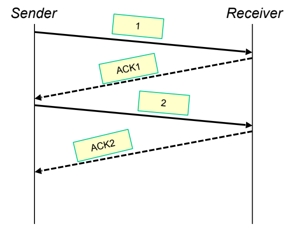
TCP Congestion Control Tutorial

- Path Capacity and Windows
- Bandwidth-Delay Product (BDP)
- Slow-Start, Congestion Avoidance
- TCP flavors: Tahoe, Reno, NewReno
- Fast Retransmit/Fast Recovery
- Throughput formula
- The TCP SACK option and Scoreboard

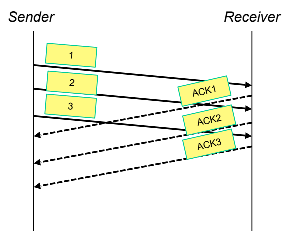
Window-based Mechanism: 1

- A simple protocol for reliable communication is *stop-and-wait*
  - *Stop-and-wait send a data segment in a packet and then wait for an acknowledgment (ACK) packet.*
  - *If the ACK does not arrive before the retransmission timeout (RTO), the data segment is retransmitted.*
- *Stop-and-wait*, delays sending a new segment until an ACK is received.
- This implies a window of 1 segment in flight.
- This wastes capacity

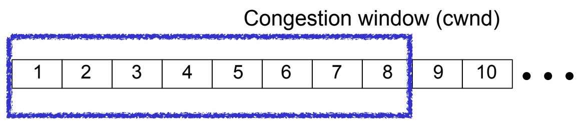


Window-based Mechanism: TCP

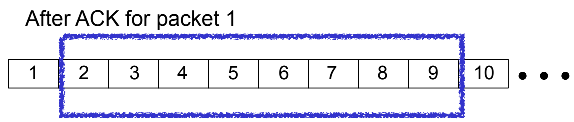
- All TCP data segments are numbered.
- A sliding window allows only a set of “in flight” segments.
- The ***congestion window*** (cwnd) is the key method to control transmission.
- A cwnd >1 allows multiple unacknowledged segments (3 in the example on the right).



An example with cwnd=8



- In this case, up to cwnd (8) segments can be sent
  - after 8, the sender has to pause and wait to receive an ACK.
- When the ACK for segment 1 is received,
  - the cwnd slides right, this allows the transmission of segment 9



Culmulative ACKs

- Sending an ACK for every segment results in many packets!

cwnd

$\boxed{1 \; 2 \; 3 \; 4 \; 5 \; 6 \; 7 \; 8}$ 9 10 11 12 13 • • •

- TCP acknowledgments are therefor cumulative:
  - An ACK for segment 3, also acknowledges receipt of segments 1 and 2
  - I.e. cwnd slides 3 positions, allowing three new segments to be transmitted

When the ACK for 3 is received:

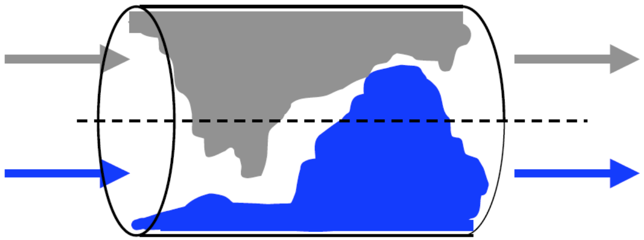
1 2 3 $\boxed{4 \; 5 \; 6 \; 7 \; 8 \; 9 \; 10 \; 11}$ 12 13 • • •

- TCP is *actually* byte-oriented
  - ACKs report byte sequence numbers
  - TCP can acknowledge a part of a segments transmitted
  - To keep things simple we will continue to discuss in terms of segments

Congestion

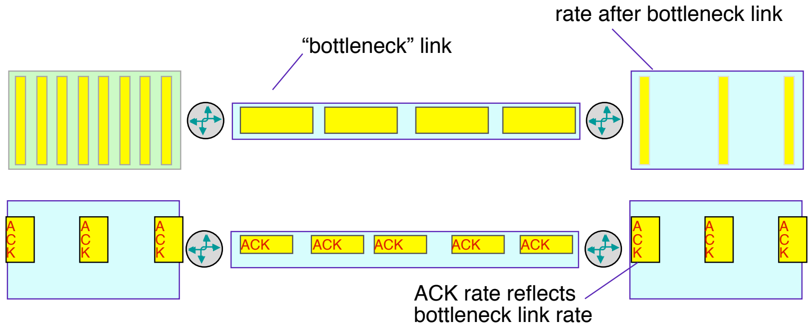
- A network (path) has limits to how fast it can send, i.e. the capacity
- Trying to send more results in overload, i.e. congestion
- Congestion needs to be avoided:
  - In road networks cars are conserved, never lost or duplicated.
    - Road speeds adapt to the load, roads choke-up
  - In the Internet, egress links operate at a sending rate
    - Packets arriving on a router link need to be buffered until they can sent
    - Buffers can fill, and then packets can be dropped at a router.
    - The path sender is responsible for detecting loss and resending lost packets.

