Advanced Networking

TCP

Renato Lo Cigno
Renato.LoCigno@disi.unitn.it

Content

- Some details on window protocols
- TCP hearders and formats
- TCP Options
- TCP flow control
- TCP Congestion control (most bulky!)

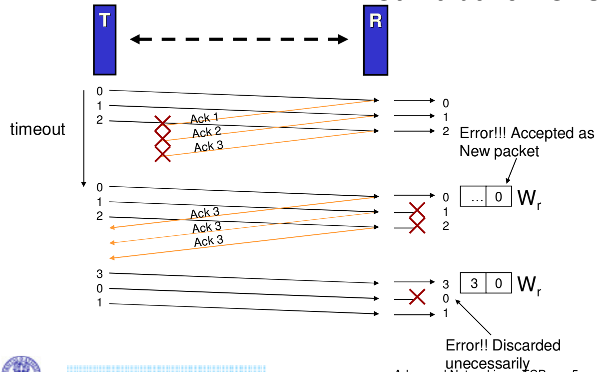
Basic Selective Repeat

- Requires 1 ACK per packet
- Positive ACK if the packet is received in order or it is received out-of-order
- Negative ACK if the packet is missing
  - Problem: lost ACKs block the protocol
- Implicit negative ACK by repeating the ACK of the last in-order packet
- Transmitter builds a local copy of the receiver window and retransmit only lost packets
- Same effect can be obtained with cumulative ACKs, with the limit of recovering 1pkt per RTT

Window relations in SR

- W = size of the counting space (bytes, packets, …)
- Wt = Transmitter window size
- Wr = Receiver window size
- Must be Wt+Wr < W to ensure working correctly
- Relation holds for both cumulative and selective ACKs

Example: W=4, Wt=3, Wr=2 Cumulative ACKs



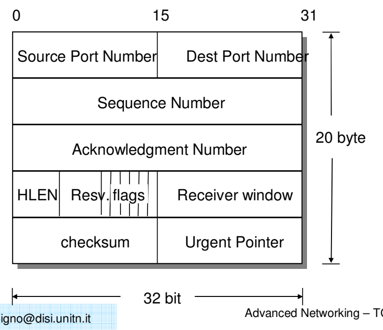
TCP: Bibliography

- Richard Stevens: TCP/IP Illustrated, Vol.1: The Protocols, 1994, Addison Wesley
- William Stallings: Data and Computer Communications, 8/Ed. Prentice Hall
- RFC 793 (1981)
  - Transmission Control Protocol
- RFC 1122/1123: (1989)
  - Requirements for Internet Hosts
- RFC 1323: (1992)
  - TCP Extensions for High Performance
- RFC 2018: (1996)
  - TCP Selective Acknowledgment Options

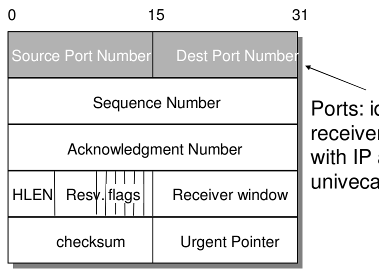
TCP: riferimenti bibliografici

- RFC 2581:
  - TCP Congestion Control (PRP STD)
- RFC 2582:
  - The NewReno Modification to TCP's Fast Recovery Algorithm
- RFC 2883:
  - An Extension to the Selective Acknowledgement (SACK) Option for TCP
- RFC 2988:
  - Computing TCP's Retransmission Timer
- ....

TCP header (no options)

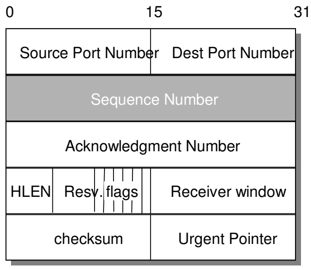


TCP header



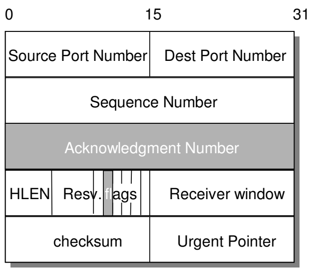
Ports: identify sender and receiver processe, together with IP addresses identify univecally a connection

TCP header



- Identify the position of the first payload byte within the stream of data
- Idependent for the two directions of the connection
- The sender decides it at the beginning of the connection with the SYN packet

TCP header



Seq. number + payload + 1 of the last packet received correctly and in order

Defines the NEXT byte the receiver expects

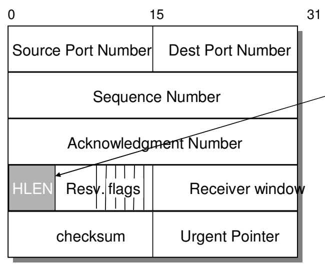
Make sense only if the ACK flag is set.

Seq. and ACK

- Numbering on 32 bit
- As a function of link speed we have different wrapping times ...
- The same application may have problems if the sequence wrap arounds or if successive connections have overlapping sequences

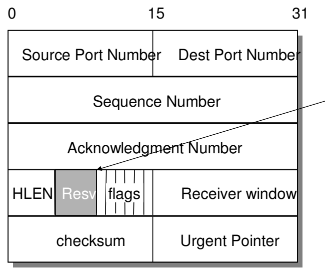
Network speed		Wrap Around Time	
T1	(1.5Mbps)	6.4	h
Ethernet	(10Mbps)	57	m
T3	(45Mbps)	13	m
FastEth	(100Mbps)	6	m
STS-3	(155Mbps)	4	m
STS-12	(622Mbps)	55	s
STS-24	(1.2Gbps)	28	s

TCP header



Header length in 32 bit words, needed with options

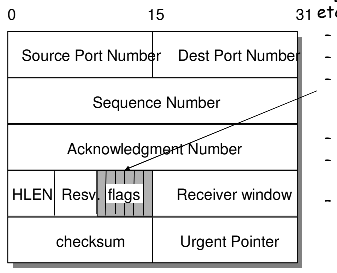
TCP header



Not yet used, but reserved

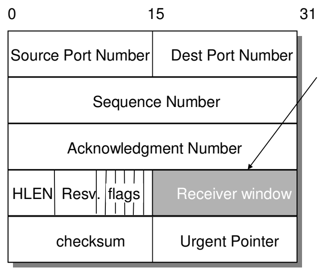
TCP header

- