Internet Protocol Stack

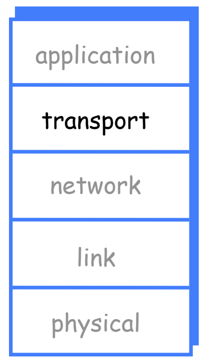
application: supporting network applications
- HTTP, SMTP, FTP, etc.

transport: endhost-endhost data transfer
- TCP, UDP

network: routing of datagrams from source to destination
- IP, routing protocols

link: data transfer between neighboring network elements
- Ethernet, WiFi

physical: bits "on the wire"

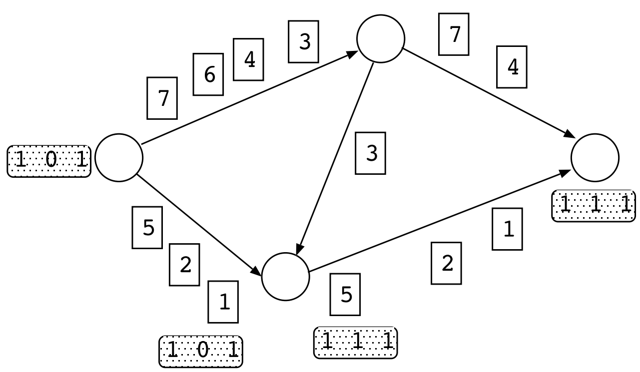


TCP: Transmission Control Protocol

Provides *reliability* on connectionless datagram network:
-
-
-

Link layer already provides sequencing and error control
Why do we need to provide reliability again at the transport layer?
-
-

Need for E2E Reliability



E2E Reliability

Causes of unreliable delivery
- re-routed packets
- bit error
- dropped/lost packets (due to congestion)
- system reboots

How to achieve reliable delivery?
Reliable delivery requires tools:
-
-
-

Sequence Number

With ARQ, packets must be numbered, why?

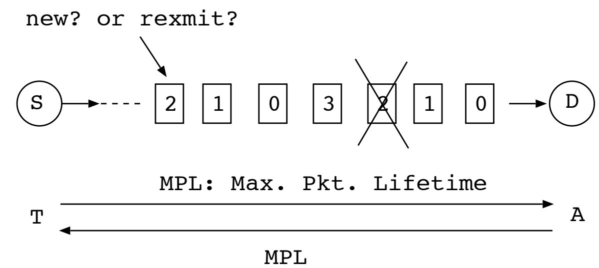
-
-

Sequence number space is finite. Issues:

1. sequence space size
2. sequence number wrap around
3. initial sequence number (ISN)

Sequence Number Space Size

If we had only 2 bits to keep track of sequence numbers:



What prevention?

Sequence Number Space Size

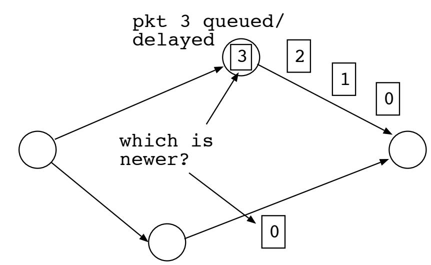
Let:
A: time taken by receiver to ACK packet
T: time sender continues retransmitting if ACK not received

Maximum Seqment Lifetime (MSL): $2\mathrm{MPL} + T + A$

Want: no seq. number may be duplicated within an MSL

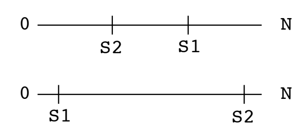
Sequence Number Wrap Around

Sequence space is finite and sequence number can wrap around:



Assuming s_1 and s_2 are not more than $N/2$ apart, $s_1 > s_2$ if either:

1. $s_1 > s_2$ and $|s_1 - s_2| < N/2$, or
2. $s_1 < s_2$ and $|s_1 - s_2| > N/2$

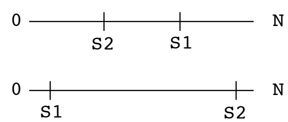


Required Sequence Number Size

Require that if $s_1 > s_2$, s_1 and s_2 are not more than $N/2$ apart ($|s_1 - s_2| < N/2$) within an MSL

Let μ be the transmission bandwidth

For TCP, $N = 2^{32}$-1, i.e., seq. number size $n = 31$ bits, want $\mu < (N/2)$/MSL or $2^n > \mu$*MSL



Example:
SF-NY MPL is 25 msec.
Let MSL = 2 min, for $n = 31$ bits,
μ must be < 17.8 MB/s (143 Mbps)

Initial Sequence Number (ISN)

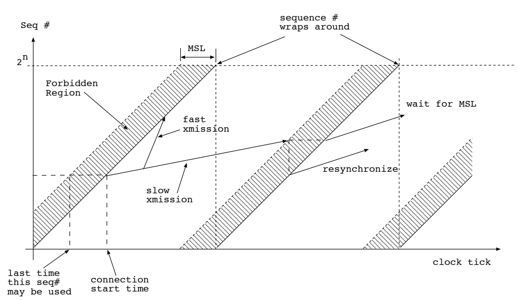
In case a connection got reincarnated, we must choose an initial sequence number that will not cause packets from the old connection to interfere

How can a connection be reincarnated?
-
-

Possible solutions:
1.
2.
3.

ISN from System Clock

Assume clock keeps ticking even when machine is down
Want: no seq. number may be duplicated within an MSL



What to do on hitting forbidden region?
1. wait for MSL before resuming transmission
2. resynchronize sequence number
either case, connection stalled

TCP's Handling of ISN

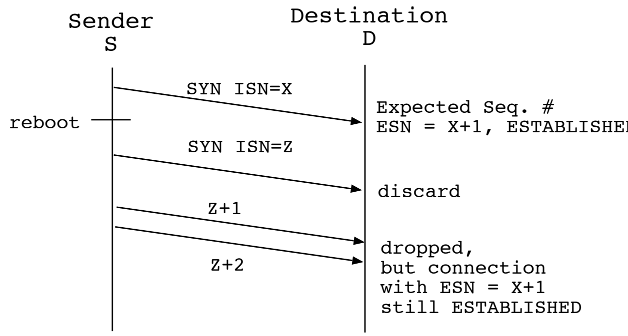
Connection identified by both addresses and port numbers and initial sequence number

Connection cannot be reused for MSL time
- on connection tear-down, wait for 2MSL (TIME-WAIT state)
  `bind: Address already in use`
- on reboot, do not create connection for MSL (2 minutes)
- on reboot, starts global ISN from 1

ISN carried in SYNchronization packet during connection establishment

Connection Establishment

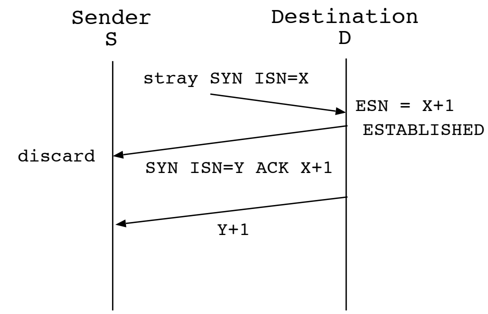
First try:



Lesson: connection request must be ACKed

Connection Establishment

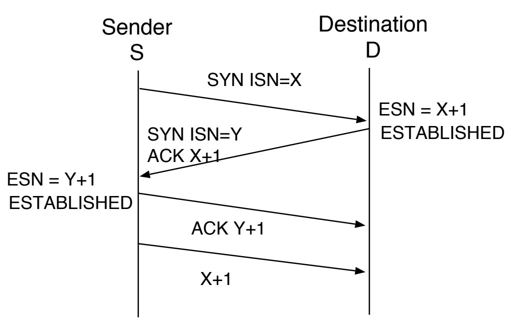
Second try:



Lesson: connection ACK must be ACKed or rejected

Connection Establishment

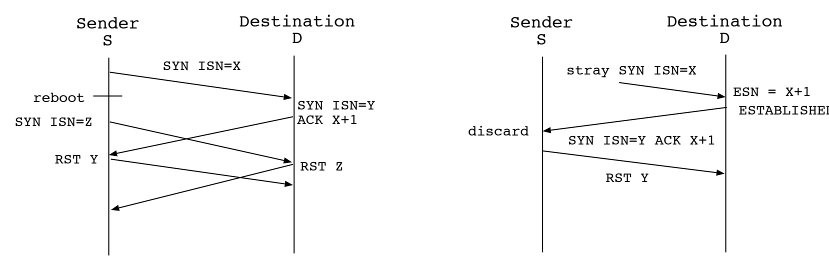
Three-way handshake:



SYN uses a seq#

Three-Way Handshake

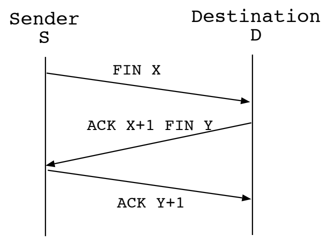
How three-way handshake solves the original problems:



Connection Tear-down

When to release a connection?
I.e., how do you know the other side is done sending and all sent packets have arrived?

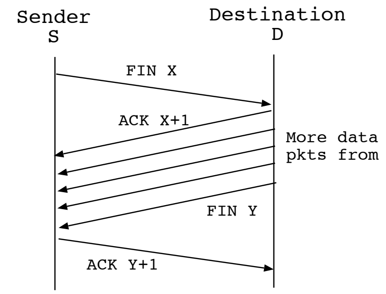
Use 3-way handshake to tear-down connection:



FIN also uses a seq#

Connection Tear-down

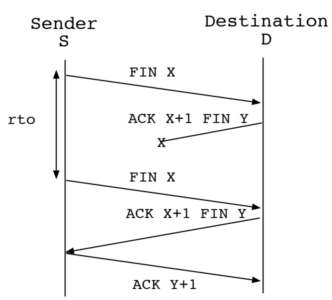
If the other side still has data to send:

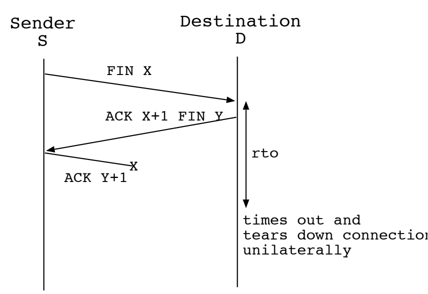


Why not send ACK X+1 along with FIN Y?

Connection Tear-down

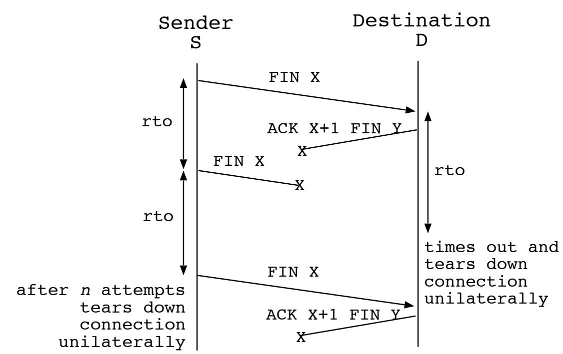
Still depends on timeout:





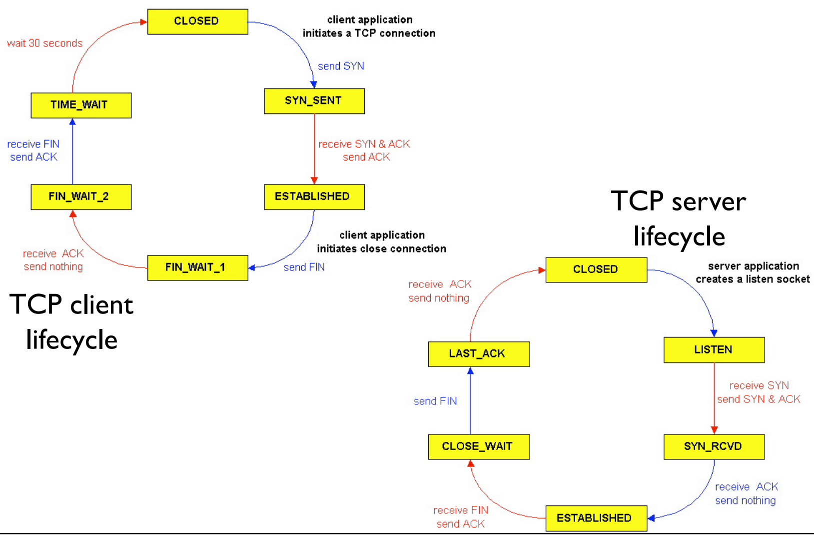
Connection Tear-down

Still depends on timeout:

