UiO : Faculty of Mathematics and Natural Sciences
University of Oslo

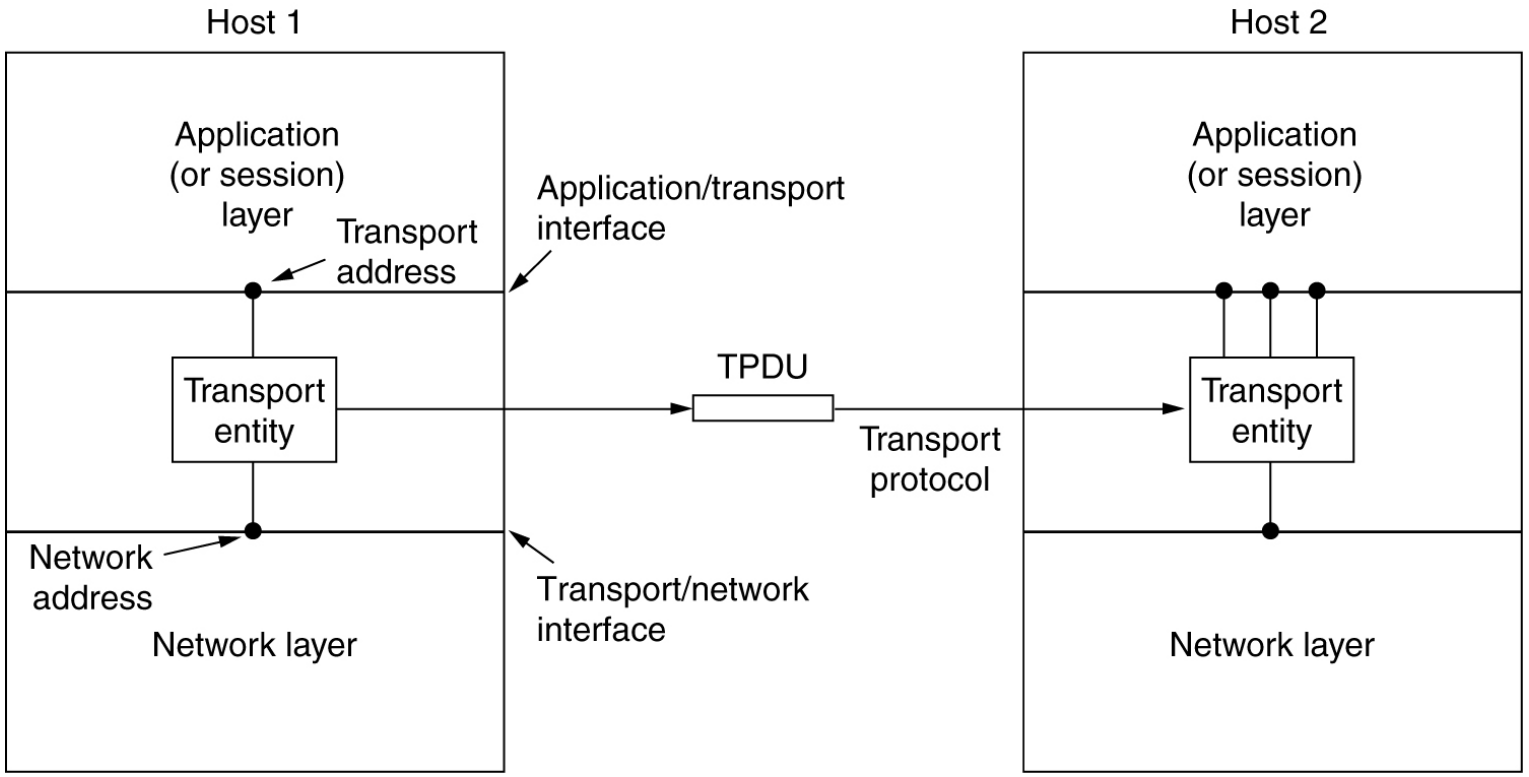
Department of Informatics
Networks and Distributed Systems (ND) group

INF 3190

The Transport Layer

Michael Welzl

Where we are in the stack...



Ambiguities: bandwidth

Traditional, "real" definition!

Mirriam-Webster online (http://www.m-w.com):

- Main Entry: band*width, Pronunciation: 'band-"width
  Function: noun, Date: circa 1937
  - 1 : a range within a band of wavelengths, frequencies, or energies; especially : a range of radio frequencies which is occupied by a modulated carrier wave, which is assigned to a service, or over which a device can operate
  - 2 : the capacity for data transfer of an electronic communications system <graphics consume more bandwidth than text does>; especially : the maximum data transfer rate of such a system

"Information rate"

- Unit: definition 1 - "Hz", definition 2 - "bit/s" (bps)

- Common interpretation in CN context:
  How many bits/sec can be transferred ("how thick is the pipe")

Ambiguities: bandwidth /2

- Various wooly “bandwidth“ terms
  - Nominal bandwidth: Bandwidth of a link when there is no traffic
  - Available bandwidth: (Nominal bandwidth - traffic) ... during a specific interval
  - Bottleneck bandwidth: smallest nominal bandwidth along a path, but sometimes also smallest available bandwidth along a path
- Throughput: bandwidth seen by the receiver
- Goodput: bandwidth seen by the receiving application (e.g. TCP: goodput != throughput)

Ambiguities: delay

- Latency - time to transfer an "empty" message
  - also: “propagation delay”
  - limit: speed of light!

- End2end delay = *latency* + *msg_length / bottleneck bandwidth* + *queuing delay*
  - just a rough measure; e.g., processing delay can also play a role, esp. in core routers (CPU = scarce resource!)

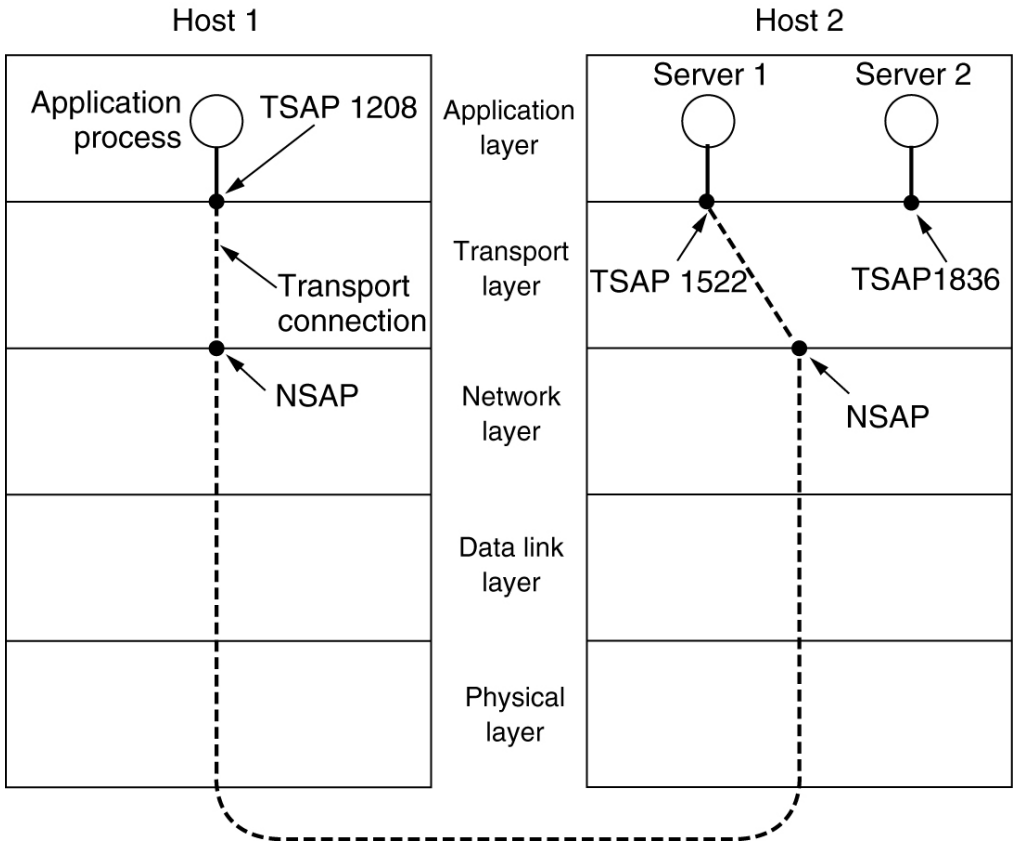
- Jitter - delay fluctuations, very critical for most real-time applications

- Round-trip time (RTT) - time a messages needs to go from sender to receiver and back

More ambiguities

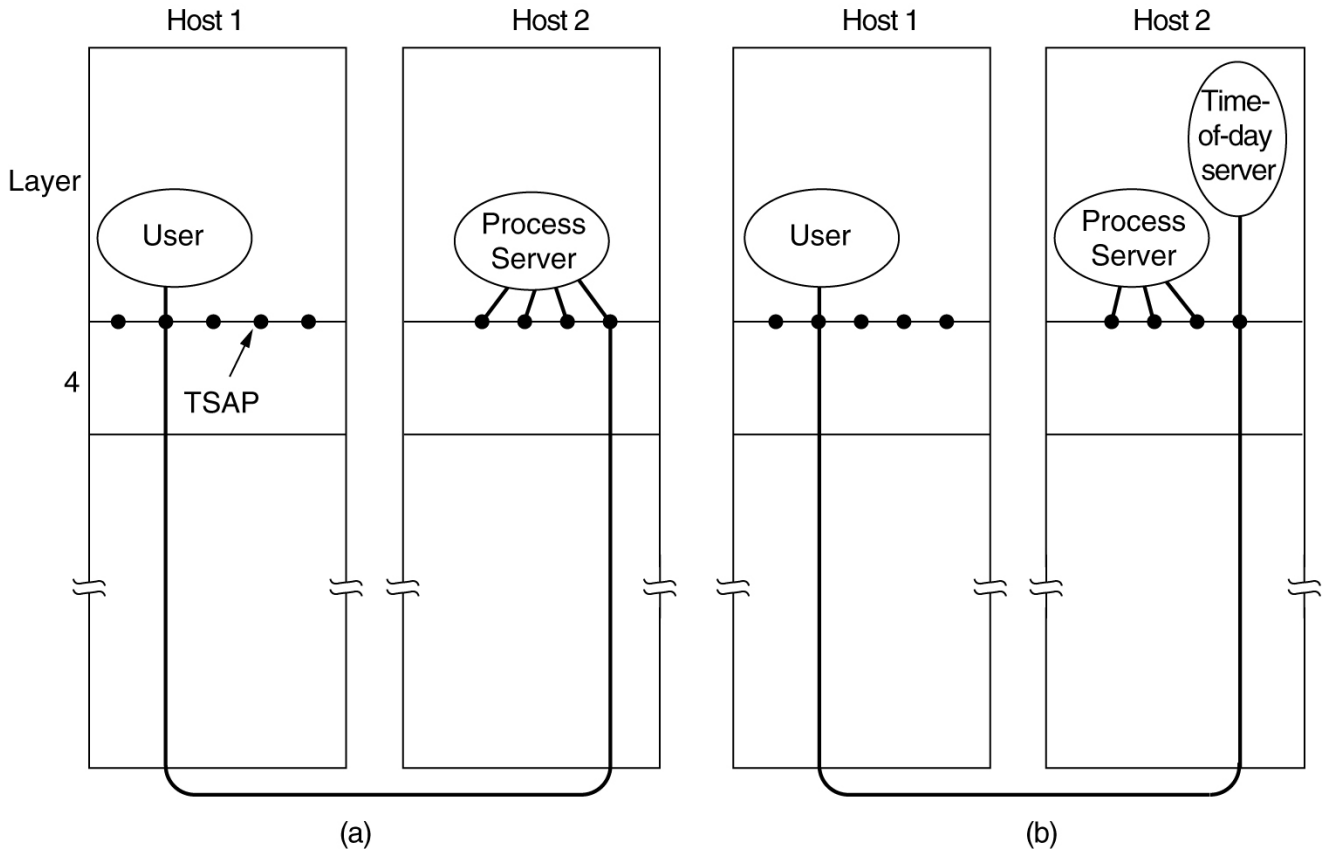
- mbit / mb / Mb / Mbit ... ?
- Latency: sometimes end2end-delay
- link: physical connection between one or more hosts or routers, or link between IP routers (may consist of multiple physical links!)
- capacity: often physical capacity, but different if you talk about TCP
- In general:
  make sure you know which layer you are talking about!

Addressing



TSAPs, NSAPs and transport connections

Connection establishment



How a user process in host 1 could establish a connection with a time-of-day server in host 2

The Internet transport layer

- Services are (mostly) defined by two protocols
  - UDP (connectionless): sends a “datagram”
  - TCP (connection oriented): transfers a reliable bytestream

- Addressing: port numbers
  - Choosing a service during connection establishment: well-known ports

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Berkeley sockets:
TCP service primitives

Focus on the Internet

- The Internet is rather important...
  - shown to be scalable, often attributed to “end-to-end argument” (simple, slightly too strict interpretation: “keep the network dumb”)
- Its transport layer includes many necessary functions
  - developed as “patches” over the years
  - TCP has grown and grown and grown... should be robust against *everything*!
  - some complementary functions inside the net
- The Internet’s design has been criticized a lot
  - especially recently: a lot of funding for “future Internet”
  - but it’s very hard to change it now

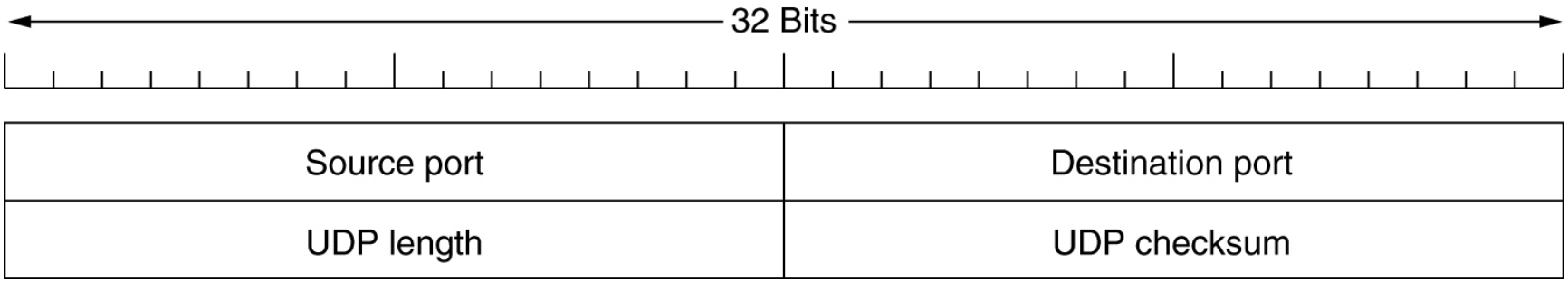
Internet terminology

- PDU, SDU, etc.: OSI terminology
  - Internet terminology: datagram, segment, packet
- Theoretically, 1 TCP segment could be split into multiple IP packets
  - hence different words used
- In practice, this is inefficient and not done
  - hence segment = packet

Speaking of packet splitting...

- (IP) fragmentation = inefficient
  - But small packets have large header overhead
- Path MTU Discovery: determine the largest packet that does not get fragmented
  - originally (RFC 1191, 1990): start large, reduce upon reception of ICMP message ➔ black hole problem if ICMP messages are filtered
  - now (RFC 4821, 2007): start small, increase as long as transport layer ACKs arrive ➔ transport protocol dependent
- Network layer function with transport layer dependencies

UDP and UDP Lite



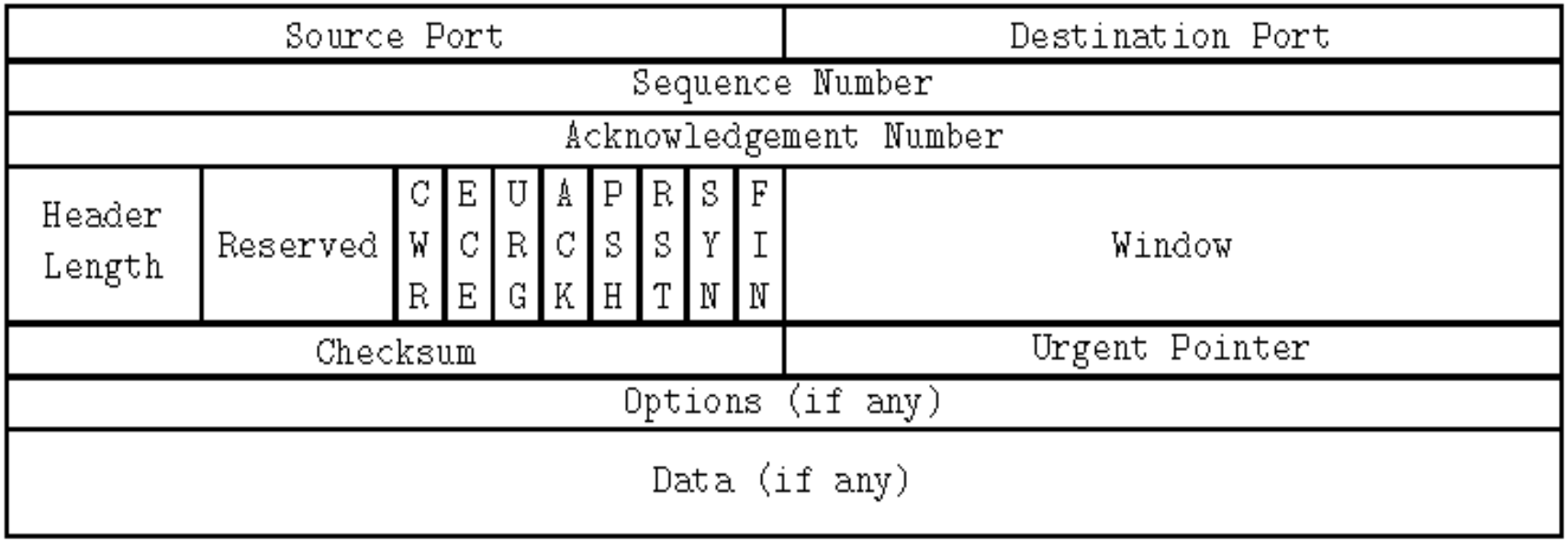
- UDP = IP + 2 features:
  - Ports: identify communicating instances with similar IP address (transport layer)
  - Checksum: Adler-32 covering the whole packet
    - checksum field = 0: no checksum at all! ➔is this useful?
    - ⇒ solution: UDP Lite (length := checksum coverage)
      - advantage: e.g. video codecs can cope with bit errors, but UDP drops whole packet
      - critical: app's depending on UDP Lite can depend on lower layers
      - usefulness: often, link layers do not hand over corrupt data
- Usage of UDP: unreliable data transmission (DNS, SNMP, real-time streams, ..)

Standard TCP

What TCP does for you (roughly)

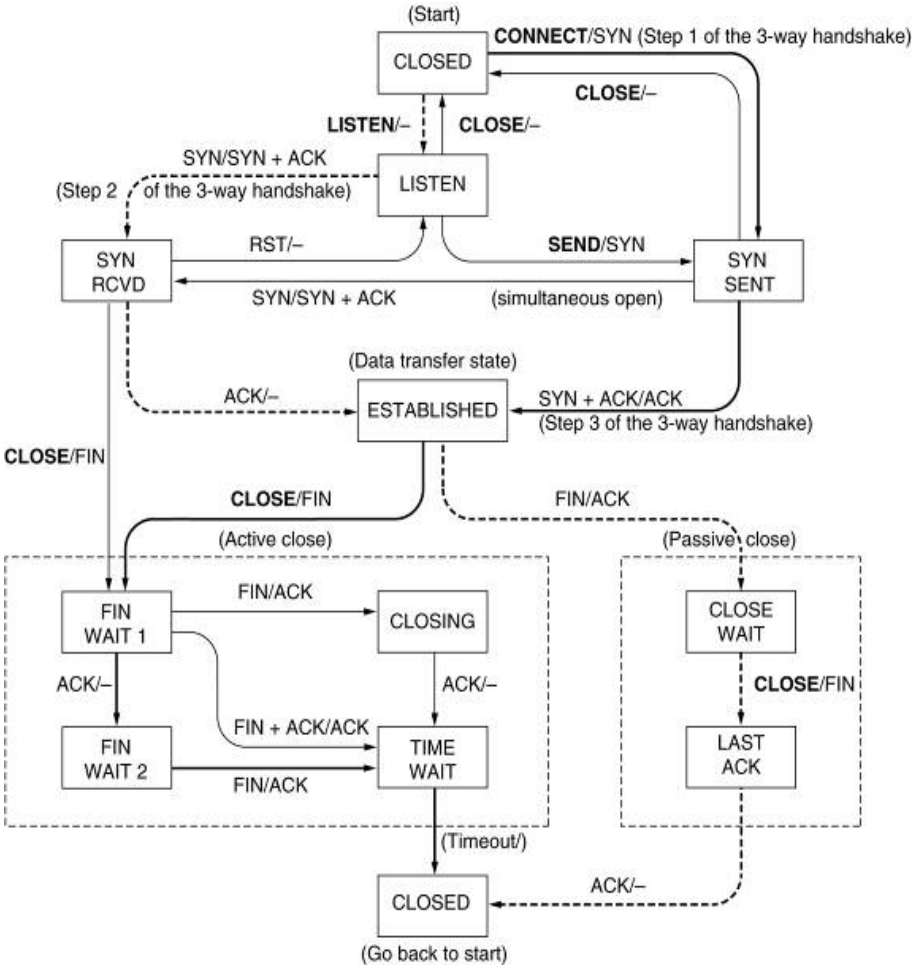
- UDP features: multiplexing + protection against corruption
  - ports, checksum
- connection handling
  - explicit establishment + teardown
- stream-based in-order delivery
  - segments are ordered according to sequence numbers
  - only consecutive bytes are delivered
- reliability
  - missing segments are detected (ACK is missing) and retransmitted
- flow control
  - receiver is protected against overload (“sliding window“ mechanism)
- congestion control
  - network is protected against overload (window based)
  - protocol tries to fill available capacity
- full-duplex communication
  - e.g., an ACK can be a data segment at the same time (piggybacking)

TCP Header



- Flags indicate connection setup/teardown, ACK, ..
- If no data: packet is just an ACK
- Window = advertised window from receiver (flow control)
  - Field size limits sending rate in today's high speed environments; solution: Window Scaling Option – both sides agree to left-shift the window value by N bit

TCP Connection Management



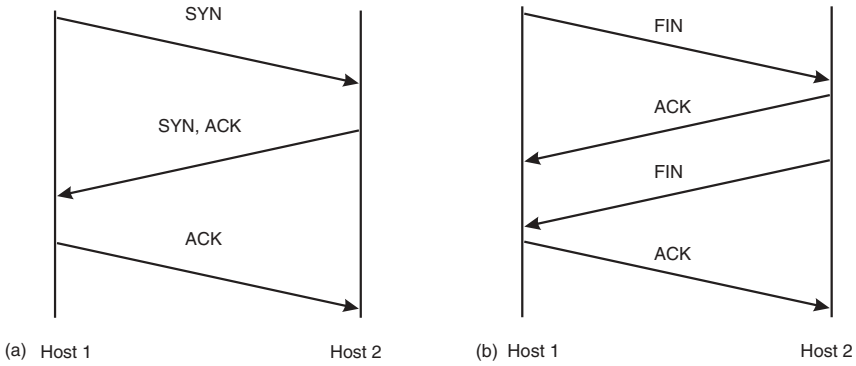
heavy solid line:
normal path for a client

heavy dashed line:
normal path for a server

Light lines:
unusual events

Connection setup

teardown



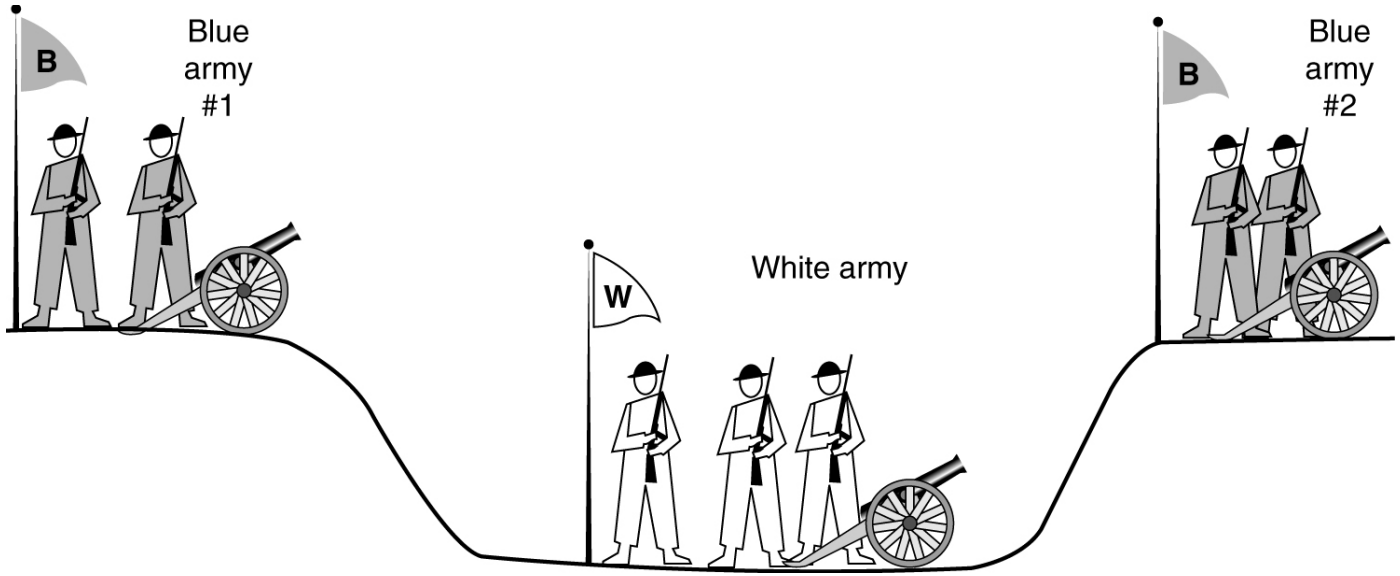
Connection establishment

- Sequence number synchronization ("SYN")
  - avoid mistaking packets that carry the same sequence number but don't belong to the intended connection

- TCP SYN sets up state ("that was the number, I sent a SYN/ACK, now I wait for a response")
  - exploited by SYN flood DoS attack
  - Solution: put state in packets ("cookie")
    - Can be implemented without changing the protocol, by encoding it in sequence numbers
    - Variant that requires changing the protocol currently (2/2012-3/2014) proposed by Google to get rid of the initial one-RTT delay

Connection release

- No way to do it without timeouts...



Error control: Acknowledgement

- ACK ("positive" Acknowledgement)
- Purposes:
  - sender: throw away copy of data held for retransmit
  - time-out cancelled
  - msg-number can be re-used
- TCP counts bytes, not segments; ACK carries "next expected byte" (#+1)
- ACKs are cumulative
  - ACK *n* acknowledges all bytes *"last one ACKed"* thru *n-1*
- ACKs should be delayed
  - TCP ACKs are unreliable: dropping one does not cause much harm
  - Enough to send only 1 ACK every 2 segments, or at least 1 ACK every 500 ms (often set to 200 ms)

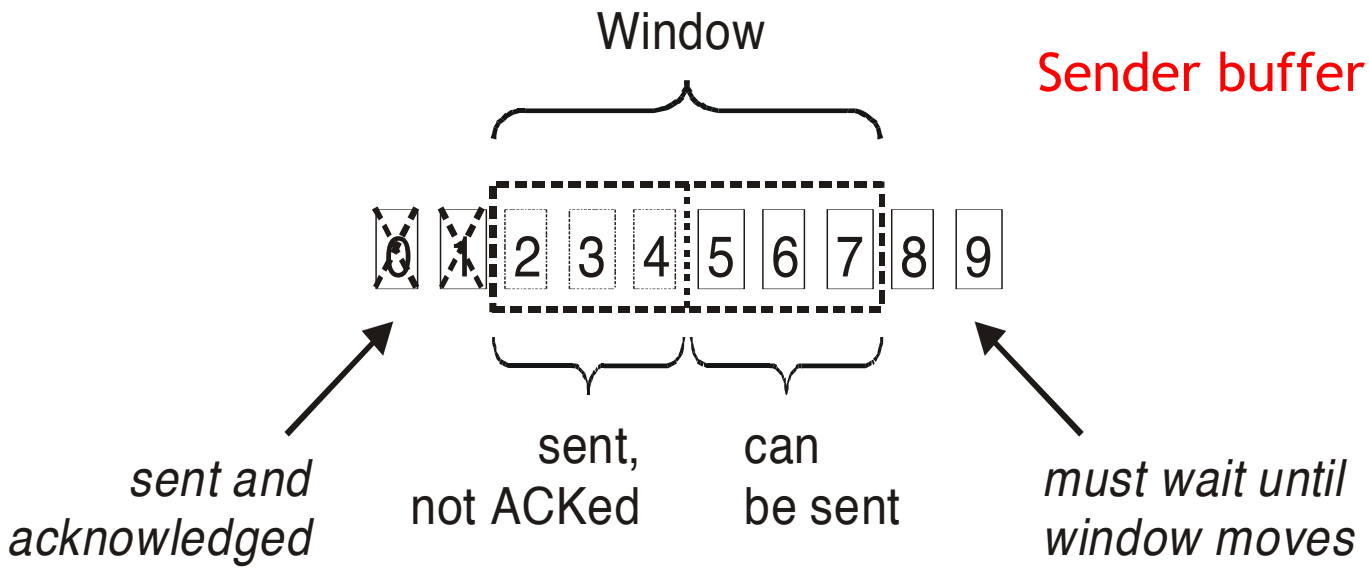
Error control: Timeout

- Go-Back-N behavior in response to timeout
- Retransmit Timeout (RTO) timer value difficult to determine:
  - too long ⇒ bad in case of msg-loss; too short ⇒ risk of false alarms
  - General consensus: too short is worse than too long; use conservative estimate
- Calculation: measure RTT (Seg# ... ACK#) , then:
  original suggestion in RFC 793: Exponentially Weighed Moving Average (EWMA)
  - SRTT = (1-α) SRTT + α RTT
  - RTO = min(UBOUND, max(LBOUND, β * SRTT))
- Depending on variation, result may be too small or too large; thus, final algorithm includes variation (approximated via mean deviation)
  - SRTT = (1-α) SRTT + α RTT
  - δ = (1 - β) * δ + β * [SRTT - RTT]
  - RTO = SRTT + 4 * δ