Sharing the Link Capacity on a Path



- A single flow is typically unable to use all the link capacity
- Flows sending packets at different times combine to use capacity
  - There is no congestion when packets are forwarded *promptly*
  - *(i.e. buffered only for a small fraction of the path RTT)*

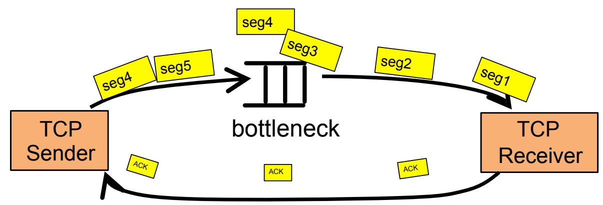
Bottlenecks and ACKs



Each ACK received indicates a packet has "left the path"
The ACK arrival rate indicates capacity of the "bottleneck" link
N.B. True even when ACKs return on a different path!

UNIVERSITY OF ABERDEEN

Overflow of the bottleneck buffer

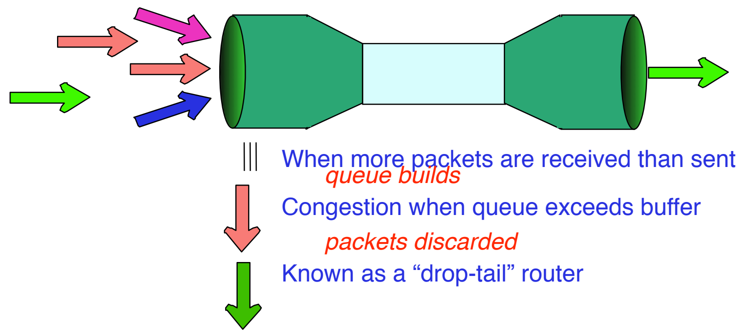


- A TCP Sender sends segments (packets) on a path
- When a router reaches capacity of the bottleneck interface, every extra received segment is ***buffered before being forwarded.***
  - If router arrival rate continues to > egress rate, the queue in the buffer grows
  - If the buffer becomes full, packets are discarded (lost).
- Congestion results when packets are buffered for *long* times
- Lost packets are not acknowledged by the TCP Receiver
  - At the TCP Sender, any loss is detected by tracking received ACKs
  - The TCP Sender needs ti reduce its rate to avoid more congestion and loss

UNIVERSITY OF ABERDEEN

Avoiding Congestion Collapse

- In the 1980's Internet **Congestion Collapse** was a real problem



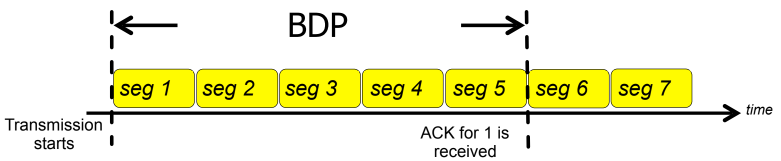
When more packets are received than sent
queue builds
Congestion when queue exceeds buffer
packets discarded
Known as a "drop-tail" router

- Three insights:
  1. Senders need to *avoid adding to congested paths* (congestion control)
  2. Senders need to *significantly reduce under severe load* (backoff)
  3. *Network control traffic* needs to be prioritised and not dropped

UNIVERSITY OF ABERDEEN

Congestion Control

- *Insight: to control it sending rate, a sender needs to creating congestion*
- The key problem: **How large fast should the sender transmit?**
  - Ideally, the number of outstanding segments should equal the path *bandwidth delay product (BDP)*
- *However, the available capacity of an Internet path cannot be known!*
- Senders there need to implement *congestion control*!