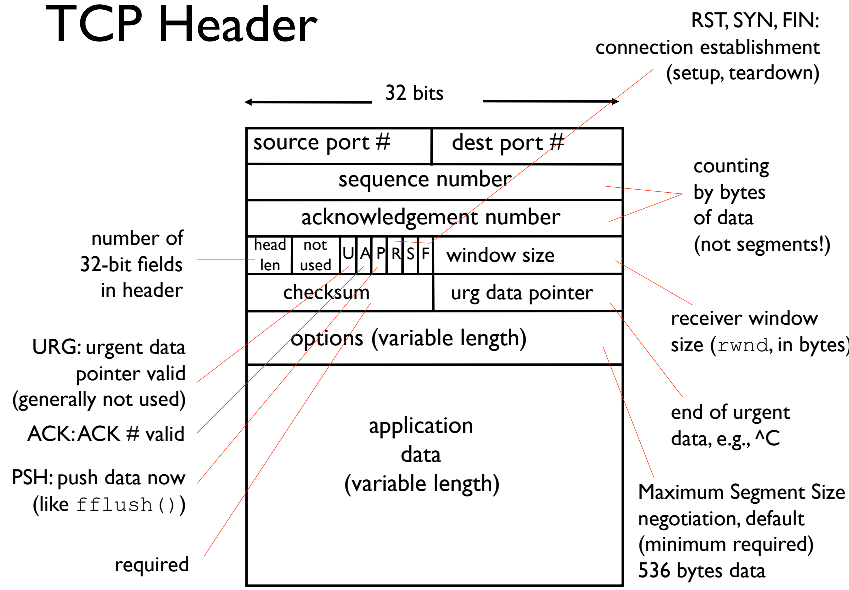


TCP connection tear-down depends on timers for correctness, but uses 3-way handshake for performance improvement

TCP Connection Management FSM



TCP Header



TCP Header Fields

Sequence number:

- data is sent in *segments* (= packets with seq#)
- sequence numbers count bytes sent
- ACK may be piggy-backed on data packet

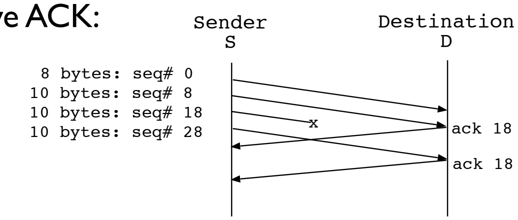
Checksum:

- checksum computed including pseudo IP header
- computed with all 0's in the checksum field
- 1's complement of result stored in checksum field

⇒ when checksum is computed at receiver, result is 0

TCP Cumulative ACK

- ACKs the last byte received in-order
- tells sender the next-expected seq#, i.e., if bytes 0 to i have been received, ACK says i+1
- subsequent out-of-order packets generate the same cumulative ACK:

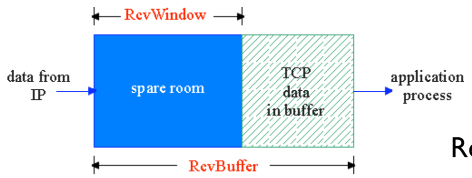


Advantage: lost ACK can be "covered" by later ACKs

Disadvantage: size of gap between two pkts not known to sender

TCP Flow Control

receive side of TCP connection has a receive buffer:



application process may be slow at reading from the buffer

flow control

sender won't overflow receiver's buffer by transmitting too much, too fast

Receiver advertises buffer space available by including the current value of `rwnd` in TCP header

Sender limits unACKed data to `rwnd`

⇒ guarantees receiver buffer doesn't overflow

TCP Flow Control Problems

Two flow-control problems:

1. receiver too slow (silly-window syndrome)
2. sender's data comes in small amount (Nagle's algorithm)

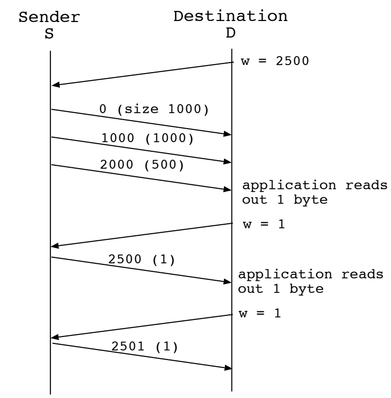
Silly-window syndrome:

receiver window opens only by a small amount, hence sender can send only a small amount of data at a time

Why is this not good?

1.

2.



Solution to Silly-window Syndrome

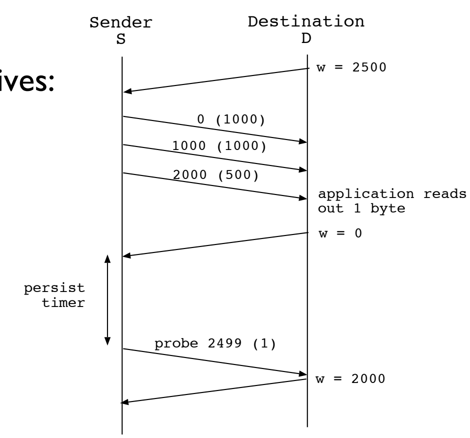
Don't advertise window until it opens "significantly"

(> ½*MSS or ½*`rwnd`)

Implementation alternatives:

- ACK with rwnd=0:
  sender probes after
  persistence timer goes off
- delayed ACK, but
  - not more than 500 ms, or
  - ACK every other segment

  (why?)