RTO calculation

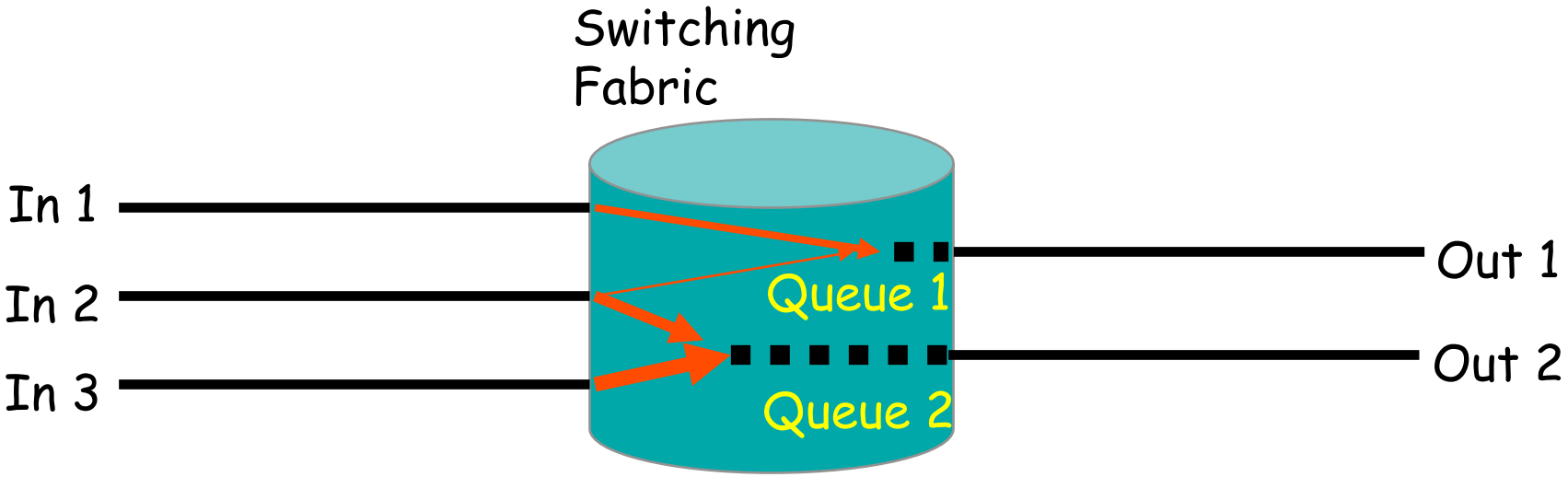
- Problem: retransmission ambiguity
  - Segment #1 sent, no ACK received → segment #1 retransmitted
  - Incoming ACK #2: cannot distinguish whether original or retransmitted segment #1 was ACKed
  - Thus, cannot reliably calculate RTO!
- Solution 1 [Karn/Partridge]: ignore RTT values from retransmits
  - Problem: RTT calculation especially important when loss occurs; sampling theorem suggests that RTT samples should be taken more often
- Solution 2: Timestamps option
  - Sender writes current time into packet header (option)
  - Receiver reflects value
  - At sender, when ACK arrives, RTT = (current time) - (value carried in option)
  - Problems: additional header space; facilitates NAT detection

Window management



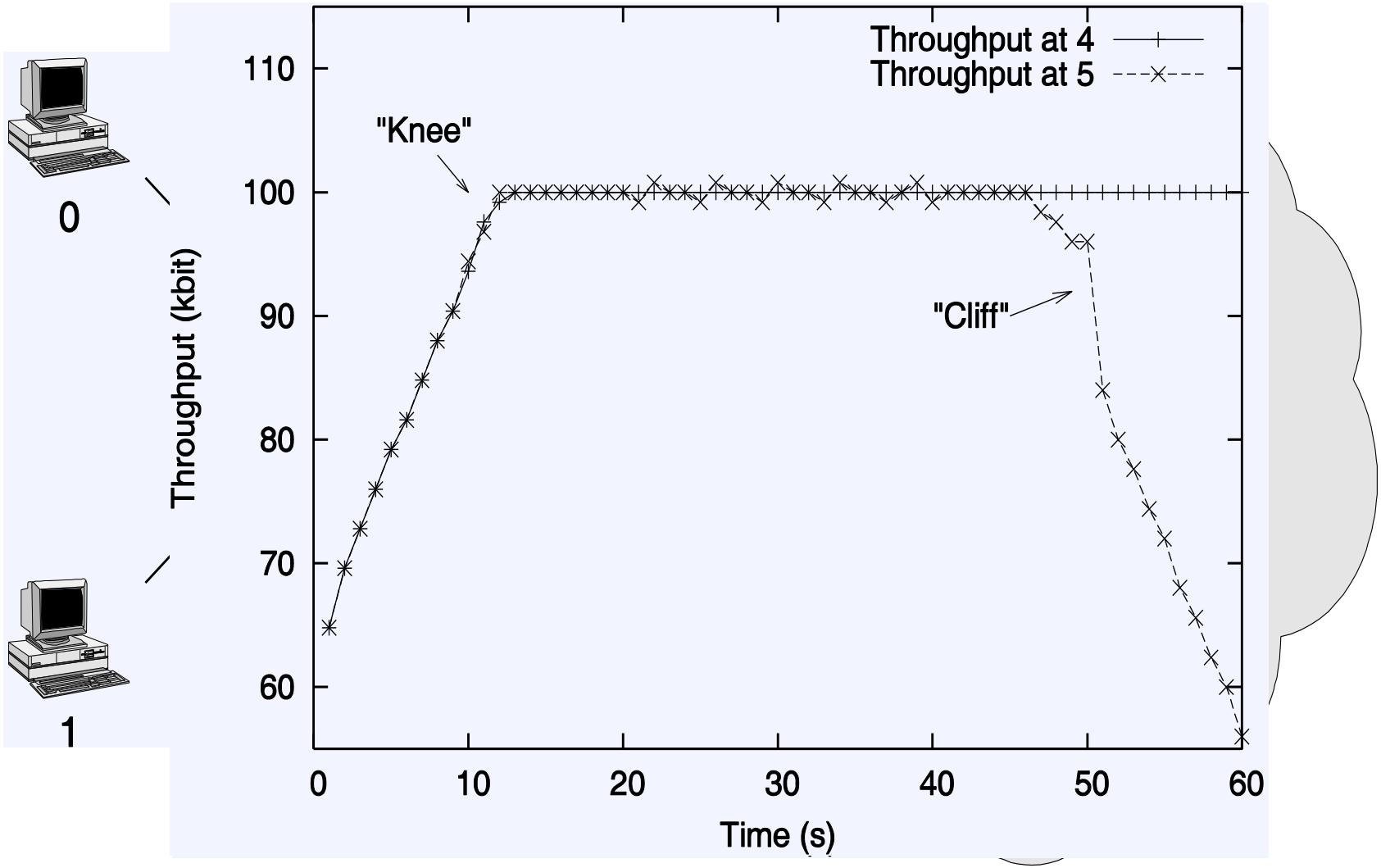
- Receiver "grants" credit (receiver window, rwnd)
  - sender restricts sent data with window
- Receiver buffer not specified
  - i.e. receiver may buffer reordered segments (i.e. with gaps)

A simple router model



- Switch(ing) fabric forwards a packet (dest. addr.)
  if no special treatment necessary: “fast path“ (hardware)

- Queues grow when traffic bursts arrive
  - low delay = small queues, low jitter = no queue fluctuations
- Packets are dropped when queues overflow (“DropTail queueing“)
  - low loss ratio = small queues


0
1
Throughput (kbit)
110
100
90
80
70
60
0
10
20
30
40
50
60
Time (s)
Throughput at 4
Throughput at 5
"Knee"
"Cliff"

Global congestion collapse

Craig Partridge, Research Director for the Internet Research Department at BBN Technologies:

Bits of the network would fade in and out, but usually only for TCP. You could ping. You could get a UDP packet through. Telnet and FTP would fail after a while. And it depended on where you were going (some hosts were just fine, others flaky) and time of day (I did a lot of work on weekends in the late 1980s and the network was wonderfully free then).

Around 1pm was bad (I was on the East Coast of the US and you could tell when those pesky folks on the West Coast decided to start work...).

Another experience was that things broke in unexpected ways - we spent a lot of time making sure applications were bullet-proof against failures. (..)

Finally, I remember being startled when Van Jacobson first described how truly awful network performance was in parts of the Berkeley campus. It was far worse than I was generally seeing. In some sense, I felt we were lucky that the really bad stuff hit just where Van was there to see it.

Internet congestion control: history

- **around 1986:** first congestion collapse
- **1988:** "Congestion Avoidance and Control" (Jacobson)
  Combined congestion/flow control for TCP
  (also: variation change to RTO calculation algorithm)

- Idea: packet loss = congestion, so throttle the rate; increase otherwise

- Goal: stability - in equilibrum, no packet is sent into the network until an old packet leaves
  - ack clocking, “conservation of packets“ principle
  - made possible through window based stop+go - behaviour

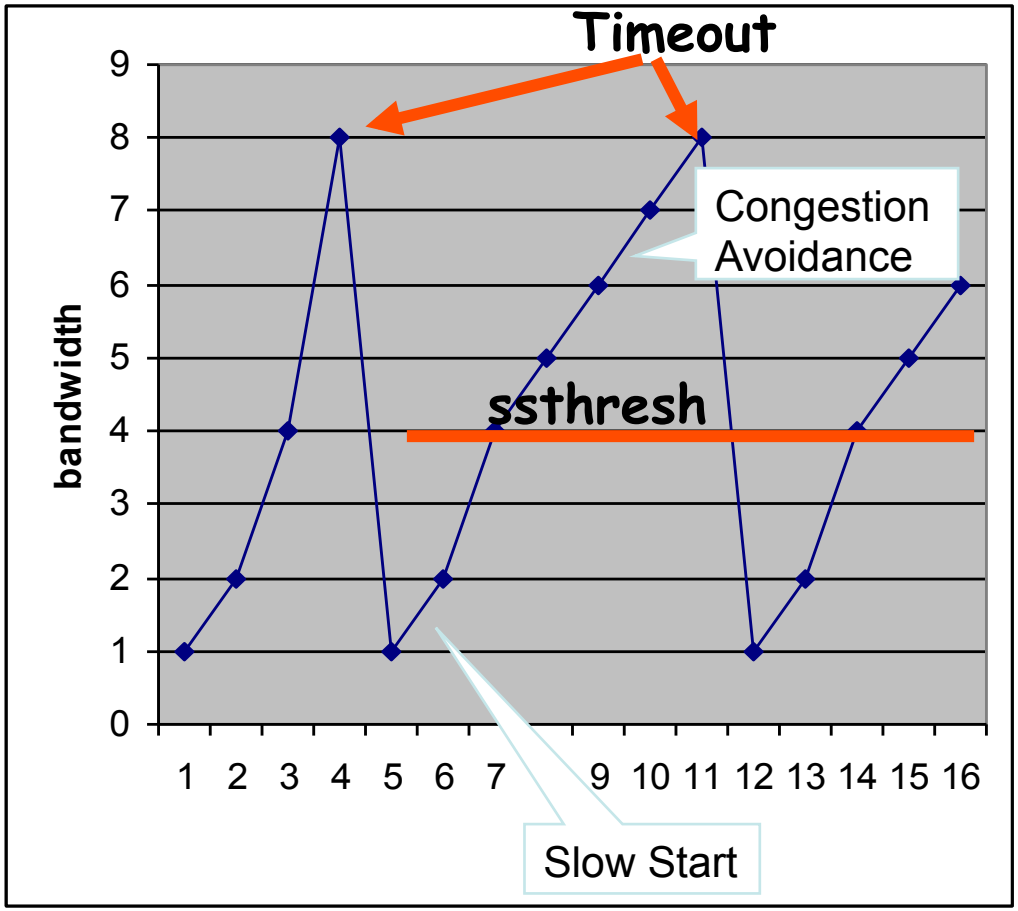
- Superposition of stable systems = stable →
  network based on TCP with congestion control = stable

TCP Congestion Control: Tahoe

- Distinguish:
  - flow control: protect receiver against overload
    (receiver "grants" a certain amount of data ("receiver window" (rwnd)))
  - congestion control: protect network against overload
    ("congestion window" (cwnd) limits the rate: min(cwnd,rwnd) used!)

