yz709-networking-sup4

Tripos question

y2016p5q6

(*b*) The formulae below are used in TCP implementations to compute a value for the retransmission time-out T. R is an estimate of the round-trip time (RTT), D is an estimate of variance, M is the most recently measured round-trip measurement, $\alpha = 0.875$ and $h = 0.25$.

$$\begin{aligned} D &\leftarrow D + h(|M - R| - D) \\ R &\leftarrow \alpha R + (1 - \alpha)M \\ T &= R + 4D \end{aligned}$$

(*i*) Give an example of how the retransmission time-out T is used within TCP. [1 mark]

- Used for congestion control in TCP, the retransmission time-out T could be a sign of congestion and trigger the sender to reduce the rate at which it is transmitting.

(*ii*) Describe why the computation of the retransmission time-out T incorporates a correction for deviation in the estimate of the RTT. [2 marks]

- Aims at capturing the variation in RTT, when the variation is small, timeout T is close to the estimate of the RTT, but when the variation is large, the variance causes the derivation to dominate the timeout T .

💡 Comments:

y2015p5q4

The following equation provides a simple way to estimate the throughput of a TCP connection, as a function of the loss probability p, the round-trip time RTT, and the maximum segment size MSS.

$$\text{TCP Throughput} = \sqrt{\frac{3}{2}}\frac{\text{MSS}}{\text{RTT}\sqrt{p}}$$

(a) Alice wants to send a large amount of data to Bob over a network path with RTT $= 100$ ms, $p = 0.01$, and MSS $= 10,000$ bits. What is the expected throughput in Mbit/s? [2 marks]

(a)

$$\text{TCP throughput} = \sqrt{\frac{3}{2}}\frac{\text{MSS}}{\text{RTT}\sqrt{p}}$$

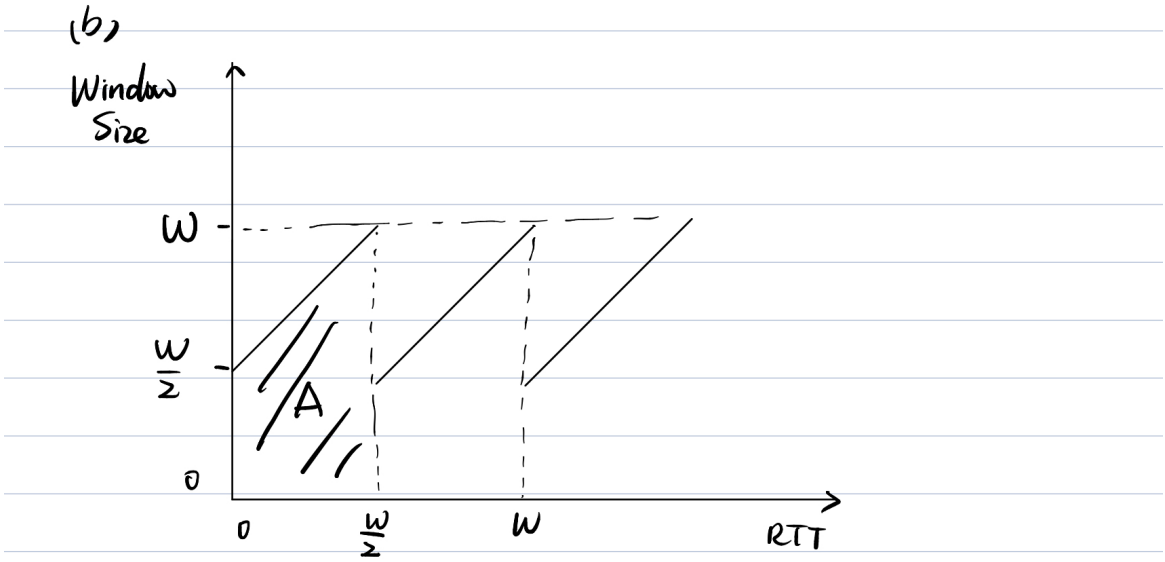
$$= \sqrt{\frac{3}{2}}\frac{10\,000 \text{ bits}}{100 \text{ ms} \times \sqrt{0.01}}$$

$$= \sqrt{\frac{3}{2}} \times 10^6 \text{ bit/s}$$

$$= \sqrt{\frac{3}{2}} \text{ 1M bit/s}$$

(b) With the aid of a clearly labelled diagram showing window-size versus time, derive the above equation. [10 marks]

- Hint: draw a plot of window size versus time and qualitatively explain the intuition between the variables in the question and your plot
- (1) assume packet drop rate/probability p, a sender will be able to send an average of $\frac{1}{p}$ packets before a packet loss, which is the area under each cycle $A = \frac{3}{8}W_{max}^2$; (2) The number of bytes $MSS \times A$ (maximum segment size $\times$ number of packets); (3) The time taken is $\frac{1}{2}W_{max} \times RTT$, hence $\text{TCP throughput } = \frac{\text{MSS}\times A}{\frac{1}{2}W_{max}\times\text{RTT}} = \sqrt{\frac{3}{2}}\frac{MSS}{\text{RTT}\sqrt{p}}$



(c) Alice has two options to improve the throughput: halving either the RTT or the loss probability p. If both cost the same, which is more cost effective and why? [2 marks]

- Having RTT is more efficient because it gives us double the throughput
- Having loss probability p gives us $\sqrt{2}$ the throughput which is smaller than $2\times$throughput

(d) Consider your derivation of the equation in part (b). State three assumptions that are made and describe when these assumptions may not hold in reality. [6 marks]

- (1) assume a sender sends an average of $\frac{1}{p}$ packets before a packet loss, but it is an assumption that the sender would fully utilise the link, in reality, the sender will be idle some times and will not continuously send packets without stop.
- (2) for a high-speed TCP, the amount of data that can be sent between drops is not a practical number
- (3) assume the time taken is $\frac{1}{2}W_{max} \times RTT$, but in reality, there might be delays because of the network, so the time taken might be longer. We have not taken account of the time for sending and receiving acknowledgements.
- (4) In reality, we use a sliding window strategy, so the sender cannot continuously send packets if the sender window is full

💡 Comments:

UDP

a. What is the role of the port number in UDP?

- A port number is used for demultiplexing. It is a unique identifier for a process on a single host, the sender sends messages to the port, receivers receive messages from the port.
- In the UDP header, the source and destination port number is embedded, the host can use IP addresses and port numbers to direct the message to the appropriate socket (i.e., provide a demultiplexing service).

b. Give three reasons an application may be designed to use UDP instead of TCP.

- The UDP has no need to establish a connection prior to data transfer, provides unreliable delivery service. So it has less overhead and latency. Low latency and resource consumption are crucial for real-time applications such as computer gaming, voice conferencing and voice over IP (VoIP).
- Used for simple request-response communication when the size of data is less and there is less concern about flow and error control. The checksum is not mandatory in UDP, hence it saves more bandwidth.
- Applications require high performance and can tolerate some packet drops.

💡 Comments:

Error control and ARQ

a. Why do packet switched protocols tend to use error detection instead of error correction?

- Packet retransmission is easy and cheap compared to the cost of correcting errors.
- There is less noise on the link and has a low probability of getting errors.
- Carrying fewer check bits save bandwidth, so we can transfer more useful data in a packet.

💡 Comments:

b. A transport protocol for packet-switched networks uses a “sliding window” Automatic Repeat reQuest (ARQ) scheme for error control and flow control.

i. As well as error detecting codes, ARQ protocols use acknowledgments and timeouts to achieve error control. Briefly explain what these are, and how they are combined to achieve reliable transmission.

- Acknowledgements are used by the receiver to respond to the sender once it has received a packet without bit errors, so that sender knows the packet has been successfully transmitted.
- Timeouts are used for loss detection, a timer is set at the beginning of the packet transmission, and if the sender has not received an acknowledgement from the receiver within the pre-set period, then the sender can assume the packet is lost during transmission and trigger a re-send.
- For a reliable transmission, the sender first sets the timer, and sends the packet to the receiver, once the receiver receives the packet without bit errors, it sends an acknowledgement to the

sender, once the sender gets the acknowledgement from the receiver, it cancels the timeout and progress to the next packet. If the receiver does not receive any packets or it receives a packet with bit errors, then it will drop the packet and will not send an acknowledgement (depends on the implementation, probably send NACK) to the sender, so once timeout, the sender will know the packet has not been delivered successfully and trigger a re-send.

ii. What two error cases might cause a receiver to send a negative acknowledgment (NACK)?

- If the packet has bit errors when checked using `checksum`.
- If an out-of-order packet has been received, the receiver might trigger a NACK for the expected packet to the sender.

iii. What happens if the NACK is lost?

- The sender will resend the packet once timeout. So before timeout, the sender will continuously send the following packets to the receiver but the receiver will drop all of the out-of-order packets, hence the throughput of the network is low and a lot of bandwidth is wasted.

> 💡 Comments:

c. Given a transport protocol that provides bidirectional communication, what optimisation can be made in the implementation of acknowledgments to reduce the total number of packets sent?

- We could implement a Selective ACK strategy.
- Firstly, we only send acknowledgements, do not send NACK. If we received a packet with bit errors, we will just drop it and send an ACK with the last sequence number successfully received.
- Secondly, the sender transmits up to n unacknowledged packets, and the receiver only indicates the lost packet, upon receiving, the sender only retransmits the lost packet k
- Hence no packet that has been successfully received will be sent more than once, but this strategy requires more complicated book-keeping (e.g., a timer per packet) although more efficient as it offers more precise information.

> 💡 Comments:

d. If two hosts are connected by a 100 Mbps link with a roundtrip time of 20 msec, how big (in bytes) should the sliding window be to maximise link usage?

- The throughput $\approx \frac{n}{RTT}$ where n is the size of the window.
- To maximise link usage, if we fully utilise the link, we will get throughput ≤ 100 Mbps, hence $n \leq 100$ Mbps $\times 20$ msec = 2×10^6 bits = 2 megabits = 250 kilobytes (Kb)

> 💡 Comments:

e. Consider a sliding window protocol with a window size of 5 using cumulative ACKs.

i. Retransmissions: Retransmissions occur under two conditions

   1. Reception of three duplicate ACKs (NOTE: a duplicate ACK is received after the initial ACK for a packet, so you need three identical ACKs *after* the initial ACK)
   2. Time out after 100 msec (timer starts at the beginning of the packet transmission)

ii. Timing:

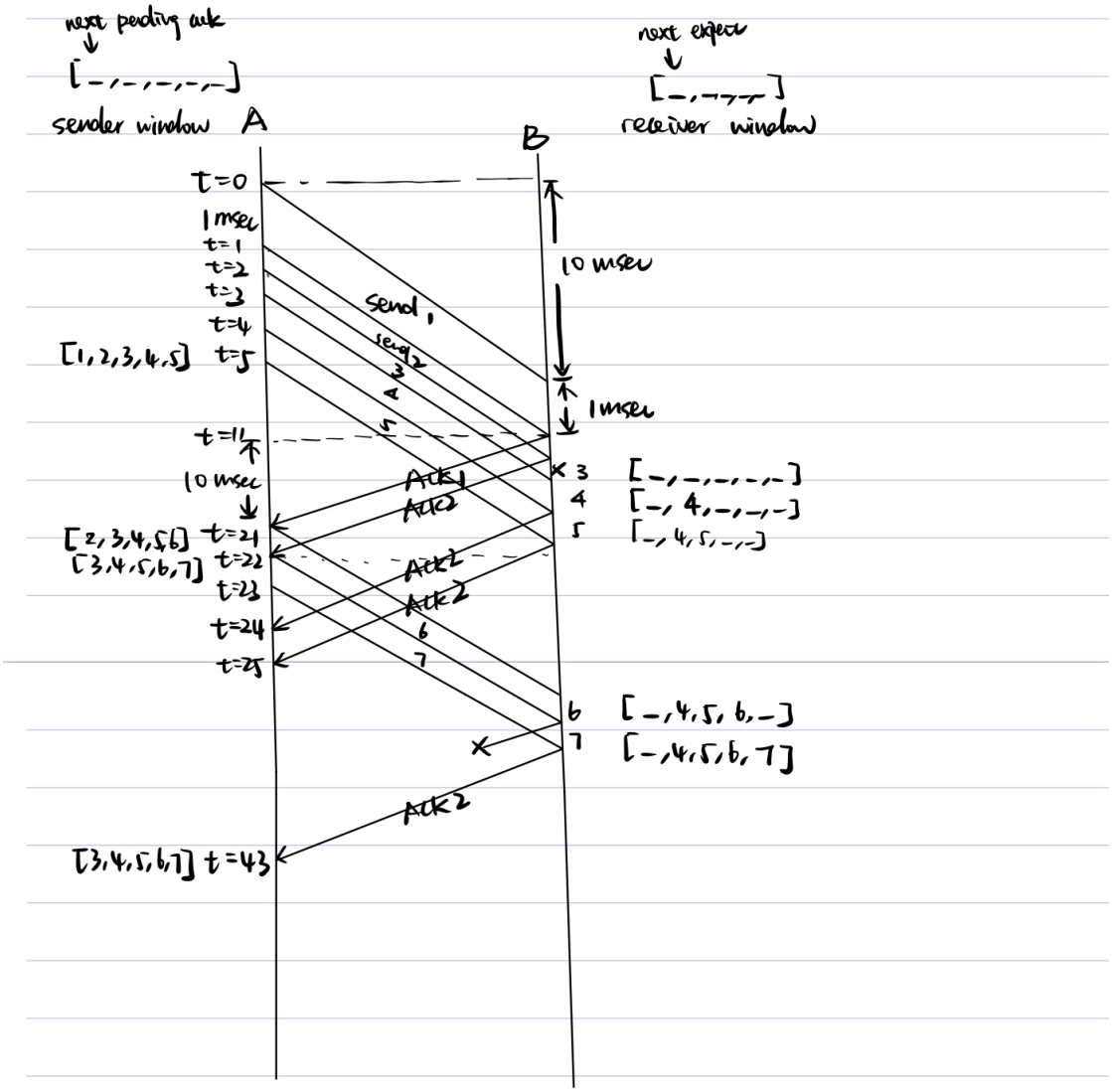
   1. Data packets have a transmission time of 1 msec
   2. ACK packets have zero transmission time
   3. The link has a latency of 10 msec.
   4. The source A starts off by sending its first packet at time t=0.

iii. Assume all packets are successfully delivered except the following:

   1. The first transmission of data packet #3
   2. The ACK sent in response to the receipt of data packet #6

When is data packet #3 first retransmitted (expressed in terms of msec after t=0)?

- The packet #3 will be retransmitted at time $t = 43$ when the sender receives three ACK 2 from the receiver expecting packet #3



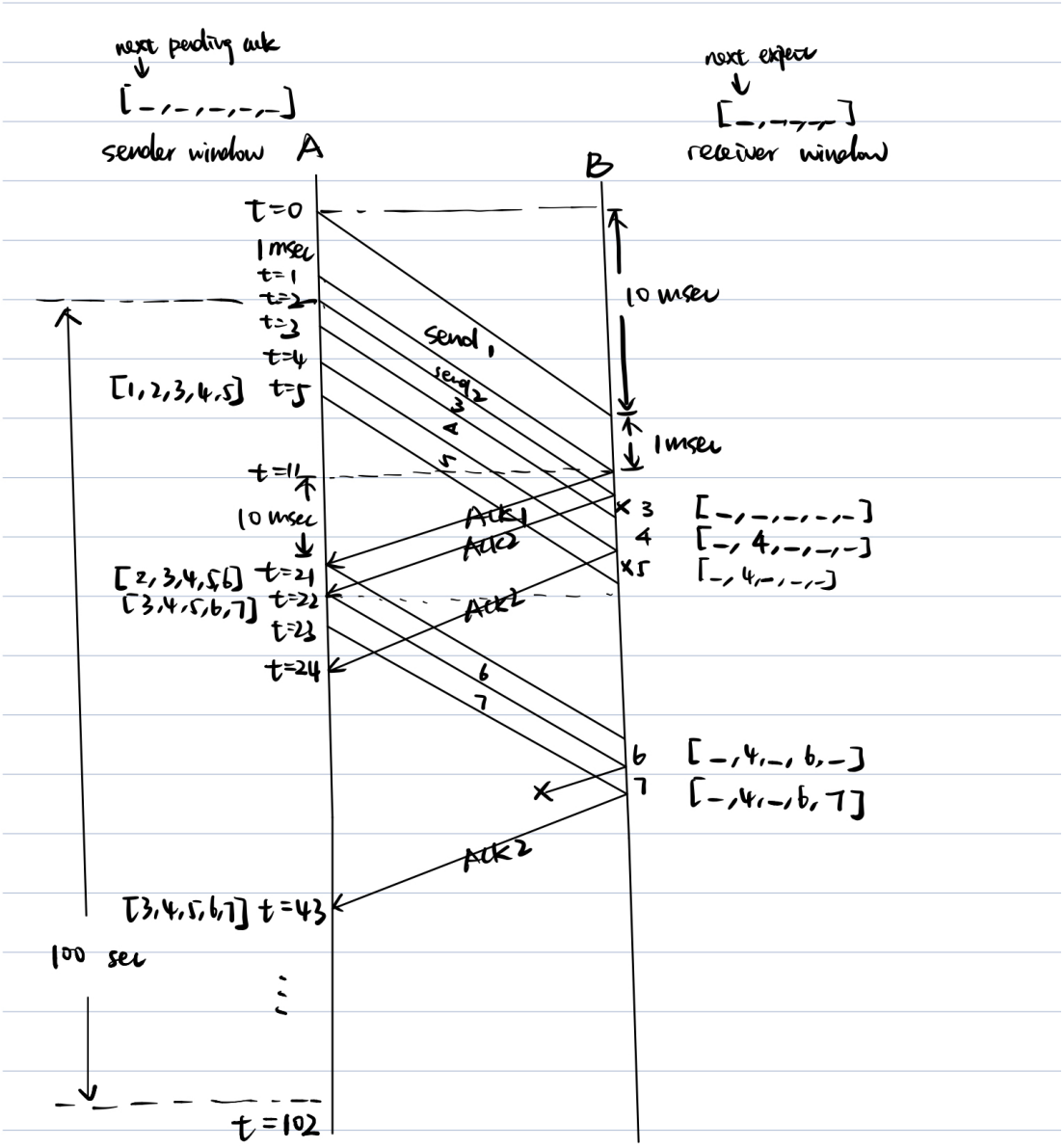
Comments:

iv. Consider the same scenario, but with everything successfully delivered except the following:

1. The first transmission of data packet #3
2. The first transmission of data packet #5
3. The ACK sent in response to the receipt of data packet #6

When is data packet #3 first retransmitted (expressed in terms of msec after t=0)?

- Packet #3 will be retransmitted after a timeout of 100 msec, at $t = 102$ msec because the sender only received two ACK 2, not triggering fast retransmission.



Comments:

v. Assume we can only observe the ACK packets arriving at the sender. The same sliding window algorithm is used, with the same timings and retransmission policies apply.

   Notation (read carefully): The notation A{N} is used to mean that the ACK packet is acknowledging the receipt of all packets up to and including data packet N. That is, A5 is acknowledging the receipt of packet 5; to be clear, the notation does not mean that the receiver is expecting packet 5 as the next data packet.

   1. Assume that the following ACK packets arrive (just the ordering is shown, no timing information is provided): A1, A2, A3, A3, A4, A5, A6.

      Which of the following five scenarios would have produced such a series of ACKs? Assume everything worked correctly except the explicit failure condition mentioned,

and consider all that apply.

(a) Data packet number 4 was dropped.

(b) Data packet number 4 was delayed, arrived immediately after data packet 5

(c) Data packet 3 was duplicated by the network

(d) ACK packet A3 was duplicated by the network

(e) ACK packet A4 was delayed, arriving after A5

- Since we have 7 ACKs but only transferred 6 packets, so there is a duplicated packet or a duplicated ACK
- (c) is possible, A1(received packet 1), A2(received packet 2), A3(received packet 3), A3(received duplicated packet 3), A4(received packet 4), A5(received packet 5), A6(received packet 6).
- (d) is possible, A1(received packet 1), A2(received packet 2), A3(received packet 3), A3(a duplicated ACK 3 by the network), A4(received packet 4), A5(received packet 5), A6(received packet 6).

2. With the same set up as in the previous problem, consider the following stream of ACK packets: A1, A2, A3, A5, A4, A6

   Which of the following five scenarios would have produced such a series of ACKs? Assume everything worked correctly except the explicit failure condition mentioned, and consider all that apply.

   (a) Data packet number 4 was dropped.

   (b) Data packet number 4 was delayed, arrived immediately after data packet 5

   (c) Data packet 3 was duplicated by the network

   (d) ACK packet A3 was duplicated by the network

   (e) ACK packet A4 was delayed, arriving after A5

   - (e) is possible, A1(received packet 1), A2(received packet 2), A3(received packet 3), A5(received packet 5), A4(delayed ACK of receiving packet 4), A6(received packet 6).

3. With the same set up as in the previous problem, consider the following stream of ACK packets: A1, A2, A3, A3, A5, A6

   Which of the following five scenarios would have produced such a series of ACKs? Assume everything worked correctly except the explicit failure condition mentioned, and consider all that apply.

   (a) Data packet number 4 was dropped.

   (b) Data packet number 4 was delayed, arrived immediately after data packet 5

   (c) Data packet 3 was duplicated by the network

   (d) ACK packet A3 was duplicated by the network

   (e) ACK packet A4 was delayed, arriving after A5

   - (b) is possible, A1(received packet 1), A2(received packet 2), A3(received packet 3), A3(received packet 5 but not packet 4, so ACK 3), A5(received a delayed packet 4, so now have received up to packet 5, ACK 5), A6(received packet 6).

💡 Comments:

TCP Specifics

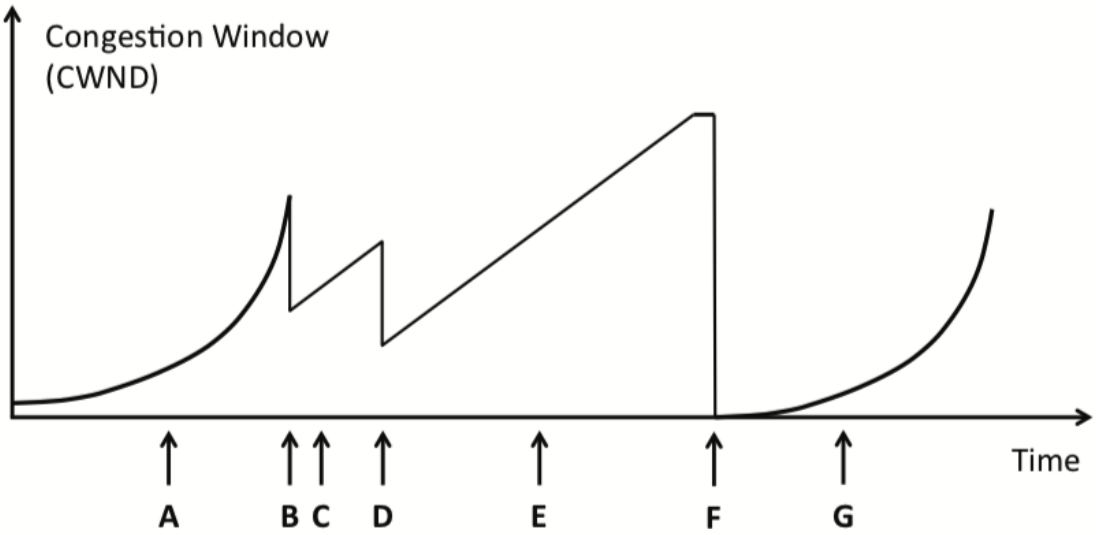
a. Consider the plot of CWND versus time for a TCP connection (below).

   i. At each of the marked points along the timeline in the figure, indicate what event has happened, or what phase of congestion control TCP is in (as appropriate), from the following set: Slow-Start, Congestion-Avoidance, Fast-Retransmit, and Timeout.

```
A: Slow-Start
B: Timeout
C: Congestion-Avoidance
D: Fast-Retransmit
E: Congestion-Avoidance
F: Timeout (retransmit timeout)
G: Slow-Start
```

   ii. Assume CWND is 10,000 right before F, what is the value of `SSTHRESH` at G?

   - On timeout, `SSTHRESH` is set to be half of the current CWND, hence `SSTHRESH` at G is 5,000.



💡 Comments:
I have some problems about this question, would you mind talking about the graph in details in the supervision?

Fairness

a. Two users, one using Telnet and one sending files with FTP, both send their traffic out via router R. The outbound link from R is slow enough that both users keep packets in R's queue at all times. Consider

outbound traffic only. Assume Telnet packets have 1 byte of data, FTP packets have 512 bytes of data, and all packets have 40 bytes of headers.

Discuss the relative performance seen by the Telnet user given the following queuing policies for R.

- We have two flows, Telnet has packets with 1 byte of data and 40 bytes of header, FTP has packets with 512 bytes of data and 40 bytes of header

i. Round-robin service

- The router will serve the Telnet and FTP flow in turn, 1 packet at a time, but since FTP packet (552 bytes) is much larger than Telnet packet (41 bytes), consecutive Telnet packets are served with a long delay in-between, hence Telnet user will experience extensive delay.

ii. Fair queuing

- Fair queuing approximates bit-by-bit fluid flow system, because packets cannot be divided or preempted. In a fluid flow system, for a time period T sending one packet of FTP packet, we can simultaneously send $\lfloor 552/41 \rfloor \approx 13$ Telnet packets. Hence in fair queuing, we would send 13 Telnet packets, then 1 FTP packet in a round-robin fashion.
- Telnet users would observe periodic delay because 13 Telnet packets can only transfer a limited amount of data.

iii. Modified fair queuing, where we count the cost only of data bytes and not IP or TCP headers.

- Consider FTP packet with 512 bytes and Telnet packet with 1 bytes, for a time period T sending one packet of FTP packet, we can simultaneously send 512 Telnet packets. Hence in modified fair queuing, we would send 512 Telnet packets then 1 FTP packet in a round-robin fashion.
- 512 packets a time is sufficient to provide a smooth user experience that Telnet user would less likely to observe periodic delay.

💡 Comments:

Notes

- Transport layer turns the host-to-host packet delivery service into a process-to-process communication channel.
- The application layer above have certain requirements that transport layer needs to provide, and the underlying network layer only provides a best-effort level of service, packets can be corrupted, dropped, reordered, delayed and duplicated.
- Transport layer uses (1) UDP provides a synchronous multiplexing (multiple combined into one)/demultiplexing service and TCP also provides a reliable, in-order, byte-stream data transfer service with flow and congestion control; (2) uses Remote Procedure Call to provide a request/reply service; (3) uses Real-time Transport Protocol to provide a Real-time service
- OS stores mapping between **sockets** and **ports** (`(local port, local IP address) <-> socket`), a packet carries a source and destination port number in its transport layer header, and an IP source and

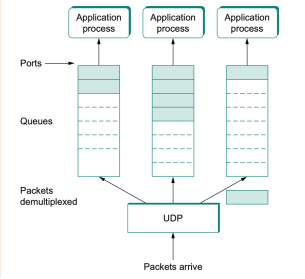
destination IP address in the IP header. Host uses IP addresses and port numbers to direct the message to the appropriate socket (demultiplexing).

- Socket: software abstraction for application processes to exchange messages with the transport layer in OS;
- Ports is a transport layer identifier, decide which socket gets which packet, e.g., ssh:22, http:80, https:443

- **UDP** adds a **demultiplexing service** on top of the best-effort service provided by the underlying network, allowing applications to identify each other on a host as there will be many processes running on the same host.
  - Lightweight communication between processes, avoid overhead and delays of ordered, reliable delivery. It contains destination IP address and port to support multiplexing; optional error checking on the packet contents (checksum).
  - A port number (16 bits) is used as a unique identifier for a process on the same host, sender sends messages to the port, receivers receive messages from the port.

How client set up connection with the server	Description
Well-known port per service	- clients send messages to a well-known port of that service (e.g., DNS sever at port 25) - clients and server agree on the port number they will use for subsequent communication, leaving the well-known port for others
Port mapper service	- only a single well-known port where the port mapper service accepts messages - port mapper service returns the appropriate port for the client to talk to the service - makes it easy to change the port associated with different services over time and for each host to use a different port for the same service

  - UDP provides **checksum** to ensure the correctness of the message, it is optional in IPv4 but mandatory in IPv6.
    - Checksum uses the UDP header, the message body and a pseudoheader (contains protocol number, source IP address, destination IP address, UDP length), this is to verify message has been delivered between the correct two endpoints (e.g., modifying destination IP address during packet transit would be detected)
  - UDP packets demultiplexed into ports, each maintains a **message queue**, but there is **no flow-control mechanism** to slow the sender, packets will be discarded if queue is full



- IP packet contains IP header, IP data, and TCP packet (TCP header and TCP data segment), IP packet is no bigger than Maximum Transmission Unit (MTU), TCP data segment is no more than

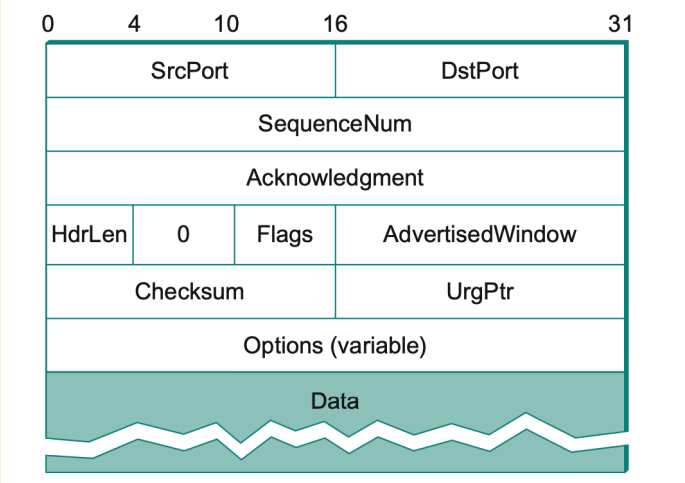
Maximum Segment Size (MSS) bytes

- MSS = MTU - (IP header) - (TCP header)
- e.g., Ethernet: MTU of 1500 bytes, IP header ≥ 20 bytes and TCP header ≥ 20 bytes, we have MSS ≤ 1460 bytes

- **TCP** offers a reliable, in-order byte-stream service, full-duplex protocol. Despite demultiplexing service, it also provides a flow-control mechanism and congestion-control mechanism.

  ▼ TCP segment format: each segment carries a segment of the byte stream

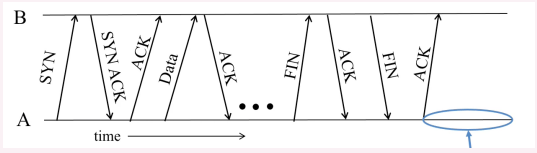
```
// involved in demultiplexing service
SrcPort: source port, multiplexing
DstPort: destination port, demultiplexing
// involved in sliding window algorithm
SequenceNum: sender -> receiver: sequence number for the first byte of data in that segment
Acknowledgement: receiver -> sender
AdvertisedWindow: receiver -> sender
// involved in relay control information between TCP peers
Flags:
  (1) SYN: establish a TCP connection; (2) FIN: terminating a TCP connection;
  (3) ACK: set any time the Acknowledgement field is valid;
  (4) URG: urgent data in the segment; (5) PUSH: sender invokes push operation;
  (6) RESET: signifies the receiver has received something unexpected, so abort the connection;
Checksum: error detection, computes over the TCP header, data, pseudoheader (as in UDP),
          checksum required in both IPv4 and IPv6
UrgPtr: used with URG flag to indicate urgent data
        such as application control signal (abort, control+C)
```



  - TCP demultiplexing key: `<SrcPort, SrcIPAddr, DstPort, DstIPAddr>`
  - **Flow control** prevents users from over-running the capacity of receivers, an end-to-end issue
  - **Congestion control** prevents too much data from being injected into the network, causing switches or links to become overloaded, an host-network issue
  - **TCP problems**: (1) misled by non-congestion losses, (2) fills up queues leading to high delays, (3) short flows complete before discovering available capacity, (4) AIMD impractical for high-speed links, (5) sawtooth throughput too choppy for some applications, (6) unfair heterogeneous RTTs, (7) tight-coupling with reliability mechanism, (8) end-hosts can cheat
    - Can be **fixed with routers**. (1)(2) can be solved by routers telling endpoints if they are congested, (3)(4)(5)(6) can be solved by routers telling endpoints what rate to send to, (7) can

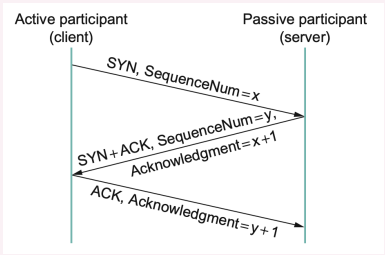
be solved by routers enforcing fair sharing

- **TCP three-way handshake algorithm** for establishing and terminating a connection:
  - Properties:
    - For the establishment - (1) acknowledgement identifies the next seqNum expected, (2) timer scheduled for the first two segments, (3) random initial seqNum (**ISN**) protects against two incarnations of the same connection reusing the same seqNum, preventing spoofing, hacking or conflicting with other data bytes transmitted over TCP
    - For the termination - a `TIME_WAIT` period to avoid reincarnation, we cannot move to the `CLOSED` state before $2 \times TTL$ because if `ACK` for `FIN` is lost in the network, then retransmission of `FIN` would cause direct termination of later incarnation if the host directly tore down and a later host connects

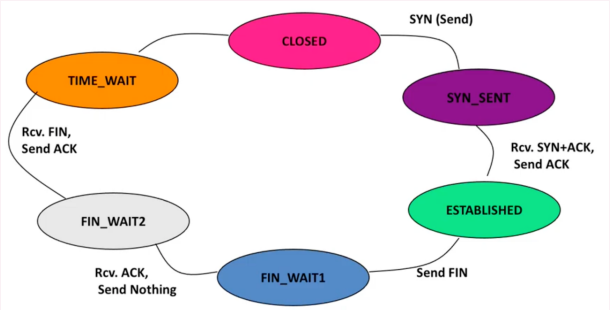


Reincarnation: B will retransmit FIN if ACK is lost

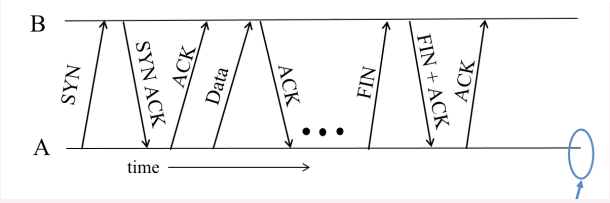
▼ Establishment: (1) Client → Sever: stating the initial sequence number it plans to use `<Flags = SYN, SequenceNum = x>;` (2) Server responds with a single segment: acknowledges the client's seqNum `<Flags = ACK, Ack = x + 1>`, and states its beginning seqNum `<Flags = SYN, SequenceNum = y>`, both bits are set in the flag field; (3) Client responds with a segment acknowledges the server's seqNum `<Flags = ACK, Ack = y + 1>`, can piggyback data on this ACK



▼ Termination: (1) Either side can initiate a FIN(finish) to close and receive remaining bytes, (2) another host ACK the bytes to confirm, then the connection is half-closed at the FIN sender side, (3) The other host can generate a FIN later, the first host ACK, then the connection is closed on both sides, (4) `TIME_WAIT` period (e.g., $2 \times TTL$) to avoid reincarnation, (5) have two types of termination: (5.1) normal termination, (5.2) abrupt termination using `RST`



A view from the client side

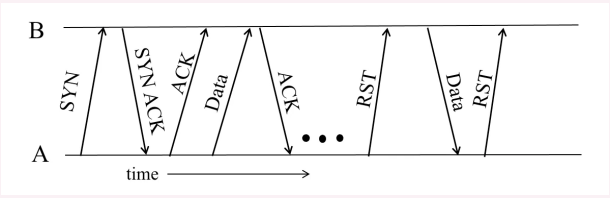


Normal termination (e.g., both terminated together):

A → B: FIN

B → A: FIN + ACK

A → B: ACK



Abrupt termination (e.g., an application at A crashed):

