GÉANT

# TCP Performance and Bulk Transfers

Alexander Gall, SWITCH
alexander.gall@switch.ch
Performance U! Winter School
Zurich, 6.-8.3.2013

# OVERVIEW

GÉANT

- Reliable, window-based transmission basics
- TCP congestion control mechanisms
- Failure of TCP on long fat networks
- High-Speed TCP variants
- Explicit congestion control methods
- Buffers, bloat, active queue management

# OVERVIEW OF BULK TRANSFERS

GÉANT

Goal: transfer a large set of data reliably with the maximum throughput offered by the network at any given time.

Basic service provided by TCP

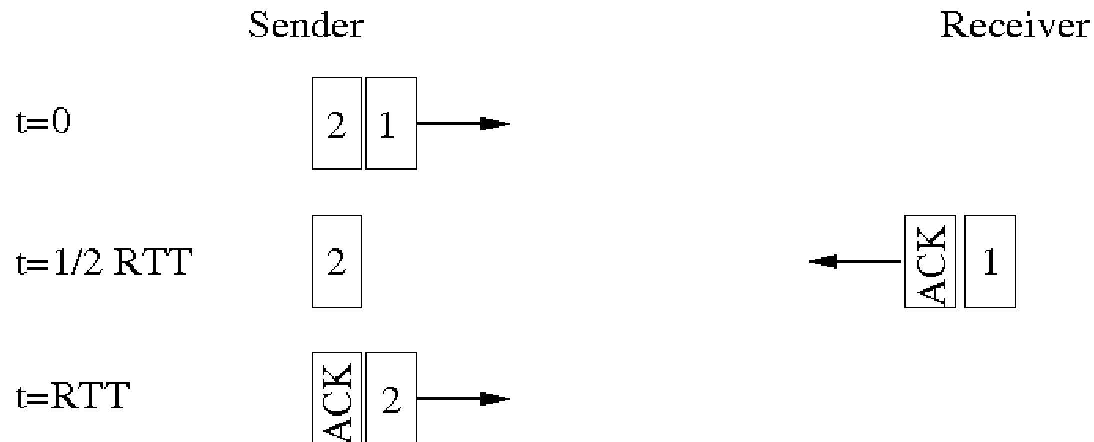
- Receiver acknowledges receipt of data, sender retransmits lost data.
- Sender numbers bytes so receiver can reconstruct proper order.
- Sender probes network for maximum data rate, backs off during congestion.

All application-layer protocols running on TCP can perform bulk transfers (HTTP, FTP, ...)

# WINDOW-BASED TRANSMISSION

GÉANT

- Trivial file transfer protocol (TFTP, RFC1350): send one packet, wait for acknowledgement (uses UDP).




- Throughput: 1 packet/Round Trip Time (RTT). Can be increased by sending more data before expecting acknowledgement. This is called the "window".
  - Send bigger packets (limited to MTU)
  - Send more packets per RTT

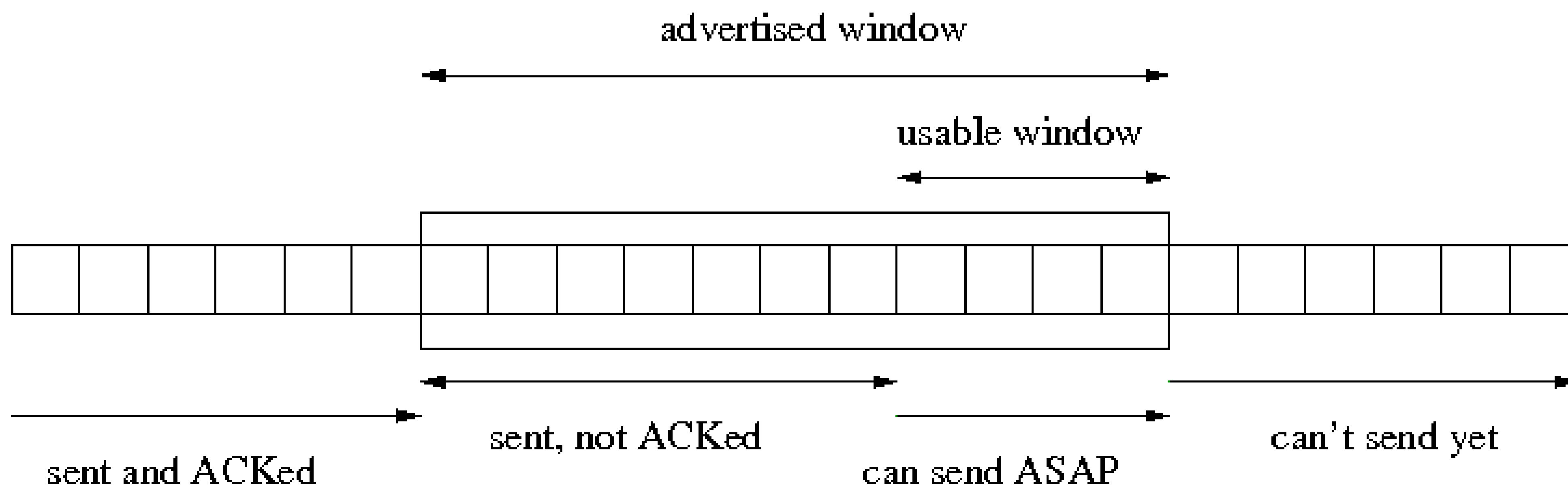
# HOW DOES WINDOW-BASED TRANSMISSION WORK?

Basic concepts:

- A packet is ‘in flight’ if it has been sent but not yet acknowledged.
- Only one ‘window-full’ of data can be in flight at any given time.
  - This is the maximum amount of data that can be in transit on the network path.
- The window ‘slides’.
  - As one packet is acknowledged and ‘leaves’ the window, another can be transmitted.
- The sender must buffer a packet until the receiver acknowledges receipt.
  - Needed for re-transmission in case of loss or corruption.
  - Means that the sender’s buffer limits the window size.