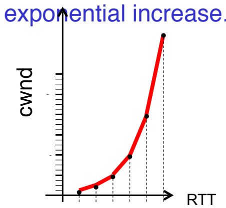


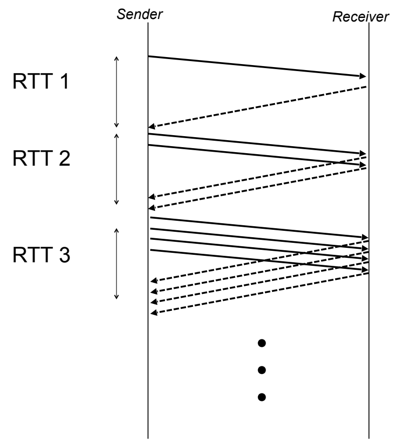
UNIVERSITY OF ABERDEEN

Slow Start

A TCP Sender uses congestion control
This controls use of capacity by dynamically increasing or decreasing the a window: the congestion window, *cwnd,* according to detected congestion.

- TCP increases cwnd :
  - TCP starts from an initial cwnd
  - It determines the available capacity
- For every ACK received, the cwnd is increased by one segment
  - This ***doubles*** the cwnd each RTT!
  - This results in exponential increase.





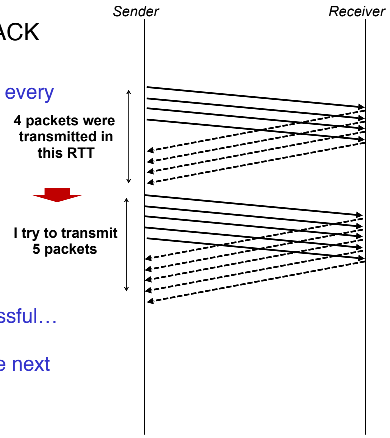
Slow Start and CA phases

- We introduce the slow start threshold variable, ssthresh
  - ssthresh is initialised to a large value at the start of a connection
- Congestion control uses 2 phases:
  - **Slow Start**: cwnd < ssthresh, the sender exponentially increases cwnd.
  - Once congestion is detected:
    - cwnd/2* is saved in ssthresh (a sender knows last RTT cwnd <= capacity)
    - ssthresh is uses as an estimate of the capacity for the next slow start increase
  - **Congestion Avoidance**: cwnd>= ssthresh
    - The sender slowly increases cwnd to use any extra capacity
  - Together this is known as Additive-Increase Multiplicative Decrease

* Complication: Senders should use Flight_Size, rather than cwnd, to adjust SSthresh after a loss

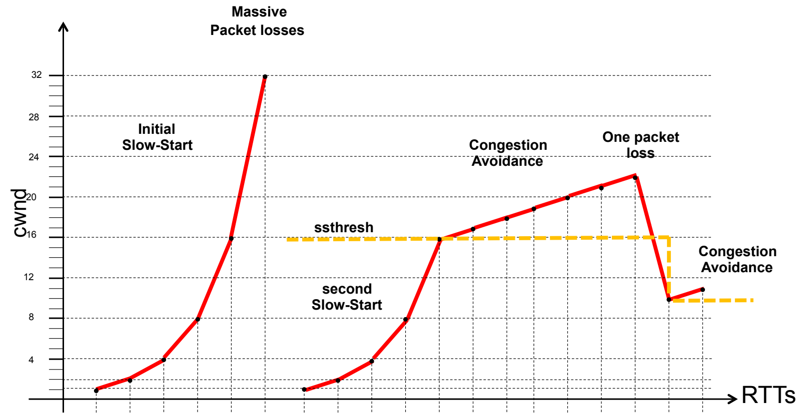
Congestion Avoidance (CA)

- When the *cwnd > ssthresh*
- cwnd increased by one every time an ACK for a cwnd of segment is received
  - Linearly increase of cwnd by one segment every RTT



- Interpretation:
  - **If** transmission of **N segments** were successful…
  - **Then** try to transmit **N+1 segments** to see whether the path has capacity to send more next RTT.

Additive-Increase Multiplicative Decrease (AIMD)



Recovery of Lost Segments

- Segments are sent as long as there is space in the cwnd
- As previous ACKs received, the sequence of sent segments to slide right
- ... until a lost segment becomes the first in the window.