A → B: RST

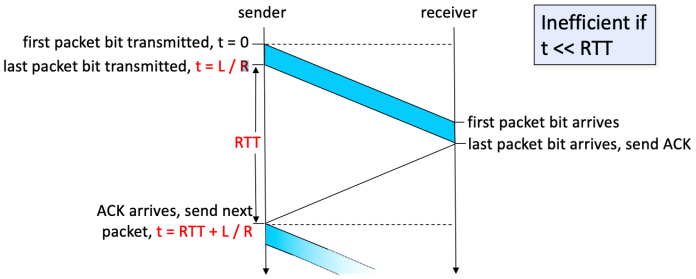
If B does not ACK the RST (lost), then any data in flight would be lost, so A reset RST until ACK received

- **Reliable data transfer of TCP**: (1) Checksum for bit error detection, (3) Timer for loss detection (i.e., corrupted acknowledgements for a packet with a particular sequence number), (4) `Sliding window` for improving efficiency of `stop-and-wait` strategy when dealing with channels with errors and loss and duplications (packet data or ACKs), (5) Piggybacking increases efficiency but many flows may only have data moving in one direction
  - Sends `ACK` with the sequence number of the last packet received OK, get rid of the usage of `NAK` (negative ACK).

Types	Cumulative ACK	Selective ACK
Definition	ACK carries the next in-order sequence number that the receiver expects	ACK individually acknowledges correctly received packets, receiver can buffer out-of-sequence packets
Implementation	Go-Back-N (GBN): (1) sender transmits up to n packets, (2) receiver only accept packets in order and discard out-of-order packets, (3) sender sets a timer for 1^{st} outstanding $ack(A+1)$, if timeout, then retransmit $A+1, A+2, ..., A+n$	Selective Repeat (SR), send things we know are missing: (1) the sender transmits up to n unacknowledged packets (e.g., k, $k+1$ are sent), (2) receiver indicates only the lost packet k, (3) sender only retransmit the lost packet k
Properties		- more efficient, offer more precise information - requires more complicated book-keeping (timer per packet)

  - For the `stop-and-wait` strategy, only two sequence numbers $(0, 1)$ will suffice because the sender sends one packet and wait for the receiver to respond before sending the next one; duplicated packets are discarded.

    ▼ Problem: It limits the use of physical resources, inefficient has a low throughput if utilisation of the sender (fraction of time sender busy sending is low), e.g., $1KB$ packet every 30 ms gives us a throughput of 1KB/30 ms ≈ 33 KB/sec.



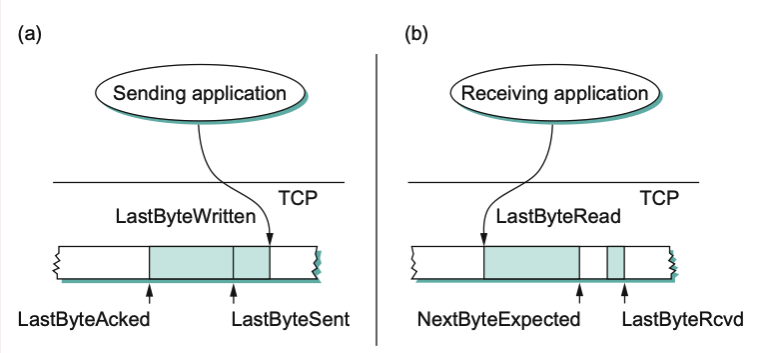
Suppose the propagation delay is 15ms, then RTT is 30ms, if we send a packet with 8000 bits on a link with bandwidth 1 Gbps. Then:

$$d_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \ bps} = 8 \text{ microseconds}$$

$$U_{sender} = \frac{L/R}{RTT+L/R} = \frac{0.008}{30+0.008} = 0.00027$$

- `Sliding window` **strategy**: (1) efficient by allowing multiple, in-flight, yet-to-be-acknowledged packets; (2) guarantees reliable and in-order data delivery; (3) enforces `flow control` with `AdvertisedWindow` field
  - Properties: (1) Throughput $\approx \frac{n}{RTT}$, where n is the size of the window, we can fully utilise the link provided n is large enough; (2) More complicated than the **sliding window algorithm** implemented on the network layer, because on the transport layer, it connects any two hosts, not two end-points on a physical link;

  ▼ Two buffers `MaxSendBuffer` and `MaxRcvBuffer`: (1) on the sending side maintains a send buffer to store data sent but not yet acknowledged and data written by the sending application but not yet send; (2) on the receiving side maintains receive buffer, holds data arrived but not yet read by the application, probably out of order; (3) receiver acknowledges the sender whenever it received data where all the preceding bytes have arrived, the window slides on successful reception/acknowledgement.

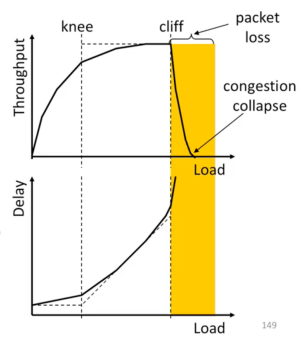


On sending end: LastByteWritten - LastByteAcked ≤ MaxSendBuffer
- LastByteAcked ≤ LastByteSent
- LastByteSent ≤ LastByteWritten
On receiving end: LastByteRcvd - LastByteRead ≤ MaxRcvBuffer
- LastByteRead < NetByteExpected
- NextByteExpected ≤ LastByteRcvd + 1

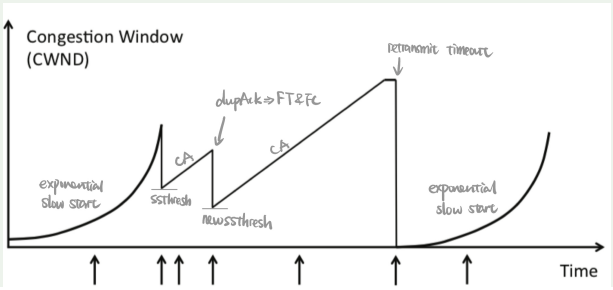
  - **Flow control** with `AdvertisedWindow` field: adjust the sending rate (#bytes/sec) without overflowing a slow receiver's buffer, an end-to-end issue determined by the receiver and reported to the sender
    - The receiver **advertise a window** size to sender indicating the amount of free space remaining in its buffer:
      - `w = AdvertisedWindow = MaxRcvBuffer - ((NextByteExpected - 1) - LastByteRead)`

- TCP on the send side then adhere to the `AdvertisedWindow` by making sure `LastByteSent - LastByteAcked <= AdvertisedWindow` to limit the amount of data it can send.
  - If `AdvertisedWindow = 0`, sender can no longer send any data, but in order to know any changes in the `AdvertisedWindow`, sender would persists in sending a segment of 1 byte periodically to get back the fresh info.
- Rate = $\frac{w}{RTT}$ bytes/sec, sender can send no faster that Rate. Sender window advances when new data acknowledged, receiver window advances as receiving process consumes data.