Window-based transmission is a key element of TCP.

# THE SLIDING WINDOW




- The window moves to the right as data is acknowledged.
- Advertised window = buffer available at receiver: sender may only send as much as the receiver can accept (receiver-based flow-control).

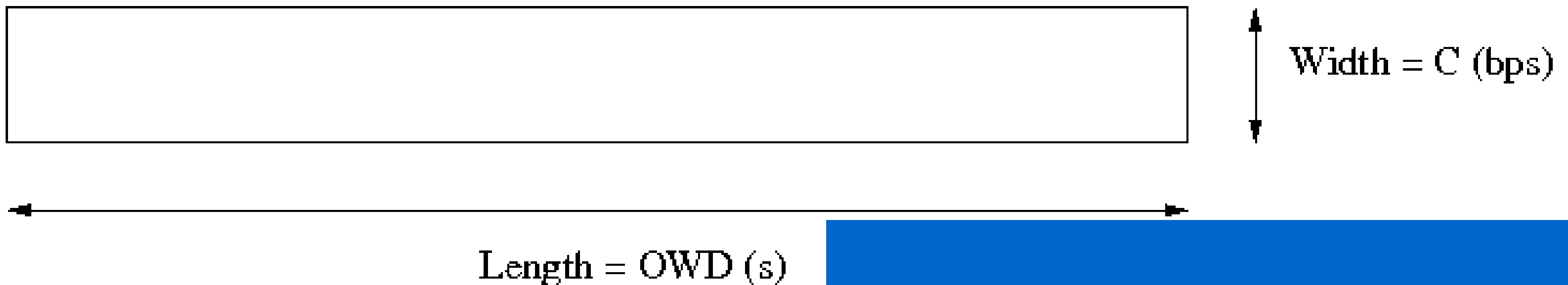
# WINDOW SIZE AND THROUGHPUT

GÉANT

- In the absence of packet loss:
  Throughput = Window / RTT.
- Increasing the window increases the throughput. Is there a value, after which further increase no longer has a benefit?
- Throughput is bounded by the bandwidth of the bottleneck link of a network path, i.e.
  $\text{Throughput}_{MAX}$ = bottleneck bandwidth
- Requires the window to be
  $\text{Window}_{MAX}$ = bottleneck bandwidth * RTT = BDP
- Introduces the **bandwith-delay product BDP**
- A smaller window limits throughput, a larger window has no effect

# LINKS, PATHS AND PIPES

- Link capacity C (aka bandwidth, often loosely called link speed): number of bits the link can transport per unit time, measured in bits per second (bps), e.g. FE = 100Mbps, GE = 1Gbps.
- Path: concatenation of links, characterized by OWD (one-way delay) and bottleneck capacity. Can be viewed as a "pipe":




- "Filled pipe" can hold C * OWD bits, e.g.

| C [Mbps] | OWD [ms] | Size of pipe[MiB] |
| --- | --- | --- |
| 100 | 50 | 0.625 |
| 1000 | 50 | 6.25 |

# WINDOW SIZE CALCULATION

- How large must the window be to keep the pipe filled? First ACK received after one RTT → window must be no smaller than the **bandwidth-delay product** (BDP):
  - Window size (bytes) ≥ Bandwidth (bytes/sec) * round-trip time (sec).
- Networks with large BDP are called 'Long Fat Networks' (LFNs or "elephants").

| C [Mbps] | RTT [ms] | BDP [MiB] |
| --- | --- | --- |
| 100 | 100 | 1.25 |
| 1000 | 100 | 12.5 |
| 10000 | 100 | 125 |

# HOW IS THE WINDOW SIZE DETERMINED?

- The window size is equal to the smallest of:
  - The sender's buffer size.
  - The receiver's advertised window size (bounded by receive buffer size).
- Used to need to guess BDP beforehand, tune buffers.
- Modern systems can dynamically tune buffers.
- TCP uses an additional *congestion window* to probe for effective BDP, more on this later.
- Note: window limited to 64KiB in original TCP. Need to use *window scaling option* (RFC1323) for windows up to $2^{30}$ bytes.

# WINDOW SIZE AND PERFORMANCE

GÉANT

- Window size is the most important factor influencing throughput in high-performance networks.
  - Degradation can occur if the Window size is too big or too small
- Window size too small:
  - Unused capacity.
- Window size too big:
  - Can monopolise system memory in popular web or file servers.
    - May run out of buffer space to open new connections.
    - May starve other processes of access to fast memory.
    - Not so relevant with auto-tuning
  - Can result in uneven bursts of traffic.
    - When large bursts of traffic combine with a network bottleneck, the result can be buffer overflows with resultant packet loss an

# TUNING A HOST FOR MAXIMUM TRANSMISSION RATES (1)

Tuning techniques and options vary between operating systems.