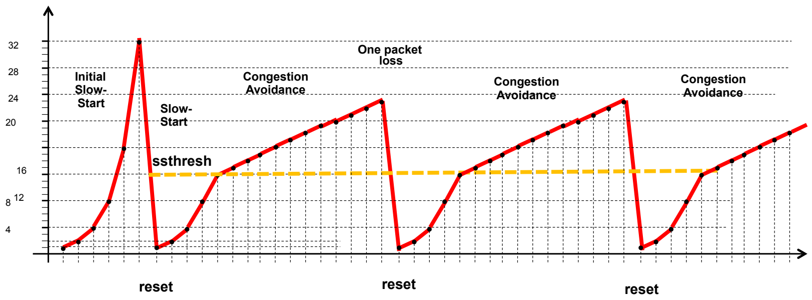
				?	?	?							
21	22	23	~~24~~	25	26	27	28	29	30	31	32	33	• • •

- E.g., If segment 24 the last ACK will be for segment 23
- A sender does not know what happened to any later segments

UNIVERSITY OF ABERDEEN

TCP Tahoe (1988)

- The sender records the cwnd before loss in the Slow Start Threshold (ssthresh)
- It resets the cwnd to one and starts growing cwnd
- All unacknowledged segments are resent (it does not know better)
- When cwnd=ssthresh, it starts linear growth (until congestion is detected)



The congestion control adapts the rate (aka adapts ssthresh) each time it detects congestion.

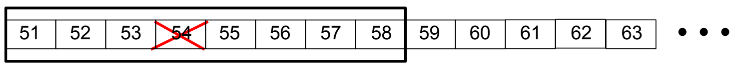
UNIVERSITY OF ABERDEEN

Fast Retransmit Recovery

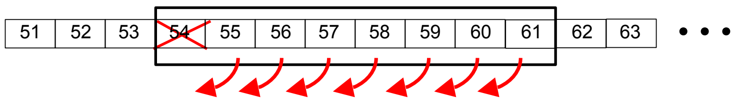
- In TCP Tahoe, Loss recovery relies on a timer to detec los
  - This is inefficient
  - TCP has to be idle until the retransmission timer expires
  - TCP has to retransmit any correctly received *as well as* lost segments
- A better solution uses ACKs to “self-clock” the sender:
  - When **the receiver** receives any out of sequence segment, it ACKs the last received **in-order** segment
  - When **the sender** sees 3 duplicated ACKs (dupack), the next in-order segment to be ACK’ed is declared lost and Fast Retransmitted
  - TCP does not need to await for a (long) RTO!!

UNIVERSITY OF ABERDEEN

Example of Fast Retransmit



Packet 54 is lost … but not the others in the window



The receiver sends DUPACKs for 53 for each of the later packets

Three DUPACKs trigger Fast Retransmit

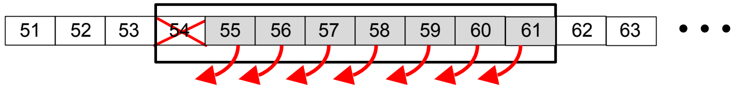
TCP retransmits 54 before the RTO expiration

UNIVERSITY OF ABERDEEN

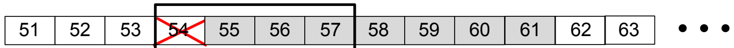
Fast Recovery (TCP Reno)

- What do DUPACKs mean for a TCP sender?
  - Three DUPACKs indicate loss, which is a congestion signal: **cwnd reduced**
  - DUPACKs also indicate a segment has left the network **another segment can be sent** (a *principle of packet conservation*)
- The sender can “inflate” cwnd, allowing it to send a more segments during Fast Recovery
- When the sender discovers correct reception of all segments that were outstanding at the time when the loss, the sender resumes transmission using normal congestion control rules (using a corrected cwnd)

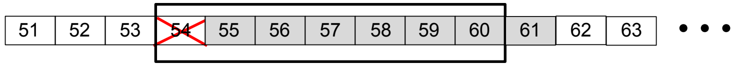
Example of Fast Recovery



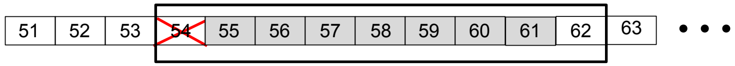
After retransmitting 54, sender shrinks cwnd by half…