- **Congestion control** with `CongestionWindow(CWND)`: adjust the sending rate (#bytes/sec) to keep from overloading the network (routers), computed by the sender using congestion control algorithm
  - Reason of congestion - statistical multiplexing: if multiple packets arrive in a short period of time, a router can only transmit one, buffer others, delays the traffic. Arriving traffic may eventually overflowing the buffer, leads to packet drops.
  - Undesirable effects of congestion - more packet delay, greater chance of packet loss, low throughput
    - After the knee point → throughput increases slowly and delay increases fast
    - After the cliff point → throughput starts to drop to zero (congestion collapse) and delay approaches infinity



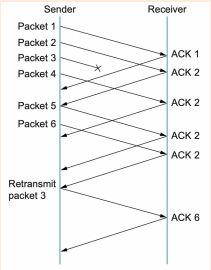
- **Slow start to estimate available bandwidth**: (1) start slow for safety and ramp up exponentially for efficiency; (2) Starts with a small congestion window (e.g., CWND = 1) → initial sending rate is MSS/RTT; (3) continues to double `CWND` each RTT until there is a loss, where a timeout causes multiplicative decrease to divide `CWND` by 2
  - Slow-start threshold (`ssthresh`): initialise to a large number, on timeout, set `ssthresh = CWND/2` and set `CWND = ssthresh`, sender switches from **slow-start to AIMD-style** increase (Congestion-avoidance phase)



- **Congestion-avoidance** - **Additive increase/multiplicative decrease** for adjusting window `CWND`, gives a sawtooth pattern:
  - Signal - **Packet loss** can indicate both congestion and bit error, we could deduce packet loss via:
    - (1) no ACK after timeout → more serious, not enough packets in progress to trigger duplicate ACKs or suffered several losses; (2) multiple duplicated ACKs → isolated loss of one packet
    - But in wireless links, packet drops are more common due to bit errors, so (1) 802.11 networks apply forward error correction (FEC) to correct errors; (2) link-layer retransmission so initial loss is not apparent to TCP; (3) distinguish between bit-error losses and congestion losses (which has increasing RTT and correlation among successive losses)
  - **Process**: (1) Interprets packet timeout as congestion, reduces transmitting rate; (2) each timeout, `CWND = CWND/2`, but not allowed to fall below the maximum segment size (single packet size, i.e., `CWND = 1`); (3) `CWND` is incremented by one MSS for each ACK, (in congestion avoidance, `CWND = CWND + 1/CWND`; in fast recovery phase, `CWND = CWND + 1`)

- **Fast retransmission**: optimisation or heuristic that uses duplicate acknowledges to trigger early retransmission
  - Solve the problem: coarse-grained implementation of TCP timeouts lead to long periods of time during which the connection went dead while waiting for a timer to expire, so fast retransmit is required

  ▼ Process: (1) Duplicate ACKs are a sign of an isolated loss, so we could trigger resend upon receiving k duplicate ACKs (e.g., TCP uses $k = 3$); (2) then set `CWND = ssthresh`, transmits the missing packet (avoid the case of a delayed packet rather than a lost packet), could either (2.1) send the missing packet and increase `CWND` by the number of duplicated ACKs, or (2.2) send the missing packet and wait for ACK to increase `CWND`;

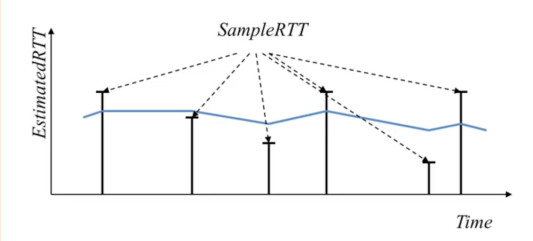


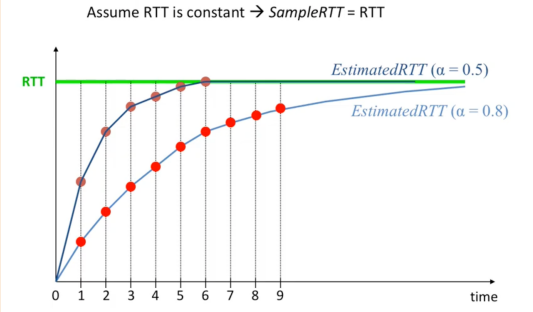
- Adaptive **retransmission timeout estimation** algorithm for reliable delivery:
  - The timeout is proportional to the RTT: $\text{Timeout} = 2 \times \text{EstimatedRTT}$, (1) a timeout too short leads to retransmitting packets that was just delayed, adding network load; (2) a timeout too long leads to low throughput

  ▼ Measuring RTT using **exponential averaging** of RTT samples: where α is selected to smooth the EstimatedRTT, we will eventually converge to RTT with more samples if assuming RTT is constant

$$\begin{gathered}\text{SampleRTT} = \text{AckRcvdTime} - \text{SendPacketTime} \\ \text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT} \\ \text{where} 0 < a \leq 1\end{gathered}$$

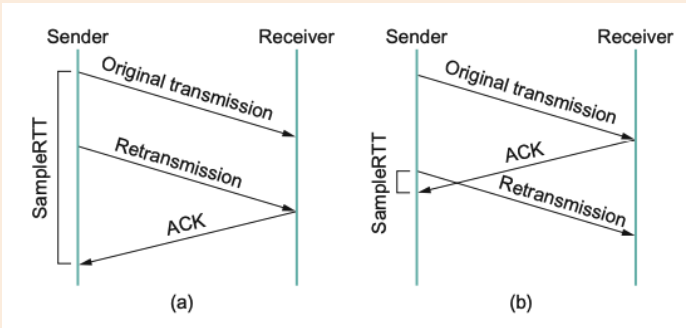
- A small α tracks changes in the RTT but is perhaps too heavily influenced by temporary fluctuations
- A large α is more stable but perhaps not quick enough to adapt to real changes





▼ Problem 1 - ambiguous measurements as we cannot differentiate between the real ACK and the ACK of the retransmitted packet → Solution 1 - Karn/Partridge Algorithm:

- (1) Measure $\mathrm{SampleRTT}$ only for original transmissions, once segment retransmitted, do not use its measurements;
- (2) Computes $\mathrm{EstimatedRTT}$ using $\alpha = 0.875$;
- (3) $\mathrm{Timeout\ value(RTO)} = 2 \times \mathrm{EstimatedRTT}$;
- (4) Employs **exponential backoff**: every time RTO timer expires, set $RTO \leftarrow 2 \times RTO$ up to a maximum ≥ 60 s; every time successful original transmission comes in, collapse RTO back to $2 \times \mathrm{EstimatedRTT}$.



Problem: ambiguous measurements
- assuming the ACK is for the original transmission but it was really for the second, then the SampleRTT is too large
- assuming the ACK is for the second transmission but it was actually for the first, then the SampleRTT is too small.

▼ Problem 2 - need to better capture variability in RTT → Solution 2 - Jacobson/Karels algorithm:

$$\text{Difference} = \text{SampleRTT} - \text{EstimatedRTT}$$
$$\text{EstimatedRTT} = \text{EstimatedRTT} + \delta \times \text{Difference}$$
$$\text{EstimatedDerivation} = \text{EstimatedDerivation} + \delta(|\text{Difference}| - \text{EstimatedDerivation})$$
$$\text{Timeout} = \mu \times \text{EstimatedRTT} + \phi \times \text{EstimatedDerivation}$$

- Where $\delta \in [0, 1]$, $\mu = 1, \phi = 4$ (normal case), the estimated derivation is the exponential average of the difference. (1) when the variance is small, timeout is close to EstimatedRTT, (2) when the variance is large, the variance causes the $\text{EstimatedDerivation}$ term to dominate the calculation

- **Fast recovery**: removes the slow start phase that happens when recovering from an isolated loss
  - Process: (1) Grant the sender temporary credit for each dupACK, to keep packets in flight; (2) at slow start, `ssthresh = CWND/2` and `CWND = ssthresh + dupACKCount`; (3) In fast recovery, `CWND = CWND + 1` for each duplicate ACK; (4) Exit fast recovery after receiving new ACK, set `CWND = ssthresh`, and increment `CWND = CWND + 1/CWND` for each ACK.