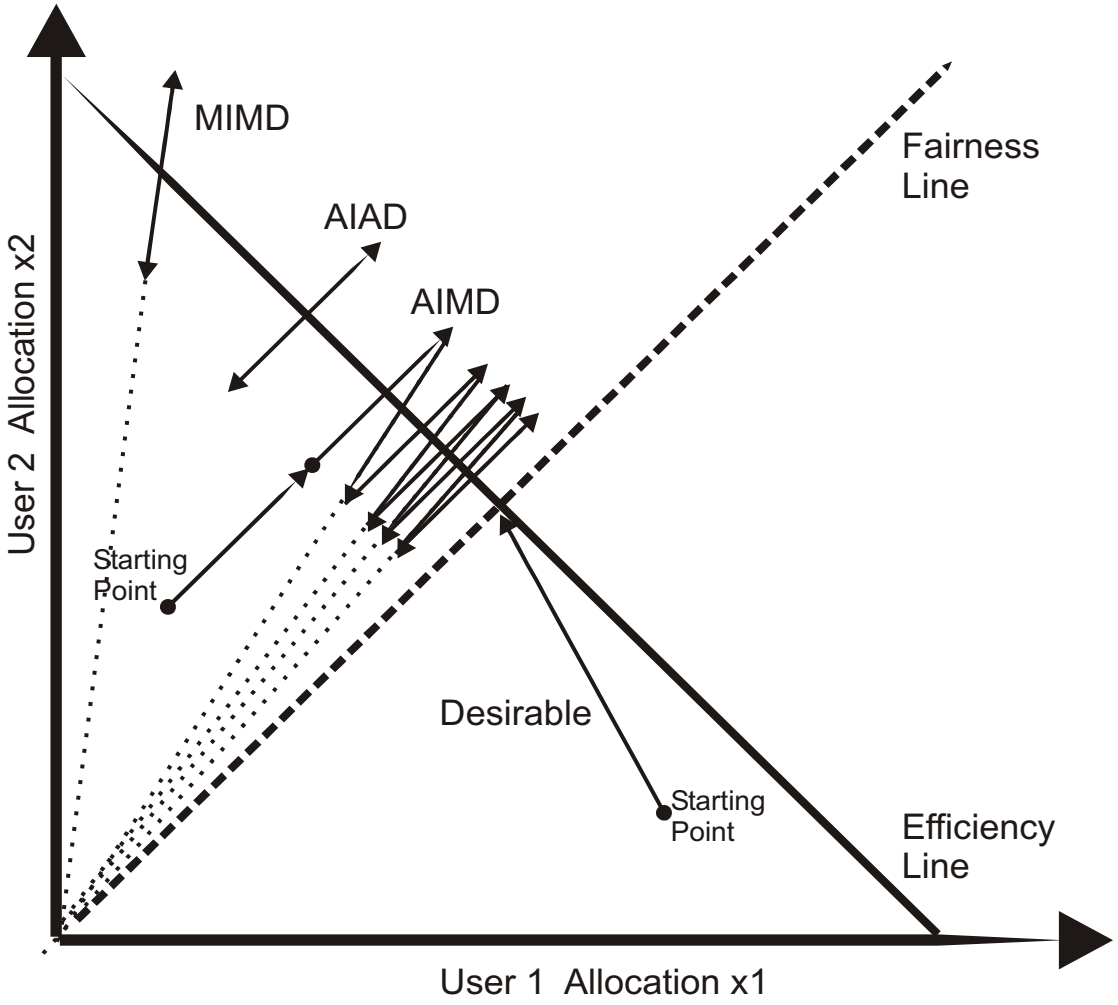
- Flow/Congestion Control combined in TCP. Two basic algorithms:
  - Slow Start: for each ack received, increase cwnd by 1 packet (exponential growth) until cwnd >= ssthresh
  - Congestion Avoidance: each RTT, increase cwnd by at most 1 packet (linear growth - "additive increase“)

TCP Congestion Control: Tahoe /2



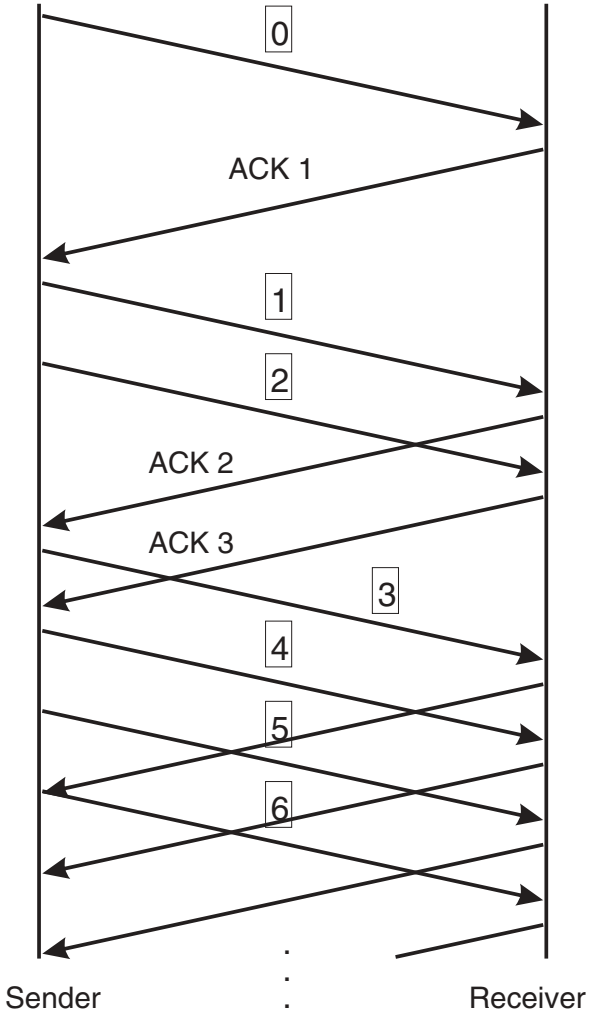
- If a packet or ack is lost (timeout), set cwnd = 1, ssthresh = cwnd / 2 ("multiplicative decrease") - exponential backoff
- *Actually, "Flightsize/2" instead of cwnd/2 because cwnd might not always be fully used*

Background: AIMD



Connection startup

- Slow start: 3 RTTs for 3 packets = inefficient for very short transfers
- Example: HTTP Requests
- Thus, initial window
  IW = min(4*MSS, max(2*MSS, 4380 byte))
  - why these values?
  - worked well a long time ago; recently increased to 10
  - Adopted in Linux as default since kernel 2.6.39 (May 2011)



Fast Retransmit / Fast Recovery (Reno)

Reasoning: slow start = restart; assume that network is empty

But even similar incoming ACKs indicate that packets arrive at the receiver!

Thus, slow start reaction = too conservative.

1. Upon reception of third duplicate ACK (DupACK): ssthresh = FlightSize/2

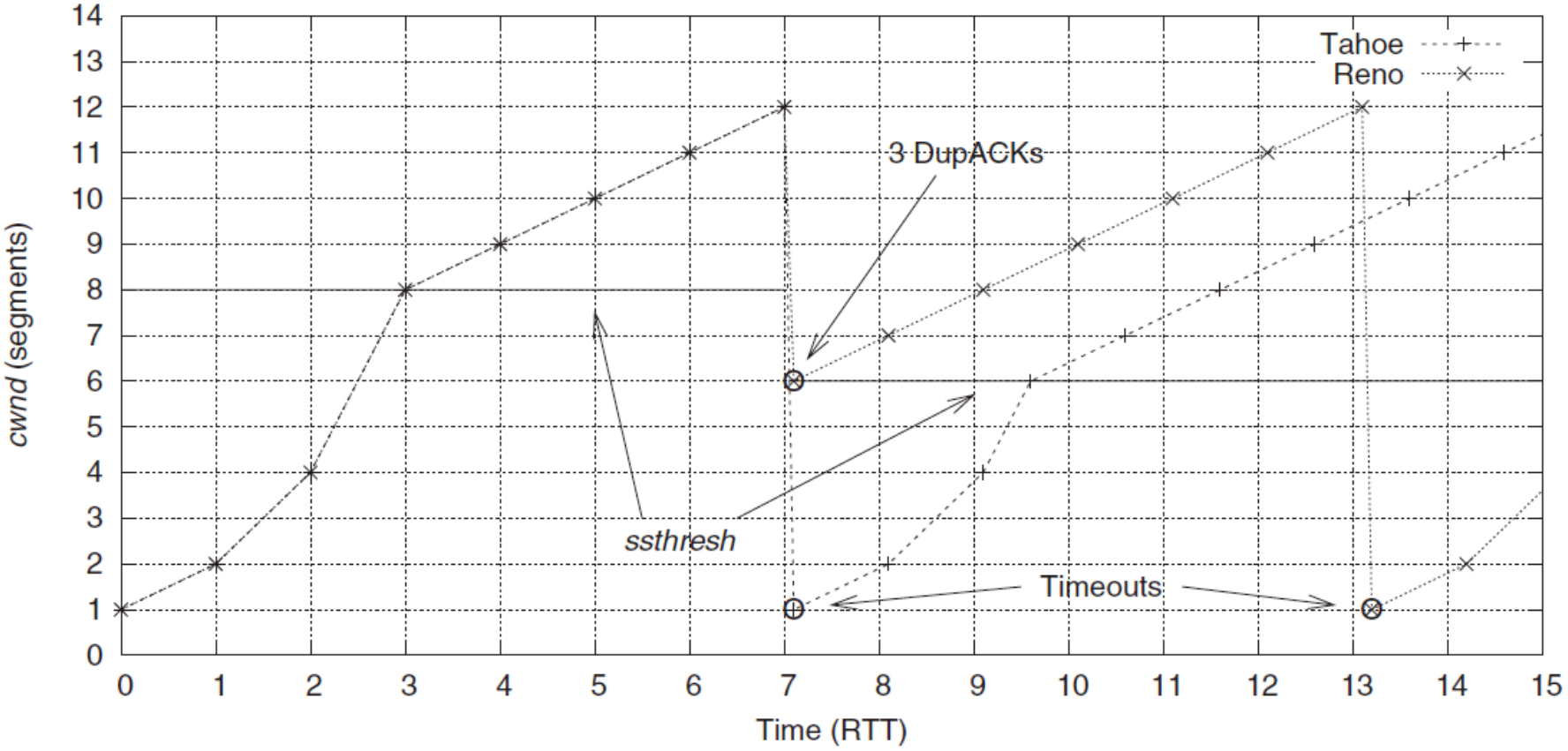
2. Retransmit lost segment (fast retransmit);
   cwnd = ssthresh + 3*SMSS
   ("inflates" cwnd by the number of segments (three) that have left the network and which the receiver has buffered)