3 DUPACKs are received, indicating loss and I can immediately increase cwnd by 3…

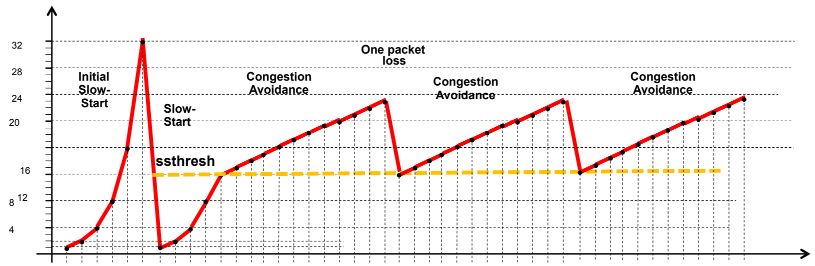


When other DUPACKs are received, sender inflates cwnd to transmit new segments



TCP Reno: using cwnd

- TCP Reno avoids slow-start at each transmission cycle



Number of Segments Sent

- The previous diagram allows for a simple TCP throughput calculation
- In "n" RTTs number of segments sent is:
  - *SegSent = n + (n + 1) + … + 2n = 3/2 * n (n+1) ~ 3/2 * n²*
- Assuming one segment is lost and retransmitted, the fraction of segments lost is:
  - *p = 1/SegSent = (2/3) * 1/(n (n+1)) ~ 2/(3n²)*
  - ***n = 2/(3p)=√(2/(3p))***

Inverted-square-root Formula

- Throughput in packet/second is:

$$Tput\ (pps) = \frac{3/2\ n(n+1)}{n\ RTT} \sim \frac{\sqrt{3/2}}{\sqrt{p}\ RTT}$$

- Throughput is inversely proportional to:
  - The RTT
  - The square root of loss-ratio
- The segment size can be used to calculate throughput in b/s

UNIVERSITY OF ABERDEEN

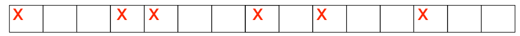
TCP New-Reno (RFC 2582)

- TCP Reno was widely implemented, but it two problems:
  - Too many lost segments in the same window of data substantially increase the duration of the recovery phase
    - With N losses N RTTs are needed to recover all losses
  - Multiple cwnd reductions can occur for the same loss episode
- TCP New-Reno partially fixed this problem in two ways:
  - Introduced a timeout on the Fast Retransmit/ Fast Recovery
  - No recovery unless all packet loss was counted.

UNIVERSITY OF ABERDEEN

Further Improvements: Selective ACKs

- TCP New-Reno suffers from the performance limitation of TCP Reno when multiple segments are lost in the same window of data
- What do we do when there is a complex pattern of loss?



Multiple losses detected in the same cwnd of segments

- The TCP SACK option allows a receiver to specify the set of segments that have been successfully received
- A sender can then retransmit only segments that have not been acknowledged

UNIVERSITY OF ABERDEEN

Improvements to AIMD

- CC methods to increase cwnd will overshoot the bottleneck capacity.
- Fills buffers before the smallest capacity (bottleneck) link on the path
- Continued overshoot will eventually result in loss
  - Triggering recovery and reduction of cwnd (a new ssthresh).
- Can we detect buffering *before loss*, and reduce this overshoot???

Yes, newer methods can:

1. Measure increases in end-to-end delay and react to this!
2. Ask routers to explicitly mark packets to tell receivers they have started to buffer packets.
3. Ensure routers don't buffer huge numbers of packets – using active queue management to drop or mark early.

UNIVERSITY OF ABERDEEN

Conclusion

- **Senders start with a conservative rate, and increase their rate**
  - A sending rate > Path capacity results in delay and loss!
- **A sender *detects* loss and *retransmits the lost segments***
  - Tahoe was a simple method using a timer, but no longer used
  - Instead, Reno retransmits lost packets based on ACKs:
    - FR/FR uses DupACKs to improve efficiency
    - The TCP SACK Option helps further
- **A sender *detects* congestion and *adjusts its rate after congestion***
  - Congestion Control uses a two-phase AIMD control function:
    - *Slow-Start* and *Congestion Avoidance*
- There are further improvements (Cubic, BBR, PRR, TLP, etc)
- ***Key take away: an adaptive control loop enables efficient transmission***

