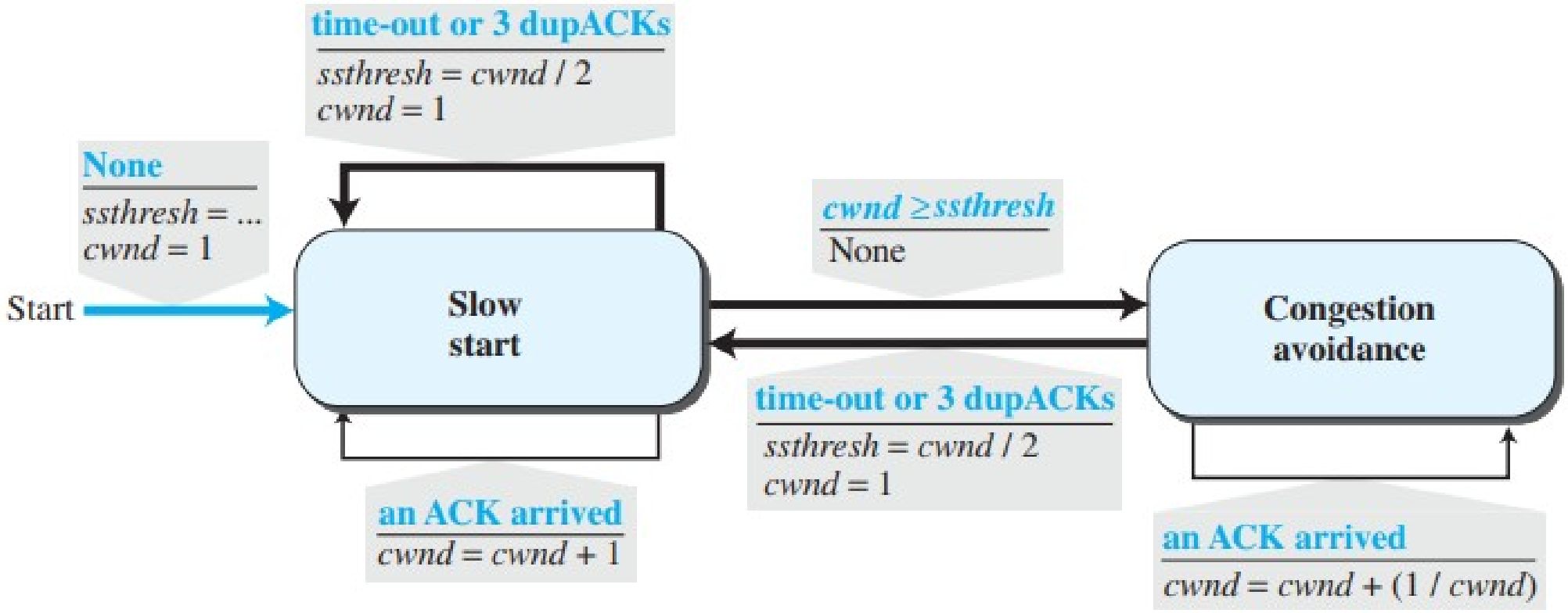
TAHO TCP

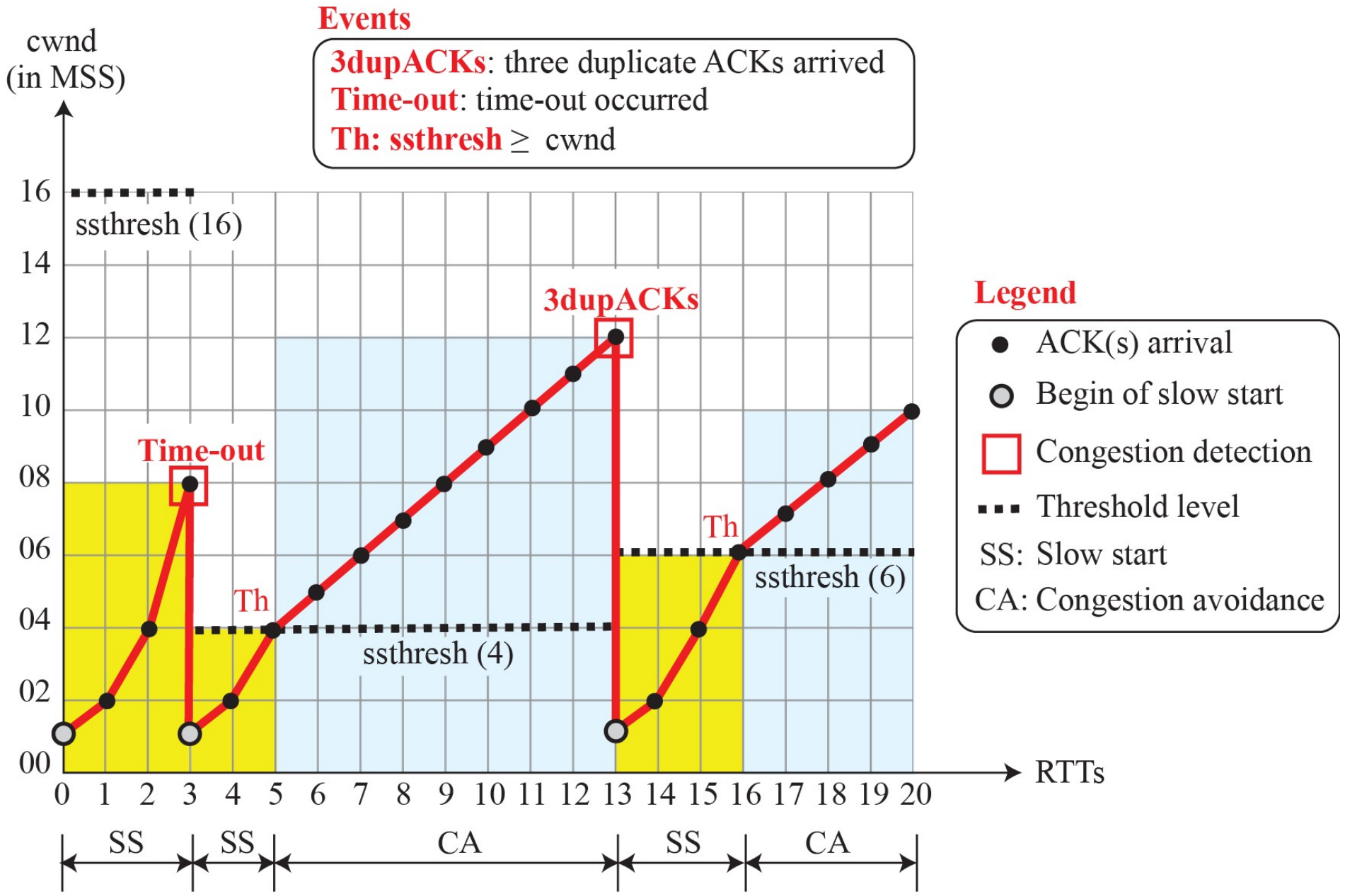
- The early TCP, known as Taho TCP, used only two different algorithms in their congestion policy: *slow start* and *congestion avoidance*.
- Taho TCP treats the two signs used for congestion detection, time-out and three duplicate ACKs, in the same way.
- In this version, when the connection is established, TCP starts the slow-start algorithm and sets the ssthresh variable to a pre-agreed value (normally multiple of MSS) and the cwnd to 1 MSS.
- If congestion is detected (occurrence of time-out or arrival of three duplicate ACKs), TCP immediately interrupts this aggressive growth and restarts a new slow start algorithm by limiting the threshold to half of the current cwnd and resetting the congestion window to 1.