However, you may be able to (also see PERT Knowledge Base):

- Change the TCP buffer size
- Most systems today implement auto-tuning (dynamically adapt TCP send/receive buffers)
  - Microsoft: Windows Vista/7/8 (receive-side only, send-side tuning on "server" editions)
  - Linux 2.4 (send-side only)
  - Linux 2.6 (send/receive)
  - FreeBSD 7 and later (send/receive)
  - Mac OSX 10.5 and later (send/receive)

# TUNING A HOST FOR MAXIMUM TRANSMISSION RATES (2)

Auto-tuning on Linux

- Main control-knob for send/receive autotuning

  ```
  echo "<min> <default> <max>" > /proc/sys/net/ipv4/tcp_wmem
  echo "<min> <default> <max>" > /proc/sys/net/ipv4/tcp_wmem
  ```

  Buffer starts at <default>, can grow up to <max>
- Sender increases buffer with growth of cwnd
- Receiver estimates throughput by measuring received data per RTT, adjusts buffer/advertised window accordingly

# TUNING A HOST FOR MAXIMUM TRANSMISSION RATES (3)

- Use Large Send Offload (LSO) to reduce load on CPU
  - Increases the amount of data that can be sent by the CPU to the network adapter for transmission.
  - Also known as TCP Segmentation Offload and TCP Multidata Transmit.
  - Disadvantages
    - Can affect packet timing and cause burstiness
    - “Protocol Fossilization”: part of transport layer protocol implemented in hardware, makes it hard to add features (IPSec, TCP MD5, …) or different transport protocol (e.g. SCTP)
    - Additional complexity, bugs
    - Multi-core CPUs make this optimization less relevant

# TUNING A HOST FOR MAXIMUM TRANSMISSION RATES (4)

- Interrupt Coalescence
  - Increases amount of incoming data that can be buffered on the network adapter before it is sent to the CPU.
  - Decreases the number of interrupts that the CPU has to deal with.
  - Less relevant for multi-core CPUs that allow binding a connection to a core
- Checksum Offload
  - Network adapter verifies/generates checksums
- TCP Offload Engine (TOE)
  - Reduces load on CPU.
  - Moves all TCP processing onto the network adapter.
  - Same disadvantages as LSO (even worse)

# TUNING A HOST FOR MAXIMUM TRANSMISSION RATES (5)

"Swing of the pendulum effect"

- Network performance increases (e.g. 100Mbps → 1Gbps)
- CPU cannot saturate network
- People implement hardware assistance (LSO, TOE, …)
- CPU performance increases
- Hardware assistance obsolete
- Network performance increases (e.g. 1Gbps → 10Gbps)
- …

Current wisdom: turn these features off, except maybe checksum offloading.

# THE GOALS OF EARLY TCP (1)

GÉANT

Basic TCP functionality has changed little since RFC 793, published in 1981. The main goals of TCP were defined as:

- Reliability:
  - Each octet is given a sequence number. The receiver uses these to:
    - Eliminate duplicate octets.
    - Re-order out-of-sequence octets.
  - Each segment must be acknowledged by the receiver:
    - Cumulative ACKs: acknowledges all data up to this point (implies that loss on ACK-path is less critical).
    - If acknowledgement does not arrive before a timeout, the sender re-transmits the segment. Requires estimation of RTT.
  - Each segment is given a checksum:
    - Receiver uses this to spot corrupted or damaged segments.

# THE GOALS OF EARLY TCP (2)

GÉANT

- The main goals of TCP according to RFC 793 (continued):
  - Flow Control.
    - Receiver governs amount of data sent by sender through mechanism of window size.
  - Multiplexing.
    - Use of ports makes many simultaneous connections possible from one server.
  - Connections.
    - Established between a pair of sockets and the TCP implementation on each side.

# 'CONGESTION COLLAPSE' IN THE LATE 1980s

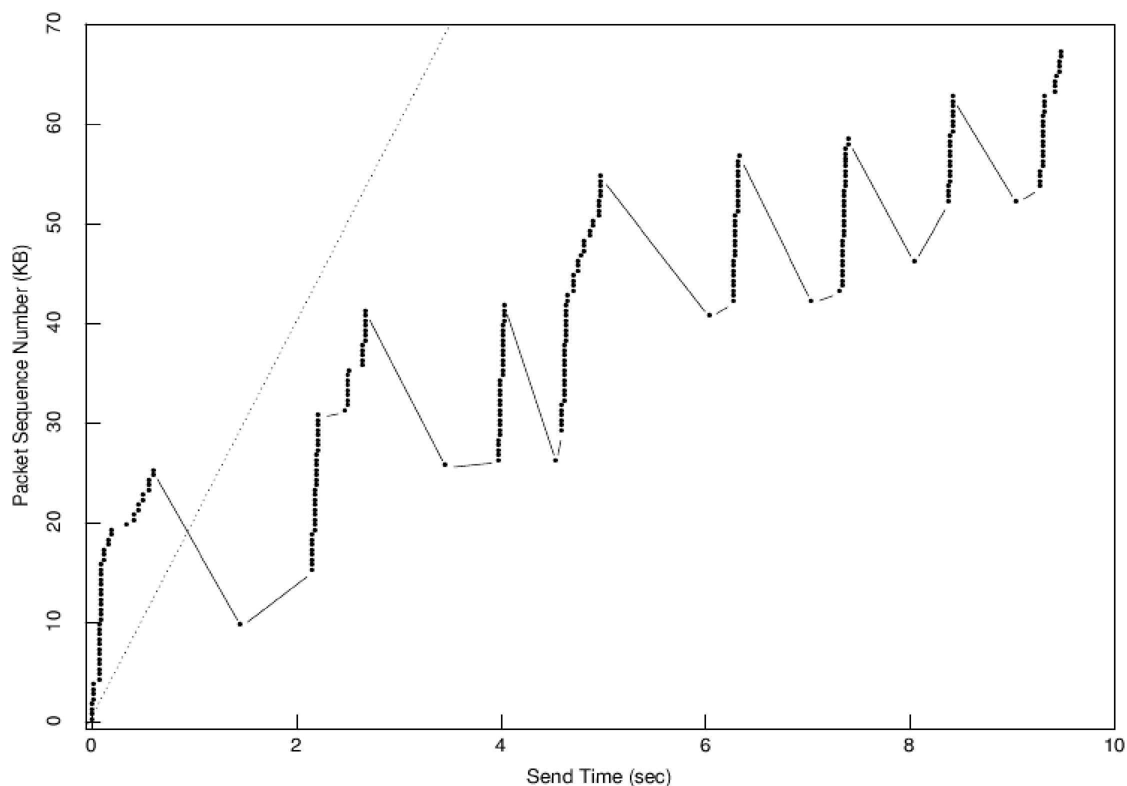
- By the late 1980s computer networks were experiencing 'explosive growth', but TCP could not cope with the increasing traffic.
- Often internet gateways would drop 10% of incoming packets due to local buffer overflows.
- From late 1986 onwards, the internet experienced a series of 'congestion collapses'.
- Van Jacobson and others identified aggressive sending strategy of early TCP as the source of the problem.