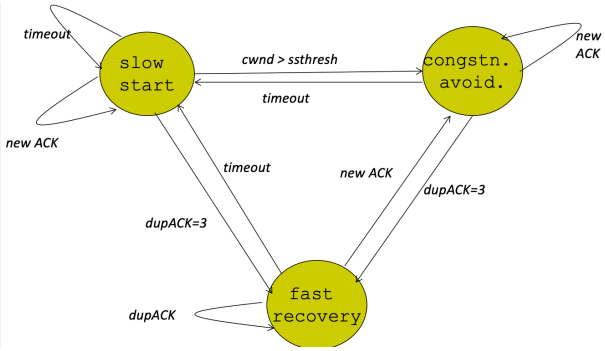
  ▼ Example: before fast recovery, no packets in flight when encounter a packet loss so ACK clocking (to increase CWND) stalls for another RTT; after fast recovery, another 10 packets in flight when encounter a packet loss

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 110) cwnd=6 + 1/5 (no xmit)
- ACK 111 (due to 101) ← only now can we transmit new packets

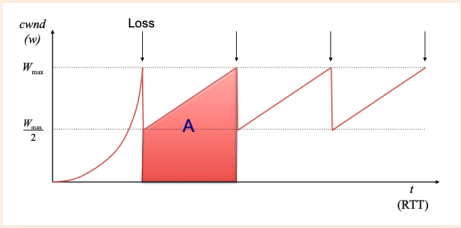
Before fast recovery, 10 packets (101-110) in flight, but packet 101 is dropped

- ACK 101 (due to 102) cwnd=10 dup#1
- ACK 101 (due to 103) cwnd=10 dup#2
- ACK 101 (due to 104) cwnd=10 dup#3
- REXMIT 101 ssthresh=5 cwnd= 8 (5+3)
- ACK 101 (due to 105) cwnd= 9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (xmit 111)
- ACK 101 (due to 108) cwnd=12 (xmit 112)
- ACK 101 (due to 109) cwnd=13 (xmit 113)
- ACK 101 (due to 110) cwnd=14 (xmit 114)
- ACK 111 (due to 101) cwnd = 5 (xmit 115) ← exiting fast recovery
- Packets 111-114 already in flight
- ACK 112 (due to 111) cwnd = 5 + 1/5 ← back in congestion avoidance

After fast recovery



- TCP throughput (bytes sent per second): $\text{TCP throughput } B = \frac{\text{MSS} \times A}{\frac{1}{2} W_{max} \times \text{RTT}} = \sqrt{\frac{3}{2}} \frac{MSS}{\text{RTT}\sqrt{p}}$
  - Calculation of TCP throughput: (1) assume packet drop rate/probability p, a sender will be able to send an average of $\frac{1}{p}$ packets before a packet loss, which is the area under each cycle $A = \frac{3}{8} W_{max}^2$; (2) The number of bytes $MSS \times A$ (maximum segment size $\times$ number of packets); (3) The time taken is $\frac{1}{2} W_{max} \times RTT$



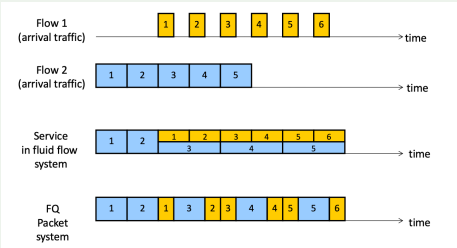
  - Problems & Solutions:
    - (1) Unfair for heterogeneous RTTs because flows get throughput inversely proportional to RTT → **Router-assisted** congestion control to ensure fairness among flows
    - (2) The TCP throughput is choppy and swings between $\frac{w}{2}$ and w, but some applications prefer sending at a steady rate → use an equation-based congestion control: measure drop percentage p and set the rate accordingly
    - (3) A high-speed TCP does not work well as the amount of data that can be sent between drops is not a practical number → (1) **Router-assisted approaches**, or (2) Multiple simultaneous connections; or (3) Set threshold speed, increase CWND faster (i.e., change from $p^{-.5}$ to $p^{-.8}$), let the additive constant in AIMD depend on CWND
      - e.g., assume RTT = 100 ms, MSS = 1500 bytes, to reach a throughput of 100 Gbps, $p \approx 2 \times 10^{-12}$ (packet drop probability), it takes $\frac{1}{2} W_{max} \times RTT \approx 16.6\ hours$ between drops, during this time ≈ 6 petabits can be sent

- Router-assisted Congestion Control aims to ensure:
  - (1) Max-min fairness between isolated flows using **fair queuing**:
    - Previous implementation: **FIFO queuing** with only one queue at the router drops packets arrive late, it has **problems**: (1) no flow isolation, cheating flows can benefit, (2) bandwidth share depends on RTT, hence no fairness among flows
    - Mental model - **Max-min fairness** in bit-by-bit round robin: (1) Flow bandwidth demands r_i and a total bandwidth C, max-min bandwidth allocations $a_i = \min(f, r_i)$, where f is the value

such that $\sum a_i = C$; (2) flows either get full demand or get f (i.e., if don't get full demand, no one gets more, similar to round robin with same packet size).

- e.g., $C = 10, r_1 = 8, r_2 = 6, r_3 = 2$: (1) $C/3 = 3.33$ can serve $r_3 = 2$, for the rest, $(C - 2)/2 = 4$, hence $f = 4$, $\min(8, 4) = 4, \min(6, 4) = 4, \min(2, 4) = 2$.

- **Fair queuing** approximates the bit-by-bit round robin model as packets with different sizes cannot be divided or preempted: (1) router classify packets into flows (e.g., same source & destination), each flow has its own FIFO queue, (3) compute the time at which the last bit of a packet would have left the router if flows are served bit by bit, (4) serve the packets in the increasing order of their deadlines in a fair fashion, take packets from the next flow in a fair order.
  - **Pros** than FIFO: (1) flow isolation, cheating flows don't benefit; (2) bandwidth share does not depend on RTT; (3) flows can pick any rate adjustment scheme they want
  - **Cons** than FIFO: (1) more complex, needs per-flow queue and additional per-packet book-keeping; (2) do not eliminate congestion, just manages it, we still want end-hosts to discover or adapt to their fair share



- **Weighted fair queuing** extends fair queuing by assigning different flows with a different share. It is implemented in almost all routers, mostly used to isolate traffic at larger granularities.

- (2) detecting congestion and do automatic adjustments using **Explicit congestion notification**:
  - Router tells the end-hosts about congestion by setting the ECN bit in the packet header;
  - Properties: (1) there is a tradeoff between link utilisation and packet delay, so router can decide when to set the bit; (2) congestion semantics can be exactly like a packet drop, the end-hosts reacts as though it saw a drop; (3) don't confuse corruption with congestion, recovery with rate adjustment; (4) can serve as an early indicator of congestion to avoid delays; (5) easier to incrementally deploy.

The sender-side window = minimum{CWND, RWND}
where CWND = the congestion window, RWND = the advertised window for flow control window
NB: in this course, assume RWND >> CWND, the CWND in a unit of MSS (maximum segment size, the amount of payload data in a TCP packet), in reality, CWND is in bytes.