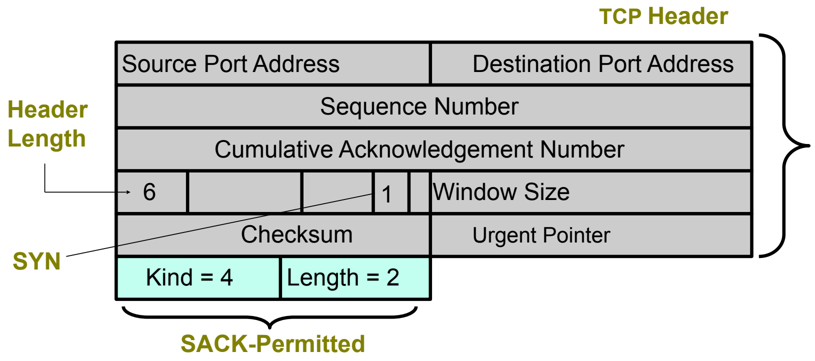
UNIVERSITY OF ABERDEEN

Spare Slides

UNIVERSITY OF ABERDEEN

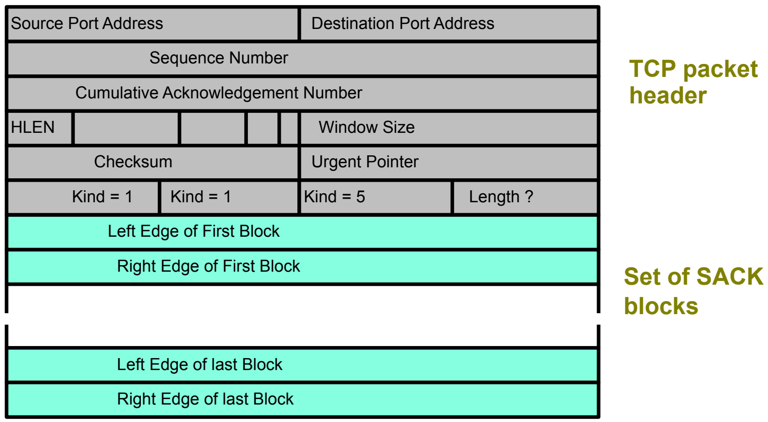
SACK Details

- Implemented using two TCP options:
  - The SACK-Permitted option sent at the start of a connection
  - The SACK Blocks sent with ACKs when SACK is permitted
- The SACK-Permitted TCP option (2 bytes) is sent in a TCP SYN
- This indicates SACK can be used once a connection established



UNIVERSITY OF ABERDEEN

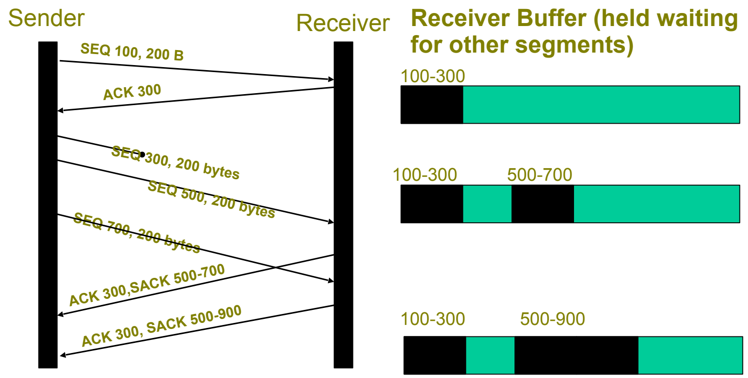
SACK Blocks



UNIVERSITY OF ABERDEEN

SACK Blocks

The sender uses SACK options to indicate ***contiguous*** and ***isolated blocks*** of segments that were successfully received



UNIVERSITY OF ABERDEEN

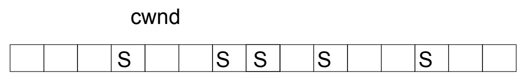
SACK Rules

- A SACK Block does not change the meaning of the ACK field
- A SACK Block cannot be sent unless the SACK permitted option was received
- If SACKs are sent, they should be included in all packets when out-of-order data has been buffered at the receiver
- First segment in a SACK must acknowledge the most recently received out-of-order segment

UNIVERSITY OF ABERDEEN

Sender/Receiver Algorithms: RFC 3517

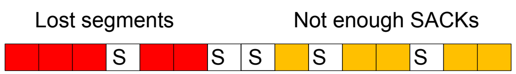
- RFC 3517 specifies how to use SACK Blocks to improve Fast Retransmit/Fast Recovery
- SACK sender use a **scoreboard**
  - The scoreboard keeps note whether each outstanding segment was received or not.
  - The scoreboard is updated every time a SACK Block is received



UNIVERSITY OF ABERDEEN

SACK Scoreboard

- A segment in the **scoreboard** is considered lost if at least three SACKs do not acknowledge it.



- If a segment in the scoreboard is not marked as lost, it is considered still *in flight* and it is *not* retransmitted yet.
  - In this example the 5 orange segments are considered still in flight
- TCP follows the same rules as New-Reno to update cwnd, but segments in flight are indicated by the scoreboard