TCP Delayed ACK Generation

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send duplicate ACK, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediately send ACK, provided that segment starts at lower end of gap

Characteristics of Interactive Applications

User sends only a small amount of data, e.g., `telnet` sends one character at a time

Problem: 40-byte header for every byte sent!

Solution: "clumping," sender clumps data together, i.e., sender waits for a "reasonable" amount of time before sending

How long is "reasonable"?

Nagle Algorithm

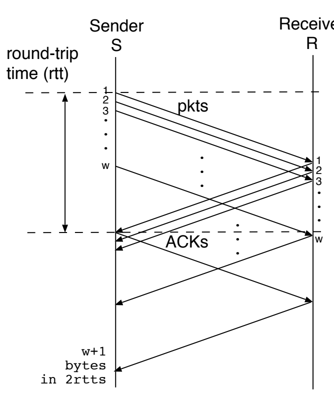
- send first segment immediately
- accumulate data until ACK returns, or
- ½ sender window or ½ MSS amount of data has been accumulated

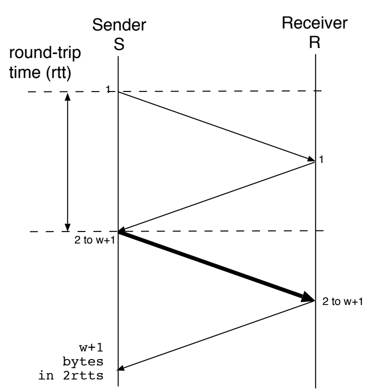
Advantages:
- bulk transfer is not held up
- data sent as fast as network can deliver

Can be disabled by `setsockopt(TCP_NODELAY)`

Nagle Algorithm

Nagle sends data as fast as network can deliver:





Retransmission Timeout

ARQ depends on retransmission to achieve reliability

Retransmission timeout (RTO) computed from round-trip time (RTT)

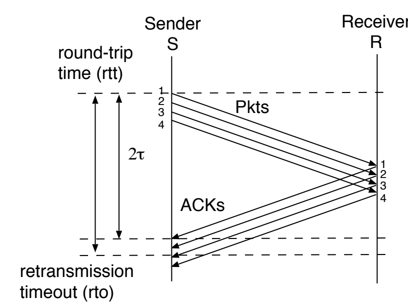
On the Internet, RTT of a path varies over time, due to:

•

•

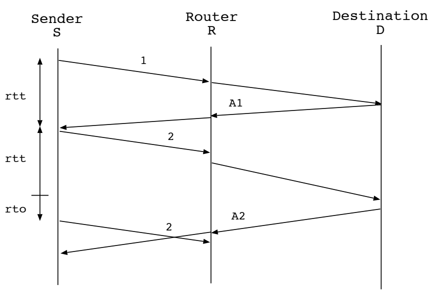
Varying RTT complicates the computation of:

1. retransmission timeout (RTO)
2. optimal sender's window size

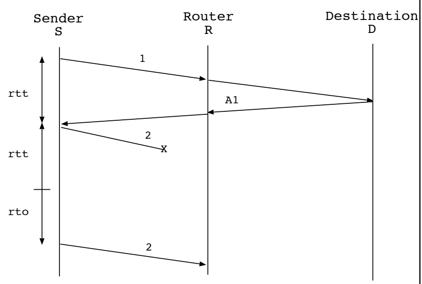


Implications of Bad RTO

RTO too small:
unnecessary retransmissions:



RTO too big:
lower throughput:



Estimating RTT

RTO must adapt to actual AND current RTT

sampleRTT: time between when a segment is transmitted and when its ACK is received

estimatedRTT computed by exponential weighted average

estimatedRTT = α*currentRTTestimate + (1-α)*sampleRTT,

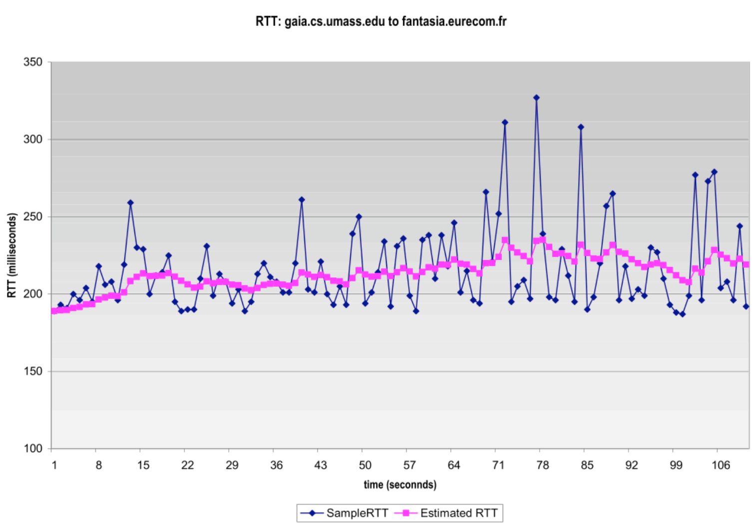
where α is the weight:

$\alpha \rightarrow 1$: each sample changes the estimate only a little bit

$\alpha \rightarrow 0$: each sample influences the estimate heavily

α is typically $7/8$ ($1-1/2^3$, which allows for fast implementation)

Example RTT Estimation

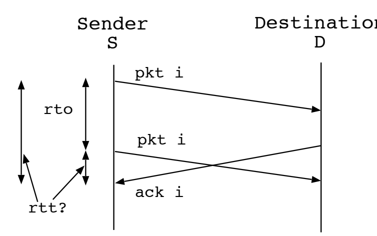


How to Compute RTO?

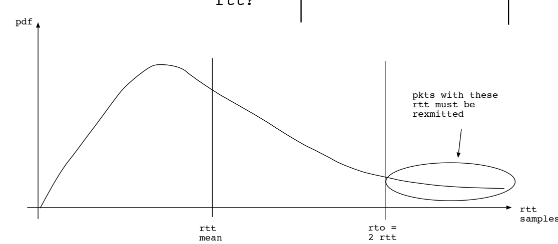
First try: RTO = β RTT, with β typically 2 or 3

Two problems:

1. which packet to associate with an ACK in case of retransmission?



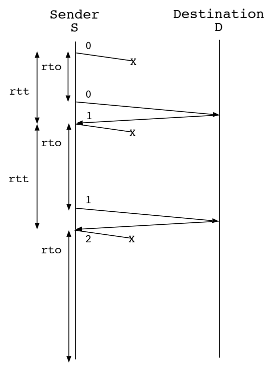
2. RTTs spread too wide



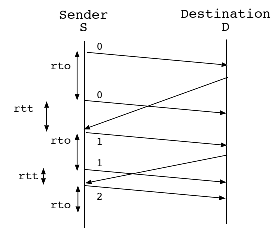
ACK Ambiguity

Which retransmitted packet to associate with an ACK?

1. original packet:
   RTO can grow unbounded



2. retransmitted packet:
   RTO shrinks



There is a feedback loop between RTO computation and RTT estimate

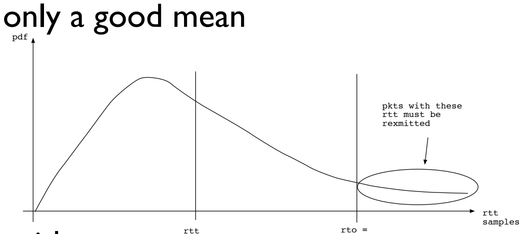
ACK Ambiguity: Karn's Algorithm

Karn's algorithm:

- adjust RTT estimate only from non-retransmitted samples
- however, ignoring retransmissions could lead to insensitivity to long delays
- so, back off RTO upon retransmission:
  $RTO_{new} = \gamma\, RTO_{old}$, γ typically = 2

RTT Spread Too Wide

RTT estimate computed using exponential weighted average gives only a good mean



Jacobson's algorithm:

- estimate the variance in sampleRTT
- use the deviation in sampleRTT (D) in RTO computation
- $D_{new} = \alpha\, D_{old} + (1-\alpha)$ |sampleRTT – estimatedRTT|
- compute new estimatedRTT
- RTO = estimatedRTT + $4D$

Timers Used in TCP

1. `TIME_WAIT`: 2* MSL
2. persistence timer
3. RTO
4. keep-alive timer: probe if connection idle too long
   may be turned on/off and idle period may be set using `setsockopt()`