# WHY DID CONGESTION COLLAPSE OCCUR? (1)

- Original TCP: Window size was equal to the lesser of:
  - Sender's static buffer size.
  - Receiver's advertised window size.
- Window size was determined by capacity at both ends of the link, but it did not allow for bottlenecks between them. Neither did the window size take network congestion into account. The results:
  - On connection start-up, bottleneck gateways could be overwhelmed with packets.
  - Buffers overflowed.
  - Packets were lost and had to be retransmitted.
  - This created a vicious circle.

# WHY DID CONGESTION COLLAPSE OCCUR? (2)

- Early TCPs started by sending an entire window in a burst, overloading router queues immediately. Ethernet connected to a 230Kbps link with buffer (15kiB) < window (16kiB):




Van Jacobson, “Congestion Avoidance and Control”, ACM SIGCOMM Computer Communication Review, 1988, Volume 18, Issue 4, pp. 314-329.

# WHY DID CONGESTION COLLAPSE OCCUR? (3)

20KBps theoretical maximum rate

Self-clocking at 230Kbps

1 window, 10Mbps

Retransmit after timeout

1 window, 10 Mbps

Packet Sequence Number (KB)

0 10 20 30 40 50 60 70

0 2 4 6 8 10

Send Time (sec)

# INTRODUCTION OF CONGESTION CONTROLS

GÉANT

- In the late 1980s, TCP evolved to prevent congestion collapse. TCP “Tahoe” introduced new flow control mechanisms and TCP “Reno” refined them:
  - The Congestion Window
    - Makes window size dependent upon a connection’s available bandwidth.
  - Slow Start and Congestion Avoidance
    - Increase window size to probe a connection for available bandwidth.
  - Congestion Control Mechanisms
    - Decrease the TCP transmission window when congestion is detected.

# AVOIDING CONGESTION: THE CONGESTION WINDOW

- The transmission window size was made equal to the lesser of:
  - The sender's buffer size.
  - The receiver's advertised window size.
  - **The Congestion Window.**
- TCP implementations could dynamically resize the congestion window to:
  - Probe a connection for available bandwidth using 'slow start'.
  - Slowdown throughput growth before collapse occurred using 'congestion avoidance'.
  - Reduce the rate at which data was sent when congestion was detected.

# SLOW START

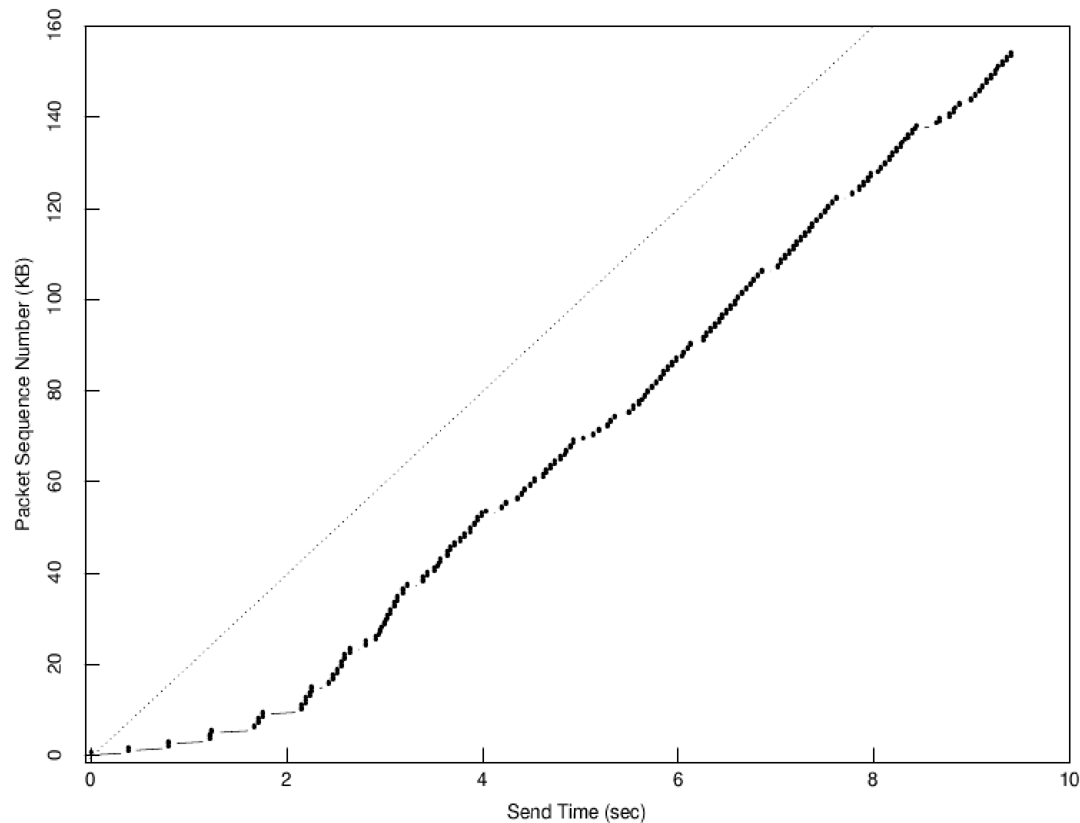
GÉANT

Slow start aims to avoid overwhelming bottlenecks with packets on start-up. It starts small and increases the rate of flow exponentially.