TAHO TCP

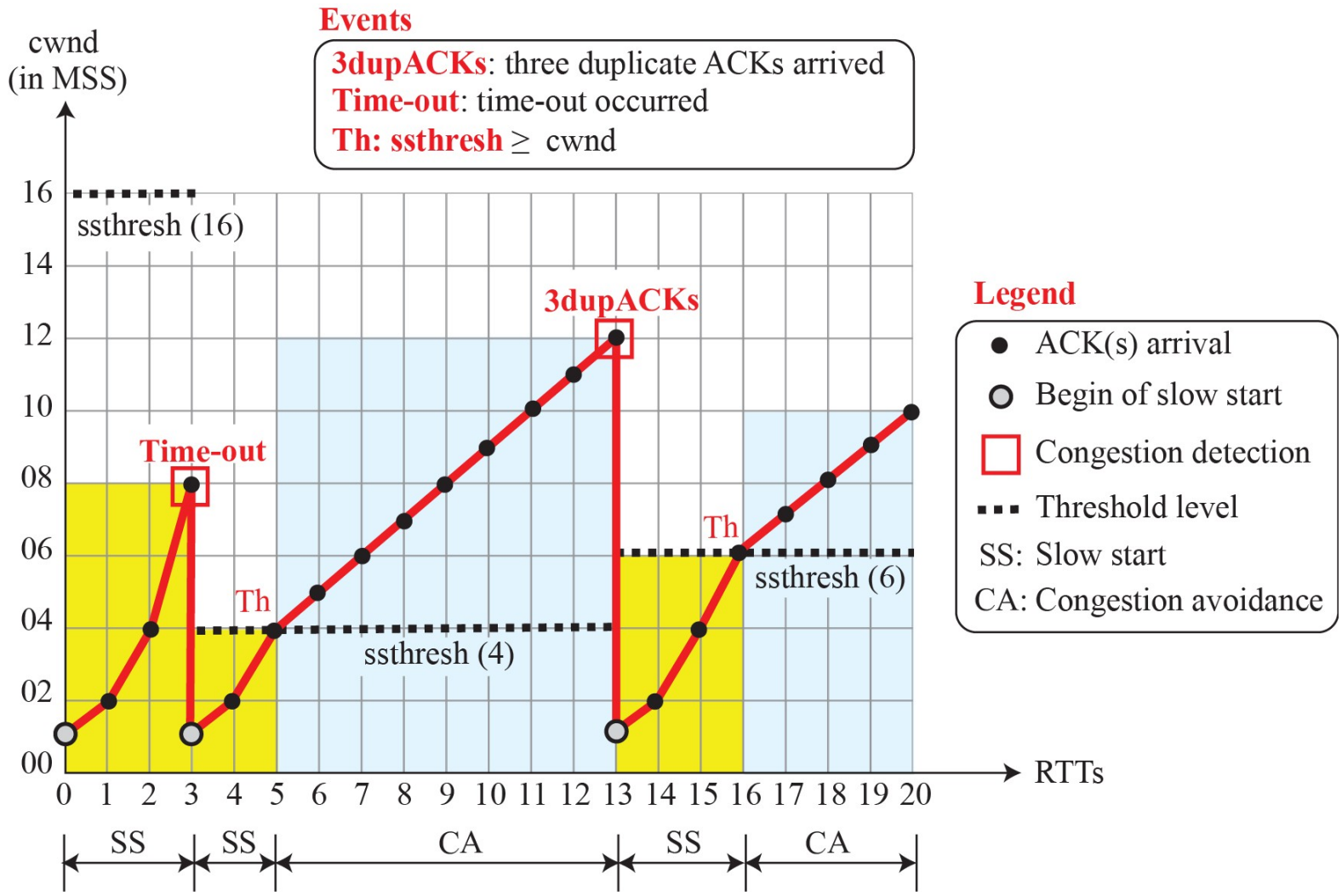
- In other words, not only does TCP restart from scratch, but it also learns how to adjust the threshold.
- If no congestion is detected while reaching the threshold, TCP learns that the ceiling of its ambition is reached; it should not continue at this speed. It moves to the congestion avoidance state and continues in that state.
- In the congestion-avoidance state, the size of the congestion window is increased by 1 each time a number of ACKs equal to the current size of the window has been received.
- Note that there is no ceiling for the size of the congestion window in this state; the conservative additive growth of the congestion window continues to the end of the data transfer phase unless congestion is detected.
- If the congestion is detected in this state, TCP again resets the value of the ssthresh to half of the current cwnd and moves to the slow-start state again.