3. For each additional DupACK received: cwnd += SMSS
   (inflates cwnd to reflect the additional segment that has left the network)

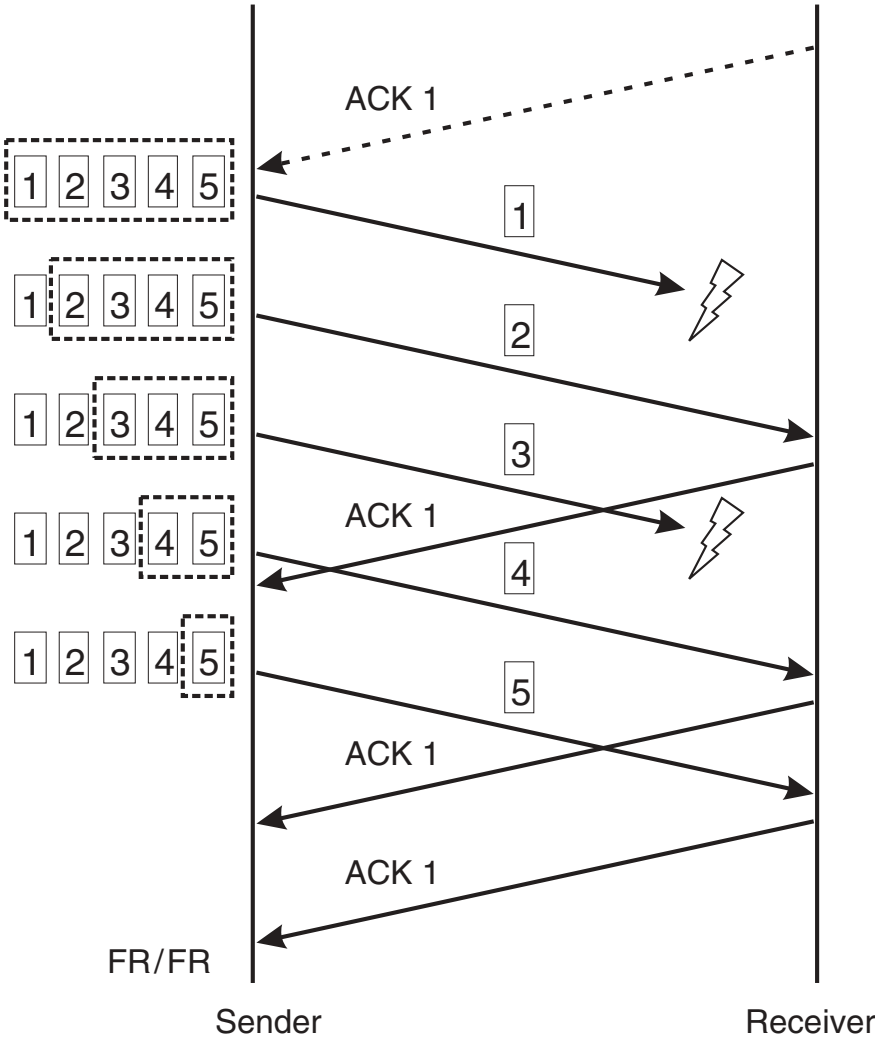
4. Transmit a segment, if allowed by the new value of cwnd and rwnd

5. Upon reception of ACK that acknowledges new data (“full ACK“):
   "deflate" window: cwnd = ssthresh (the value set in step 1)

Tahoe vs. Reno

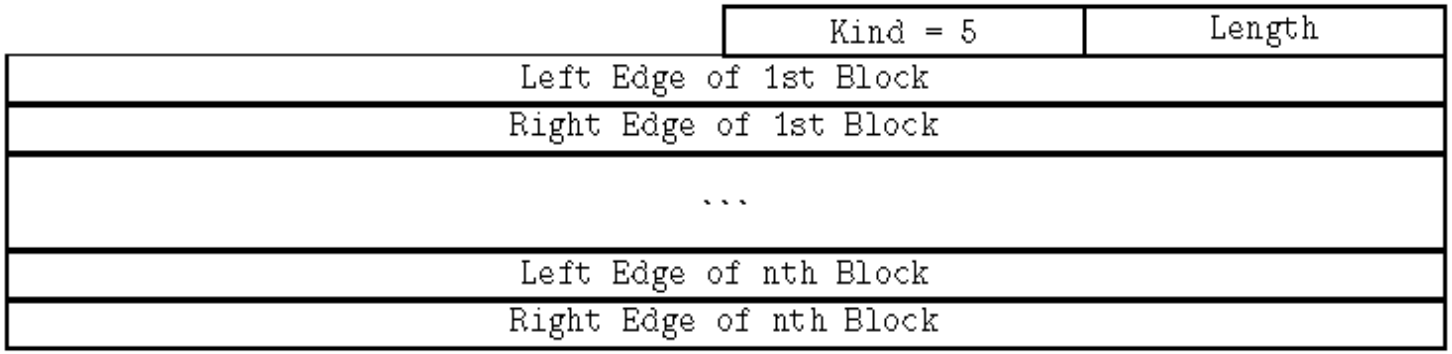


Multiple dropped segments



- Sender cannot detect loss of multiple segments from a single window
- Insufficient information in DupACKs
- NewReno:
  - stay in FR/FR when partial ACK arrives after DupACKs
  - retransmit single segment
  - only full ACK ends process
- Important to obtain enough ACKs to avoid timeout
  - Limited transmit: also send new segment for first two DupACKs

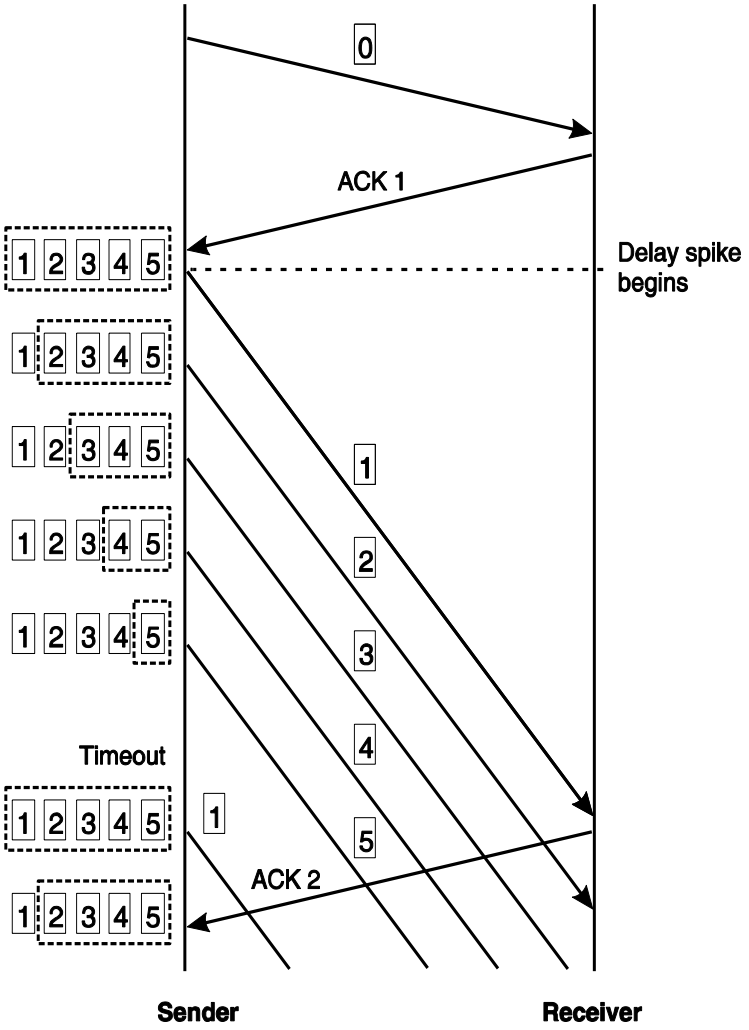
Selective ACKnowledgements (SACK)



- Example on NewReno slide: send ACK 1, SACK 3, SACK 5 in response to segment #4
- Better sender reaction possible
  - Reno and NewReno can only retransmit a single segment per window
  - SACK can retransmit more (RFC 3517 – maintain scoreboard, pipe variable)
  - Particularly advantageous when window is large (long fat pipes)
- but: requires receiver code change
- Extension: DSACK informs the sender of duplicate arrivals

Spurious timeouts

- Possible occurrence in e.g. wireless scenarios (handover): sudden delay spike
- Can lead to timeout
  → slow start
  - But: underlying assumption: "pipe empty" is wrong! ("spurious timeout")
  - Old incoming ACK after timeout should be used to undo the error
- Several methods proposed
  Examples:
  - Eifel Algorithm: use timestamps option to check: timestamp in ACK < time of timeout?
  - DSACK: duplicate arrived
  - F-RTO: check for ACKs that shouldn't arrive after Slow Start



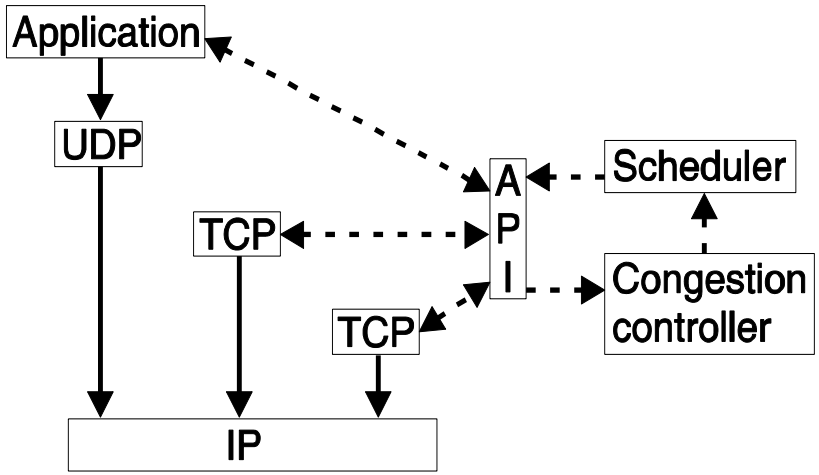
Appropriate Byte Counting

- Increasing in Congestion Avoidance mode: common implementation (e.g. Jan'05 FreeBSD code): cwnd += SMSS*SMSS/cwnd for every ACK (same as cwnd += 1/cwnd if we count segments)
  - Problem: e.g. cwnd = 2: 2 + 1/2 + 1/ (2+1/2)) = 2+0.5+0.4 = 2.9
    thus, cannot send a new packet after 1 RTT
  - Worse with delayed ACKs (cwnd = 2.5)
  - Even worse with ACKs for less than 1 segment (consider 1000 1-byte ACKs)
    → too aggressive!

- Solution: Appropriate Byte Counting (ABC)
  - Maintain bytes_acked variable; send segment when threshold exceeded
  - Works in Congestion Avoidance; but what about Slow Start?
    - Here, ABC + delayed ACKs means that the rate increases in 2*SMSS steps
    - If a series of ACKs are dropped, this could be a significant burst ("micro-burstiness"); thus, limit of 2*SMSS per ACK recommended

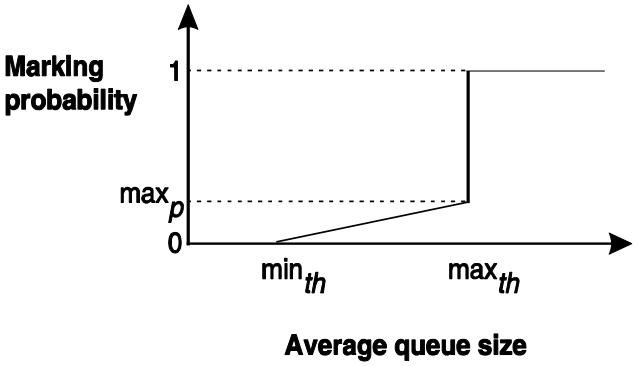
Maintaining congestion state

- TCP Control Block (TCB): information such as RTO, scoreboard, cwnd, ..
- Related to network path, yet separately stored per TCP connection
  - Compare: layering problem of PMTU storage
- TCB interdependence: affects initialization phase
  - Temporal sharing: learn from previous connection (e.g. for consecutive HTTP requests)
  - Ensemble sharing: learn from existing connections here, some information should change - e.g. cwnd should be cwnd/n, n = number of connections; but less aggressive than "old" implementation
- Congestion Manager
  - One entity in the OS maintains all the
  - congestion control related state
  - Used by TCP's and UDP based applications
  - Hard to implement, not really used

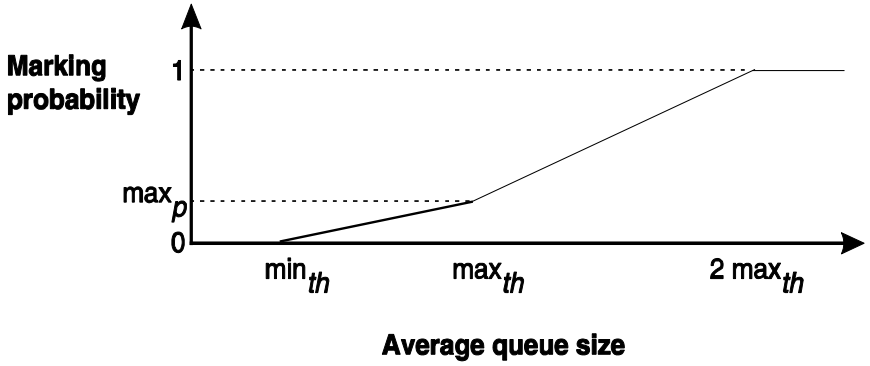


Active Queue Management

- Monitor queue, not only drop upon overflow ⇒ more intelligent decisions
  - Qavg = (1 - Wq) x Qavg + Qinst x Wq
    (Qavg = average occupancy, Qinst = instantaneous occupancy,
    Wq = weight - hard to tune, determines how aggressive RED behaves)
- Goals: keep average queue low, eliminate phase effects, manage fairness, ("punish" flows that are too aggressive)
  - Aggressive flows have more packets in the queue; thus, dropping a random one is more likely to affect such flows
  - Also possible to differentiate traffic via drop function(s)
- Explicit Congestion Notification (ECN): instead of dropping, set a bit

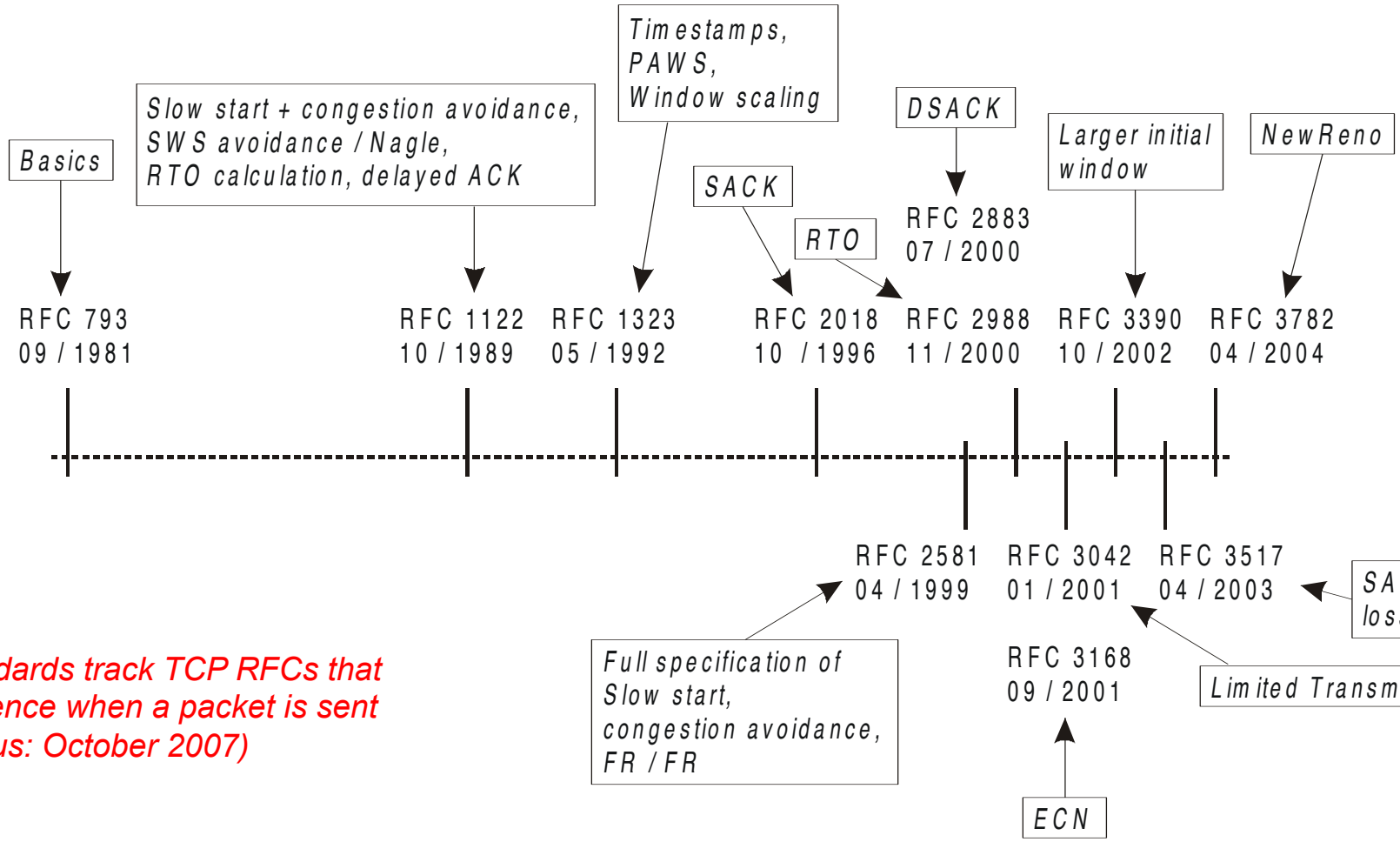


RED



RED in "gentle" mode

TCP History



Standards track TCP RFCs that influence when a packet is sent (status: October 2007)

TCP …beyond the standard

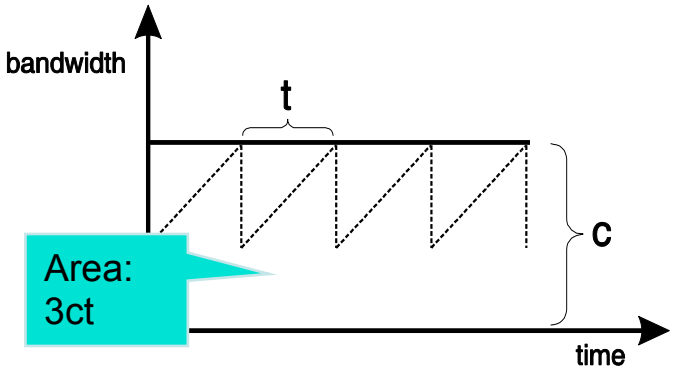
TCP with High-Speed links

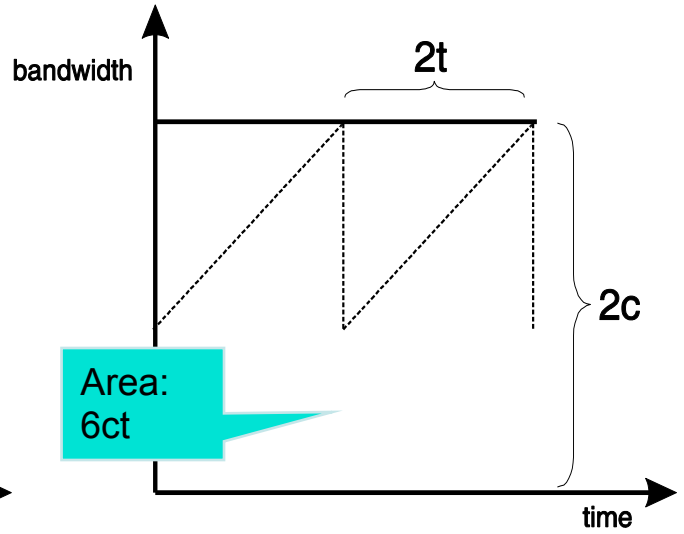
- TCP over “long fat pipes“: large bandwidth*delay product
  - long time to reach equilibrium, MD = problematic
  - from RFC 3649 (HighSpeed RFC, Experimental):

    For example, for a Standard TCP connection with 1500-byte packets and a 100 ms round-trip time, achieving a steady-state throughput of 10 Gbps would require an average congestion window of 83,333 segments, and a packet drop rate of at most one congestion event every 5,000,000,000 packets (or equivalently, at most one congestion event every 1 2/3 hours). This is widely acknowledged as an unrealistic constraint.

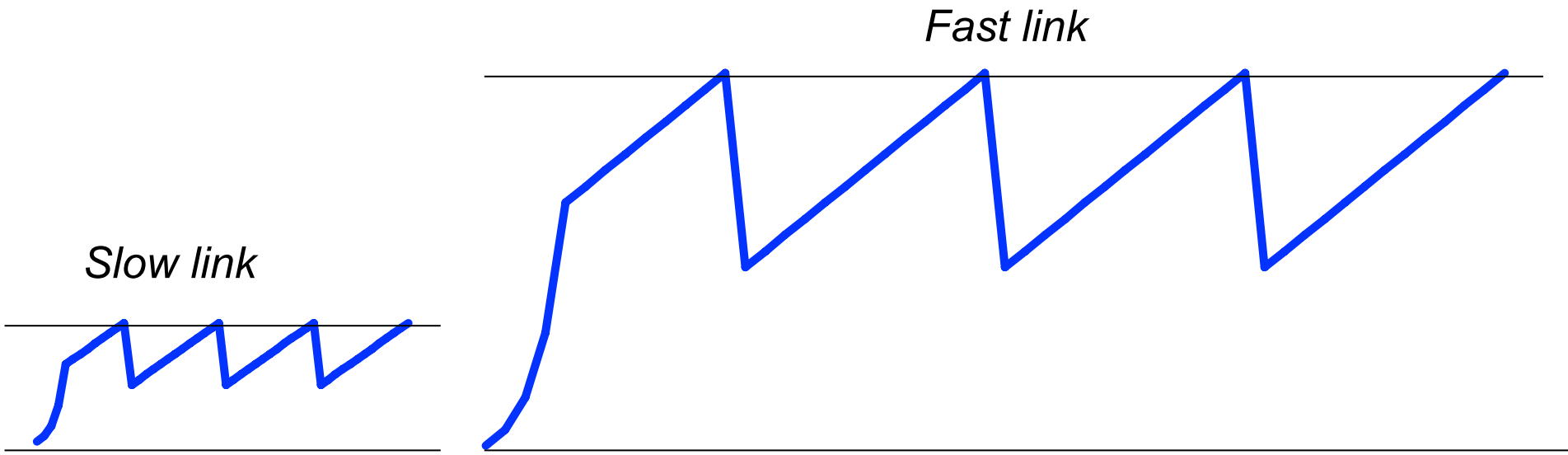
Theoretically, utilization independent of capacity

But: longer convergence time



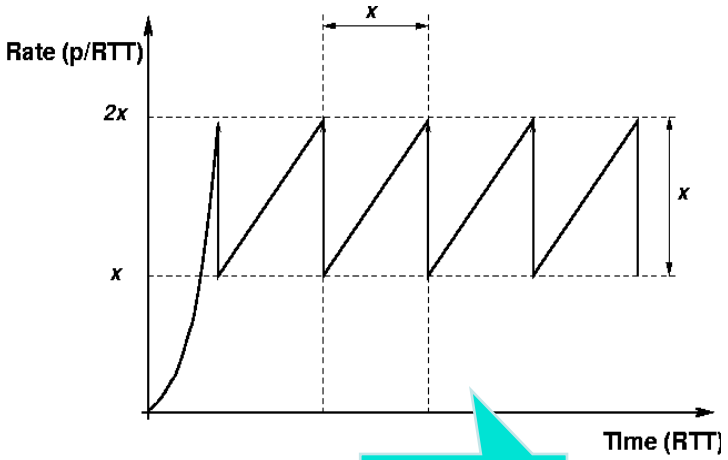


Slow convergence animation

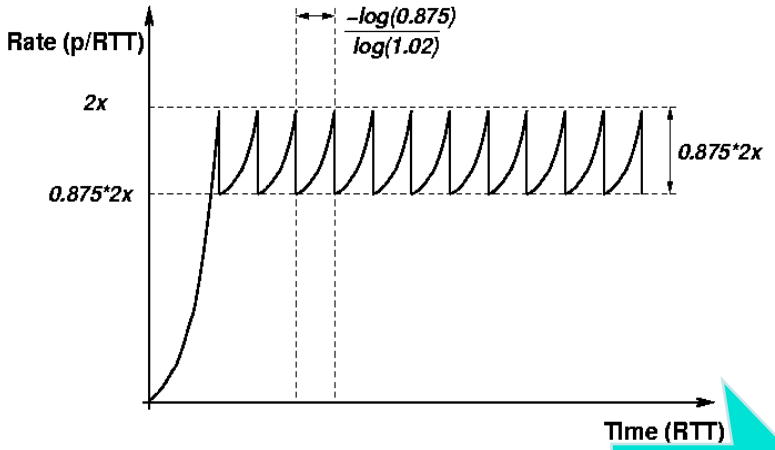


Proposed solutions

- Standards: larger initial window / window scaling option, TCP SACK
- Scalable TCP: increase/decrease functions changed
  - cwnd := cwnd + 0.01 for each ack received while not in loss recovery
  - cwnd := 0.875 * cwnd on each loss event
    (probing times proportional to rtt but not rate)



Standard TCP



Scalabe TCP

Source: http://www.deneholme.net/tom/scalable/

Proposed solutions /2

Rate	Standard TCP recovery time	Scalable TCP recovery time
1Mbps	1.7s	2.7s
10Mbps	17s	2.7s
100Mbps	2mins	2.7s
1Gbps	28mins	2.7s
10Gbps	4hrs 43mins	2.7s

- HighSpeed TCP (RFC 3649 includes Scalable TCP discussion):
  - response function includes a(cwnd) and b(cwnd), which also depend on loss ratio
  - less drastic in high bandwidth environments with little loss ***only***
  - **Significant step!**
  - Previously, either TCP-friendly or better-than-TCP; no combinations!

- TCP Westwood+
  - different congestion response function (proportional to rate instead of $\beta = 1/2$)
  - Proven to be stable, tested in real life experiments, available in your Linux

Proposed solutions /3

- FAST TCP
  - Variant based on window and delay
  - Delay allows for earlier adaptation (awareness of growing queue)
  - Proven to be stable
  - Commercially announced + patent protected, by Steven Low's CalTech group
  - another delay-based example: TCP Vegas
    - Vegas = impractical because less aggressive than standard TCP

- BIC, CUBIC
  - BIC (Binary InCrease TCP) uses binary search to find the ideal window size:
    - when loss occurs, current window = max, new window = min
    - check midpoint;
    - if no loss ⇒ new min, increase; else new window = new max
  - CUBIC = BIC++ using cubic function; growth does not depend on RTT

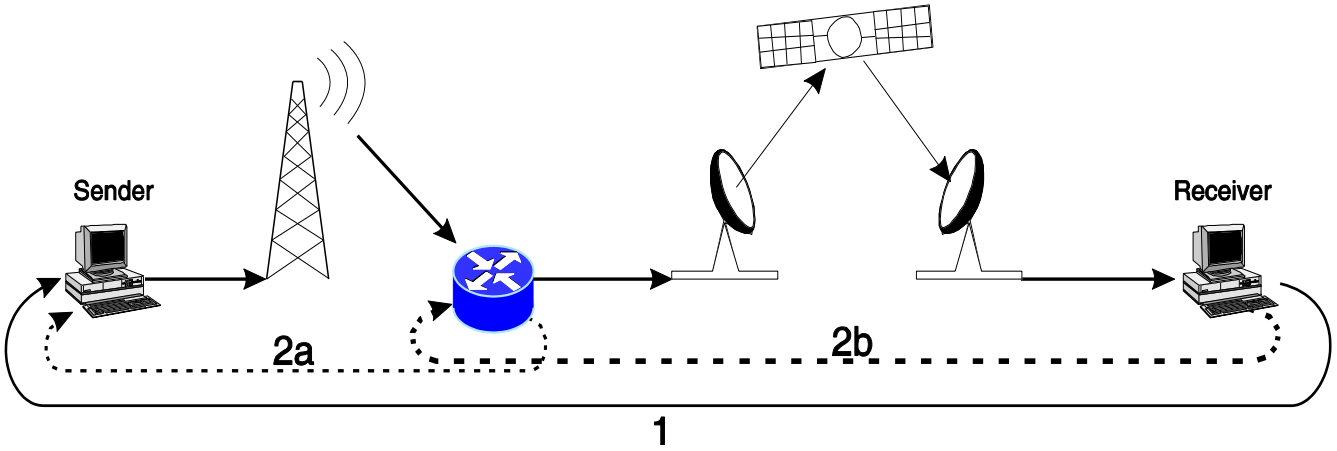
High-speed TCP reality check

- After major press release (Slashdot: “BIC-TCP 6000 times quicker than DSL“), BIC became default TCP CC. in Linux in mid-2004
  - Later replaced with CUBIC
- Compound-TCP (CTCP) = default TCP CC. in Windows Server 2008
  - For testing purposes; disabled by default in standard release
- How do these protocols interact?
  - Standards desirable
- Process devised to evaluate proposals in IETF with pre-evaluation in IRTF Internet Congestion Control Research Group (ICCRG)
  - CTCP and CUBIC proposals on the table, but activity stopped

TCP over Satellite and PEPs

- Satellites combine several problems
  - Long delay
  - High capacity
  - Wireless (but usually not noisy (for TCP) because of link layer FEC)
  - Can be asymmetric (e.g. direct satellite downlink, 56k modem uplink)

- Thus, TCP over satellite is a major research topic
  - Transparent improvements ("Performance Enhancing Proxies") common
  - Figure: split connection approach: 2a / 2b instead of control loop 1
  - Many possibilities - e.g. Snoop TCP: monitor + buffer; in case of loss, suppress DupACKs and retransmit from local buffer



Pacing

- "Micro burstiness" can lead to packet drops
- Generally, packet gap dictated by bottleneck link; but incoming stream at bottleneck can be bursty (e.g. from slow start)
  - Put the "pacing device" (PEP) close to bottleneck
- Pacing is hard at high speeds (clock granularity)
- Various solutions
  - e.g. "gap frames" that are later dropped by a link layer device
  - Burst control mechanisms in Linux