- On start-up, the congestion window is set to the Maximum Segment Size (MSS) of the connection.
- On each ACK, congestion window is increased by 1 MSS
  - Doubles congestion window every RTT.
- Congestion window continues to increase at same rate until:
  - The receiver's advertised window size is reached, or
  - The senders buffer size is reached, or
  - Congestion is detected in the connection, or
  - There is no traffic waiting to take advantage of increased window, or
  - The slow-start threshold is crossed.

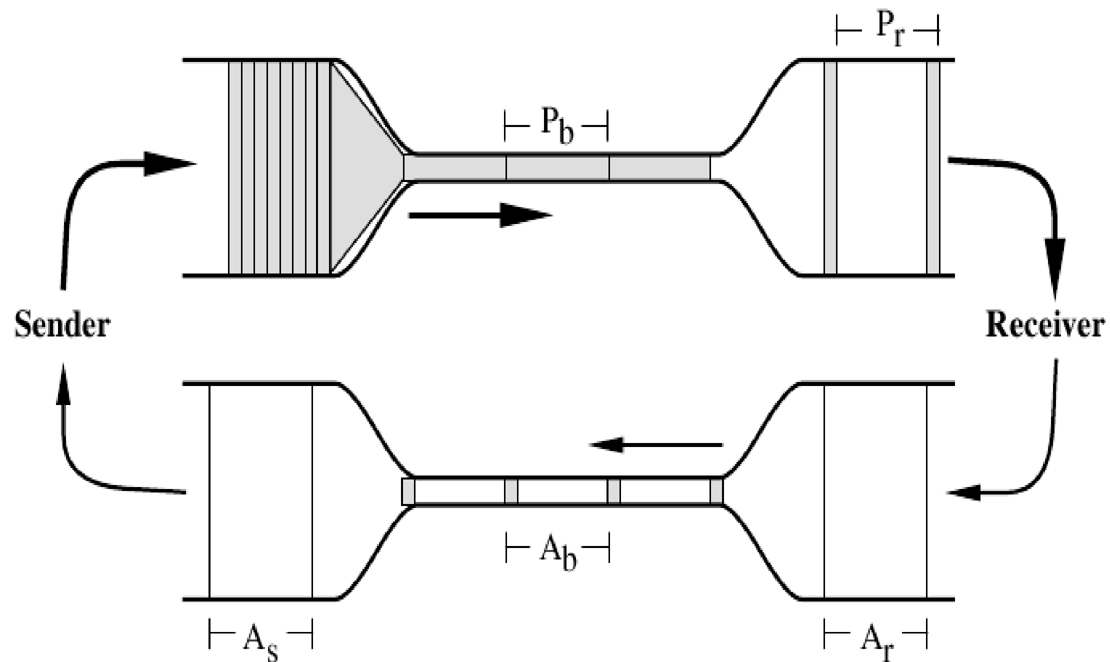
# CONGESTION COLLAPSE FIXED

- Same transmission as on slide 21 but with slow start (window-limited transfer, self-clocking)



# DIGRESSION: SELF-CLOCKING

- Sender has a window in flight. ACK spacing dictated by bottleneck bandwidth → window moves at the same clock.



# TWEAKING THE INITIAL CONGESTION WINDOW (1)

Time taken up by slow start to increase congestion window from 1 to BDP with respect to the path MTU (in units of RTT)

$$BDP = MTU\, 2^{T}$$

$$T = \log_2 \left( BDP / MTU \right)$$

Initial congestion window of N instead of 1

$$BDP = MTU * N * 2^{\bar{T}}$$

$$\bar{T} = T - \log_2 N$$

# TWEAKING THE INITIAL CONGESTION WINDOW (2)

Advantages

- Saves round trips for small transaction (a few KB)
- Reduces time to ramp up to large BDP (not very relevant in practice)

Disadvantages

- Initial burst can overflow buffers (if buffers are small or when there is congestion)

# TWEAKING THE INITIAL CONGESTION WINDOW (3)

- Original TCP spec: initial window = one segment
- RFC 2414, 1998: increase initial window from one segment to about 4KB (3 segments @ 1500 byte MTU). This is optional but is probably part of all current TCP stacks.
- RFC 3390, 2002 added some more analysis and results of experiments
- No change since then. 2009: Google experiments with 10 segments, claims that this causes no harm http://code.google.com/speed/articles/tcp_initcwnd_paper.pdf
  - Reduces user-visible latency by 10% (fewer round trips for small transfers)
  - No congestion observed
  - http://tools.ietf.org/html/draft-ietf-tcpm-initcwnd
  - Used in practice (e.g. Linux since 2.6.39)

# CONGESTION AVOIDANCE AND THE SLOW START THRESHOLD

- The slow-start threshold (ssthresh) is a TCP variable that is used to slow exponential congestion-window growth before congestion occurs (set to “infinity” on session start).
- ssthresh is TCP's memory of recently experienced congestion. Some systems cache ssthresh for a destination and use it as initial sshtresh for a new connection to the same destination.
- When the congestion window exceeds ssthresh, TCP’s strategy changes from ‘slow start’ to ‘congestion avoidance’.
  - During congestion avoidance, the congestion window is still increased in response to each ACK, but only in a slower, linear fashion (*additive increase*).
    - cwnd = cwnd + (1/cwnd)
    - Results in an increase of 1 MSS per RTT (cwnd ACKs, each adding 1/cwnd).

# THE SLOW START THRESHOLD AND CONGESTION AVOIDANCE: AN EXAMPLE

Slow start threshold reached.
Congestion avoidance begins.

Linear growth
During congestion avoidance

Congestion
Window
Size

Slow start
begins

Exponential growth
during 'slow start'

Time

# DEALING WITH CONGESTION: PRINCIPLES

When congestion is detected, the sender should respond by reducing its share of the available bandwidth.

- This means that it must reduce its rate of packet transmission.
- In practical terms, this means that it must reduce its transmission window size.

- Heavy congestion is deduced if a Retransmission Time-Out (RTO) occurs.
  - In other words, if a packet is not acknowledged before a timeout derived from actual RTTs, it is deemed lost due to congestion.
- Lighter congestion is deduced if three “duplicate ACKs” are received.

# DUPLICATE ACKNOWLEDGEMENTS

GÉANT

- Hole in the sequence umber space: receiver gets a segment but is missing at least one segment up to that point
- Receiver cannot acknowledge the new packet (ACKs are cumulative)
- Instead, sends an ACK for the highest sequence number up to which it has received all segments
- Additional segments will trigger the same "duplicate" ACK until the missing segment arrives
- A duplicate ACK tells the receiver three things
  - A packet has likely been lost (false positive in case of reordering)
  - Other packets are still getting through
  - A packet has left the network

# TCP'S RESPONSE TO CONGESTION

GÉANT

TCP's response depends upon whether heavy or light congestion was detected.

- Heavy congestion (an ACK times out):
  - Reduce cwnd to initial value.
  - Reduce ssthresh to half the flight-size when the ACK timed out (multiplicative decrease → exponential backoff, no less will do).
  - Enter slow start mode.
- Light Congestion (three duplicate ACKs are received):
  - Perform fast retransmit and fast recovery. Goal is to
    - Re-send lost packet.
    - Reduce the sending rate to half of that when the loss was detected.
    - Keep packets flowing until the re-transmitted packet is acknowledged

# FAST RETRANSMIT AND FAST RECOVERY

- Fast Retransmit:
  - Retransmit lost segment immediately (do not wait for RTO). Will take one RTT to be acknowledged.
- Fast Recovery, keep pipe from draining:
  - Reduce ssthresh to half the flight-size.
  - Duplicate ACKs don't move the window. Inflate congestion window artificially to keep packets flowing:
    - Set congestion window to ssthresh + 3 MSS (initial 3 duplicate ACKs).
    - Increase congestion window by 1 MSS for each extra duplicate ACK received.
  - On acknowledgement of retransmitted segment, reduce congestion window to value of ssthresh (reverses inflation during fast recovery) and enter congestion avoidance mode.

# THE WINDOW CLOSES AGGRESSIVELY
## (IN RESPONSE TO RETRANSMISSION TIMEOUT)

Congestion detected through timeout

Slow start threshold

Slow start threshold set to half the flight size at timeout

Congestion Window Size

Slow start begins

Time

Congestion detected through timeout

# THE WINDOW CLOSES MODERATELY
## (IN RESPONSE TO DUPLICATE ACKS)

# PROBLEMS WITH TCP CONGESTION CONTROL

Fast recovery does not handle multiple packet loss well.

- Assumes that a single packet was lost per RTT, due to cumulative nature of ACKs.
- If two or more packets are lost at the same time, the losses are dealt with in a 'serial' fashion.
  - E.g. Fast recovery reduces the cwnd to half the flight size and then repeats this operation for each lost packet.
  - Can lead to 'ACK starvation' and deadlock (recovery through timeout).

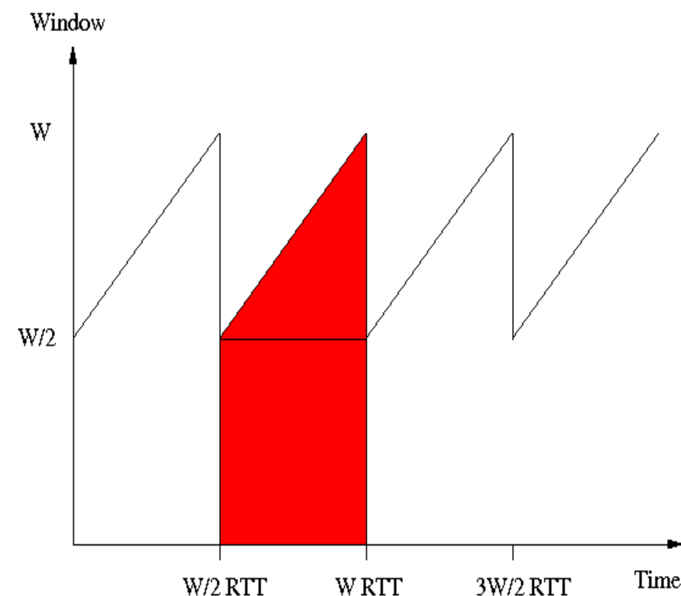
Modern TCPs (derived from "New Reno") use

- Modified fast recovery algorithm using "partial ACKs".
- Selective acknowledgements (SACK, RFC2018).

# SIMPLE MODEL FOR TCP THROUGHPUT (1)

- Path with constant RTT and average loss rate p. Assume
  - Perfectly periodic loss events, i.e. sequences of 1/p packets (of size MSS) followed by one lost packet.
  - Every packet is acknowledged, i.e. window increases by 1 per RTT, cycle lasts W/2 RTT seconds.
- Throughput = Data per cycle / cycle time
- Data per cycle (units of MSS): $\frac{3}{2}\left(\frac{W}{2}\right)^2 = \frac{1}{p}$

$$T = \frac{MSS}{p} \frac{1}{\frac{W}{2} RTT} = \frac{MSS}{RTT} \frac{2}{pW} = \frac{MSS}{RTT} \sqrt{\frac{3}{2p}}$$



# SIMPLE MODEL FOR TCP THROUGHPUT (2)

Observations:

- Lossy TCP behaves like window-limited lossless TCP

$$T = \frac{W_{eff}}{RTT} ; W_{eff} = MSS\sqrt{\frac{3}{2p}}$$

- The effective window as a function of average packet loss in units of the MSS is called the “response function” of TCP
- *Short RTT can tolerate more loss than large RTT*

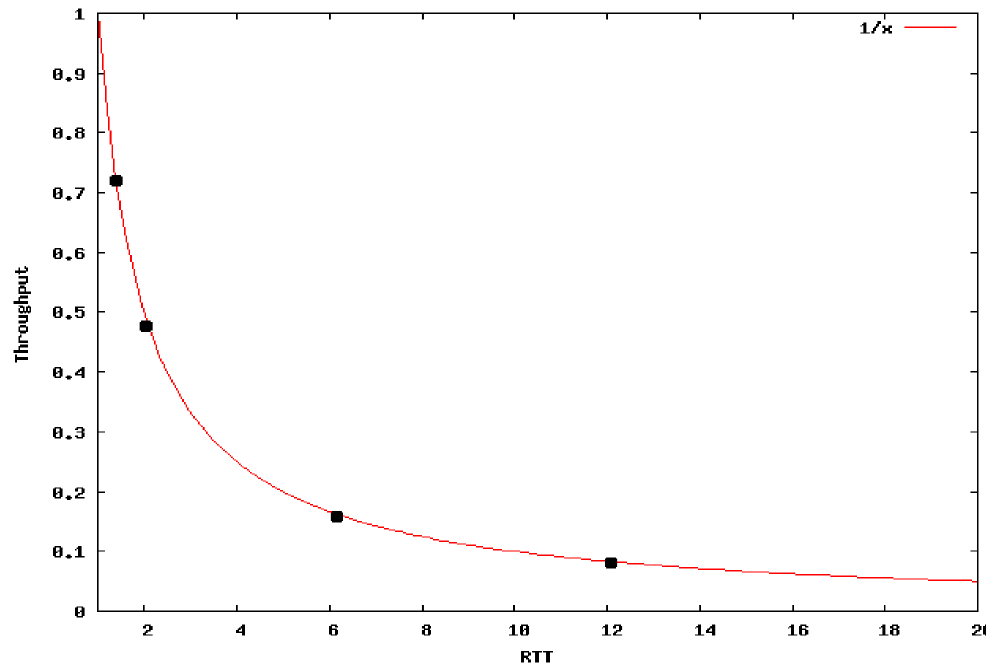
Helps to avoid a common pitfall! Consider this case:

- User reports low throughput on a large RTT path
- NOC on one end performs throughput test on a short segment of the path, finds much higher throughput
- NOC concludes that loss can't be on their part of the path

What's wrong with that?

# SIMPLE MODEL FOR TCP THROUGHPUT (3)

Plot throughput as a function of RTT. If it looks like this (arbitrary scale)




it is likely that the transfer is limited by loss and the loss is probably located on the segment common to all measurements.

# TROUBLE WITH THE ELEPHANTS (1)

GÉANT

Problem with 'traditional TCP' on Long Fat Networks (LFNs):

- BDP is high. Therefore:
  - Optimal congestion window is big.
  - Packet loss exponentially reduces congestion window.
  - Recovery from congestion takes too long, because of large congestion window and high round-trip time.
- Given realistic rates of packet loss, traditional TCP cannot maintain an optimal-sized average congestion window in an LFN. Therefore:
  - Throughput is sub-optimal.

# TROUBLE WITH THE ELEPHANTS (2)

GÉANT

An example:

- Consider a path of 10 Gbps at 100 ms RTT.
  - BDP = 125 MiB or 83333 1500 byte packets.
- It takes log(83333)/log(2) = 17 RTTs (1.7 seconds) to inflate congestion window to equal BDP during slow-start (initial cwnd = 1, ACK each packet).
  - Time is not a big issue, but increasing the window in huge chunks may be one
- It takes 41666 RTTs (4166 seconds) to inflate congestion window from half maximum size back to maximum size during congestion avoidance.
- With a bit error rate of $10^{-13}$, there will be a corrupt packet per 1000 seconds on average → unlikely to ever reach maximum window

# HIGH-SPEED VARIANTS OF TCP

GÉANT

Possible improvements for congestion avoidance algorithm

- Smaller reduction during multiplicative decrease: $W \rightarrow (1-\beta)W$
  - Regular TCP: $\beta = 0.5$
  - High-Speed: $\beta < 0.5$
- Larger increase of cwnd per RTT: $1 \rightarrow \alpha\ (> 1)$
- Non-linear increase during congestion avoidance
  - Recall regular TCP
    - Response function $W(p) \sim p^{-\frac{1}{2}}$
    - Linear increase of window by α per RTT from $(1-\beta)W_{max}$ to $W_{max}$, takes T RTTs

$$
\begin{aligned}
W(t) &= W_{max} - \alpha(T - t) \\
T &= \frac{\beta}{\alpha} W_{max}
\end{aligned}
$$