TAHO TCP: FSM

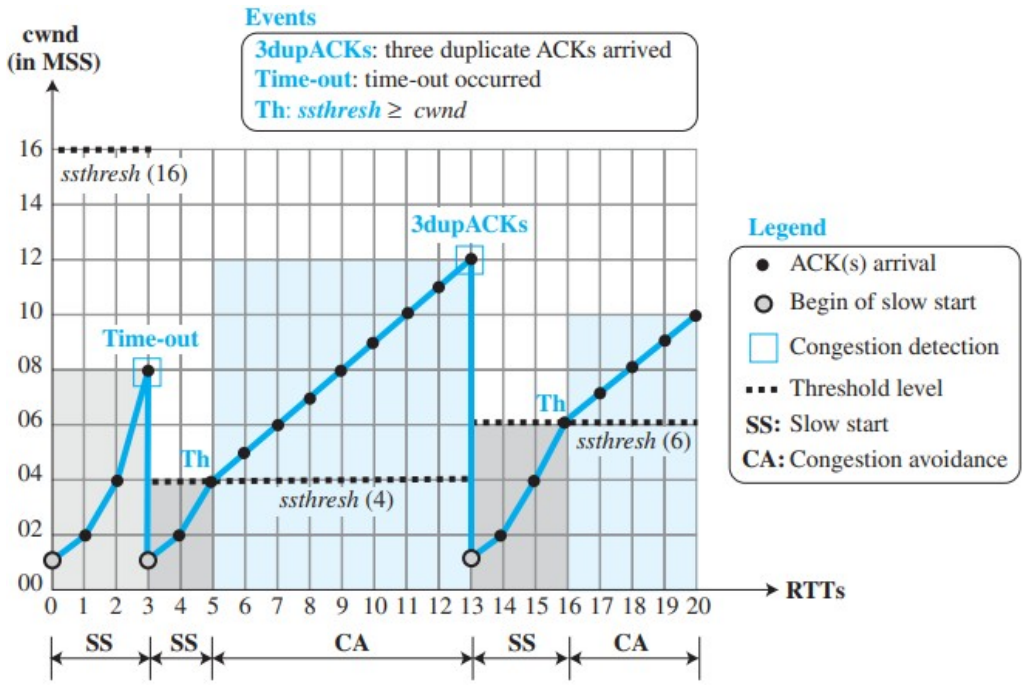

time-out or 3 dupACKs
ssthresh = cwnd / 2
cwnd = 1
None
ssthresh = ...
cwnd = 1
Start
Slow start
cwnd ≥ ssthresh
None
Congestion avoidance
time-out or 3 dupACKs
ssthresh = cwnd / 2
cwnd = 1
an ACK arrived
cwnd = cwnd + 1
an ACK arrived
cwnd = cwnd + (1 / cwnd)


Events
3dupACKs: three duplicate ACKs arrived
Time-out: time-out occurred
Th: ssthresh ≥ cwnd
cwnd
(in MSS)
16
14
12
10
08
06
04
02
00
ssthresh (16)
Time-out
Th
ssthresh (4)
3dupACKs
Th
ssthresh (6)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
RTTs
SS
SS
CA
SS
CA
Legend
ACK(s) arrival
Begin of slow start
Congestion detection
Threshold level
SS: Slow start
CA: Congestion avoidance

TAHO TCP: EXAMPLE



TAHO TCP: EXAMPLE



- TCP starts data transfer and sets the ssthresh variable to an ambitious value of 16 MSS. TCP begins at the slow-start (SS) state with the cwnd = 1. The congestion window grows exponentially, but a time-out occurs after the third RTT (before reaching the threshold). TCP assumes that there is congestion in the network. It immediately sets the new ssthresh = 4 MSS (half of the current cwnd, which is 8) and begins a new slow start (SA) state with cwnd = 1 MSS. The congestion grows exponentially until it reaches the newly set threshold. TCP now moves to the congestion avoidance (CA) state and the congestion window grows additively until it reaches cwnd = 12 MSS. At this moment, three duplicate ACKs arrive, another indication of the congestion in the network. TCP again halves the value of ssthresh to 6 MSS and begins a new slow-start (SS) state. The exponential growth of the cwnd continues. After RTT 15, the size of cwnd is 4 MSS. After sending four segments and receiving only two ACKs, the size of the window reaches the ssthresh (6) and the TCP moves to the congestion avoidance state. The data transfer now continues in the congestion avoidance (CA) state until the connection is terminated after RTT 20.

Reno TCP

1. Introduction of Fast-Recovery State:

- In the newer version of TCP, known as Reno TCP, a fast-recovery state was added to the congestion control Finite State Machine (FSM).

2. Handling Congestion Signals:

- Reno TCP treated congestion signals differently: time-out and arrival of three duplicate ACKs.
- If a time-out occurred, TCP moved to the slow-start state (or started a new round if already in slow start).
- If three duplicate ACKs arrived, TCP moved to the fast-recovery state and stayed there as long as more duplicate ACKs were received.

3. Fast-Recovery State Behavior:

- The fast-recovery state was positioned between slow start and congestion avoidance.
- It behaved like slow start, with the congestion window (cwnd) growing exponentially, but it started with the value of ssthresh plus 3 Maximum Segment Sizes (MSS) instead of 1.

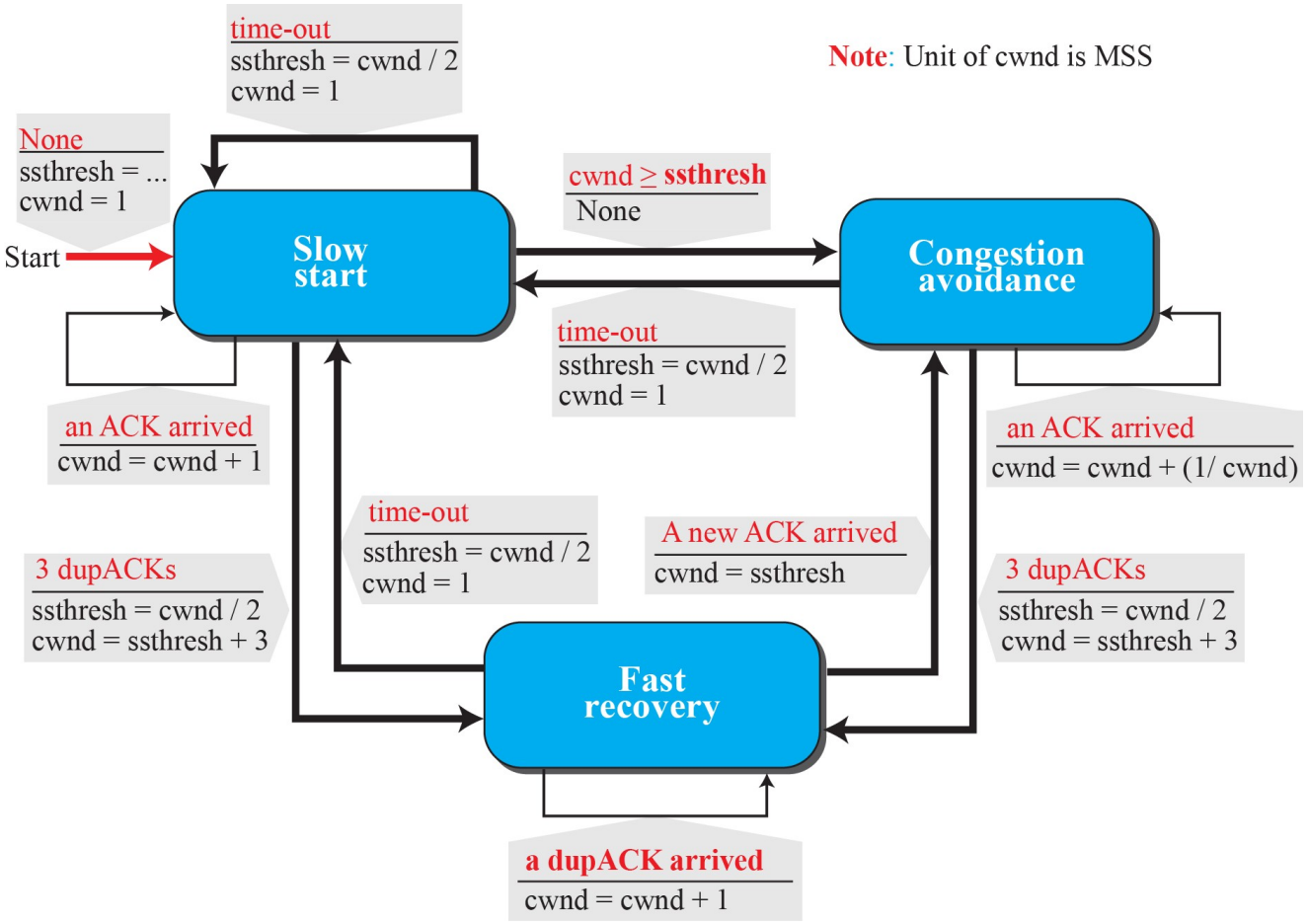
Reno TCP

4. Events in Fast-Recovery State:

In the fast-recovery state, several events could occur.

- If duplicate ACKs continued to arrive, TCP stayed in this state, and cwnd grew exponentially.
- If a time-out occurred, TCP assumed real congestion and moved to the slow-start state.
- If a new (non-duplicate) ACK arrived, TCP moved to the congestion-avoidance state but reduced the cwnd size to the ssthresh value, as if the three duplicate ACKs had not occurred.

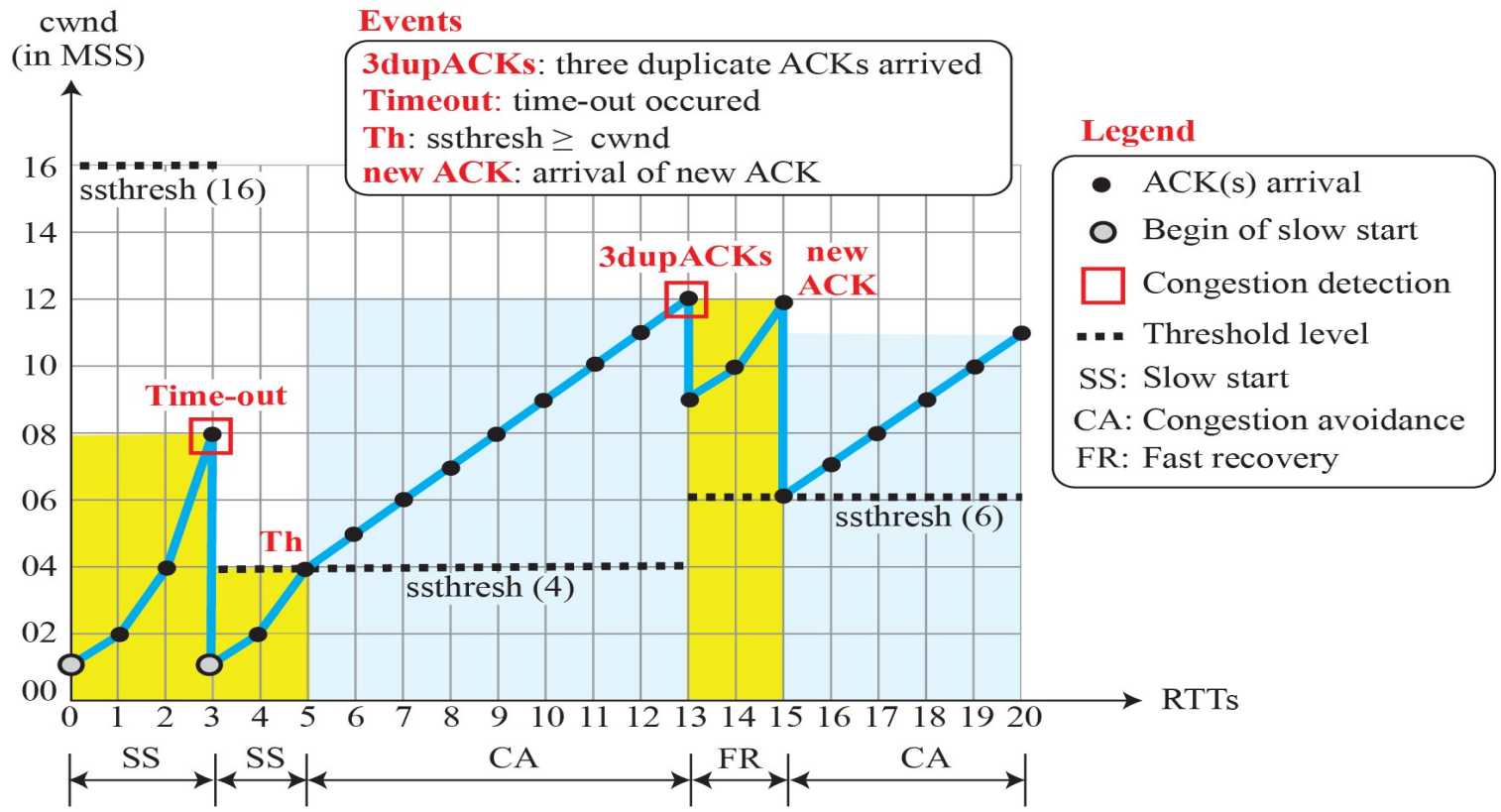
Reno TCP: FSM


time-out
ssthresh = cwnd / 2
cwnd = 1
Note: Unit of cwnd is MSS
None
ssthresh = ...
cwnd = 1
Start
Slow start
cwnd ≥ ssthresh
None
Congestion avoidance
time-out
ssthresh = cwnd / 2
cwnd = 1
an ACK arrived
cwnd = cwnd + 1
an ACK arrived
cwnd = cwnd + (1/ cwnd)
time-out
ssthresh = cwnd / 2
cwnd = 1
A new ACK arrived
cwnd = ssthresh
3 dupACKs
ssthresh = cwnd / 2
cwnd = ssthresh + 3
3 dupACKs
ssthresh = cwnd / 2
cwnd = ssthresh + 3
Fast recovery
a dupACK arrived
cwnd = cwnd + 1

Reno TCP: Example

Figure(on next page) shows the same situation as Figure(taho tcp), but in Reno TCP. The changes in the congestion window are the same until RTT 13 when three duplicate ACKs arrive. At this moment, Reno TCP drops the ssthresh to 6 MSS, but it sets the cwnd to a much higher value (ssthresh + 3 = 9 MSS) instead of 1 MSS. It now moves to the fast recovery state. We assume that two more duplicate ACKs arrive until RTT 15, where cwndgrows exponentially. In this moment, a new ACK (not duplicate) arrives that announces the receipt of the lost segment. It now moves to the congestion avoidance state, but first deflates the congestion window to 6 MSS as though ignoring the whole fast-recovery state and moving back to the previous track.

Reno TCP: Example



NewReno TCP

- A later version of TCP, called NewReno TCP, made an extra optimization on the Reno TCP.
- In this version, TCP checks to see if more than one segment is lost in the current window when three duplicate ACKs arrive.
- When TCP receives three duplicate ACKs, it retransmits the lost segment until a new ACK (not duplicate) arrives.
- If the new ACK defines the end of the window when the congestion was detected, TCP is certain that only one segment was lost. However, if the ACK number defines a position between the retransmitted segment and the end of the window, it is possible that the segment defined by the ACK is also lost.
- NewReno TCP retransmits this segment to avoid receiving more and more duplicate ACKs for it.

Additive Increase, Multiplicative Decrease

- Out of the three versions of TCP, the Reno version is most common today. It has been observed that, in this version, most of the time the congestion is detected and taken care of by observing the three duplicate ACKs.
- Even if there are some time-out events, TCP recovers from them by aggressive exponential growth.
- In other words, in a long TCP connection, if we ignore the slow-start states and short exponential growth during fast recovery, the TCP congestion window is **cwnd = cwnd + (1/cwnd)** when an ACK arrives (congestion avoidance), and **cwnd = cwnd / 2** when congestion is detected, as though SS does not exists and the length of FR is reduced to zero.
- The first is called additive increase; the second is called multiplicative decrease. This means that the congestion window size, after it passes the initial slow start state, follows a saw tooth pattern called **additive increase, multiplicative decrease (AIMD)**

Additive Increase, Multiplicative Decrease