# HIGH-SPEED TCP: CUBIC (1)

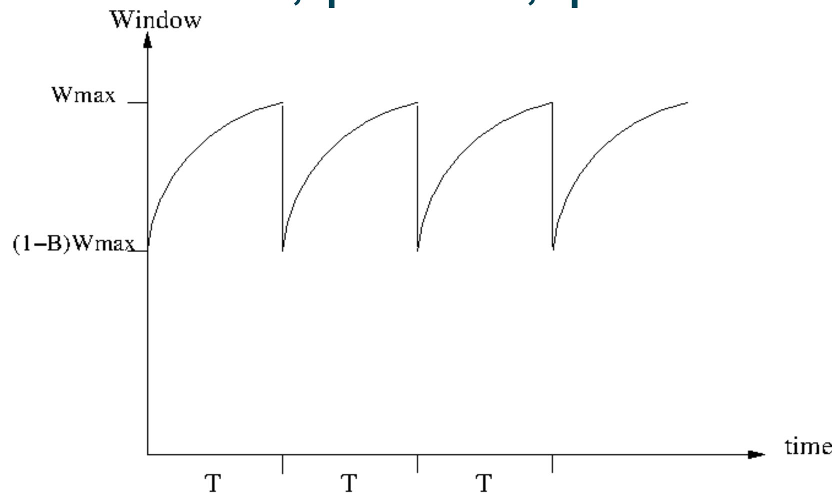
- Generalization

$$W(t) \quad \text{¿} \quad W_{max} - \alpha (T - t)^{\gamma}$$

$$T \quad \text{¿} \quad \left( \frac{\beta}{\alpha} W_{max} \right)^{\frac{1}{\gamma}}$$

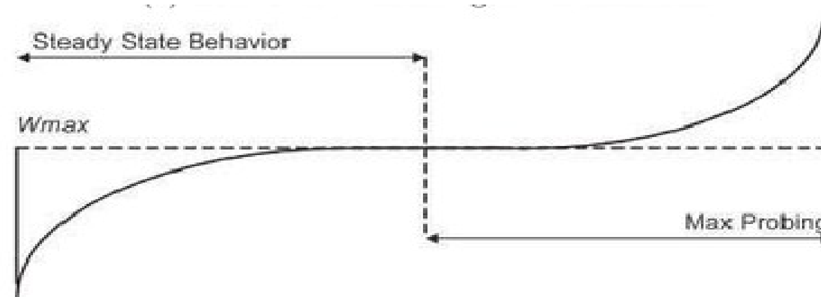
$$W(p) \quad \sim \quad p^{-\frac{\gamma}{\gamma+1}}$$

  - Regular TCP: α = 1, β = 0.5, γ = 1
  - CUBIC: α = 0.4, β = 0.2, γ = 3



# HIGH-SPEED TCP: CUBIC (2)

- Response function is still a power law
  - CUBIC morphs into standard TCP for low BDP paths
  - For high BDP paths, increase of cwnd during congestion avoidance is sped up
- The previous maximum cwnd is at the saddle point of the cubic growth function




- Used per default in Linux since 2.6.19

# HIGH-SPEED TCP: CUBIC (3)

GÉANT

- Previous example (10 Gbps, 100ms RTT)
  - # of RTTs to grow window from 0.8 $W_{max}$ to $W_{max}$ $$T=\left(\frac{W_{max}}{2}\right)^{\frac{1}{3}}=35$$
  - Standard TCP (½ $W_{max} \rightarrow W_{max}$): $$T=\frac{W_{max}}{2}=41666$$

# EXPLICIT CONGESTION CONTROL PROTOCOLS (1)

- 'Traditional TCP' congestion control mechanisms rely on information gathered by the sender and the receiver only.
  - I.e. they use information from the two ends of the path to deduce that the middle is congested.
- By contrast, explicit congestion control protocols rely on routers in 'the middle' of the path to supply information.
  - Routers return information (sometimes via the receiver) about:
    - Congestion that has been experienced, or
    - The optimal 'share' of capacity that should be allocated to the path.
  - This information is often held within packet-headers.
  - In response, the sender re-sizes its congestion window.

# EXPLICIT CONGESTION CONTROL PROTOCOLS (2)

- Advantage
  - Congestion notification is usually quicker and more sensitive.
- Disadvantage
  - Routers do more work and need to support Explicit Congestion Control Protocols; often they do not.
- Examples of Explicit Congestion Control Protocols:
  - Source Quench (now obsolete).
  - Explicit Congestion Notification (ECN)
    - More information about each of these is available in the PERT Knowledge Base.

# NETWORK TUNING FOR BETTER CONGESTION CONTROL (1)

- Router buffers
  - When incoming traffic exceeds outbound capacity, routers buffer packets in a queue.
  - When the buffer is full, newly arriving packets are simply dropped.
  - Whole bursts of packets can be dropped.
    - Can lead to synchronised traffic bursts and lower throughput.
  - Packets can remain in buffers for a long period.
    - Leads to increased one-way delay and round-trip times.
- In response to these issues, router buffers can be *actively managed*.

# NETWORK TUNING FOR BETTER CONGESTION CONTROL (2)

- Active Queue Management (AQM)
  - Network nodes send congestion signals to avoid buffers filling (ECN approach).
  - Most common form of Active Queue Management is Random Early Detection (RED).
    - RED is supported by many routers, but is not active by default.
    - Needs to be tuned.
    - Samples queue-size over time.
    - Can drop packets to keep queue-size small.
    - Short queue keeps one-way delay and round-trip time low.
    - Also helps to avoid synchronisation effects and related throughput degradation.
  - Variants exist, e.g. BLUE, which determines optimal queue length dynamically

# NETWORK TUNING FOR BETTER CONGESTION CONTROL (3)

- Sizing of Network Buffers
  - Approach 1: Traditional Wisdom
    - Suggests that network node should be able to buffer an end-to-end round-trip time's worth of line-rate traffic.
    - This is to accommodate bursts.
  - Approach 2: Suggested by recent research
    - Suggests that much smaller buffers are sufficient when there is a high degree of multiplexing of TCP streams.
- "Bufferbloat": excessive buffers in network equipment can cause huge delays (up to minutes have been observed in the wild)
  - Hosts
  - CPE devices
  - Telco networks ("we never drop a packet, and we're proud of it")
  - Messes up TCP's control loop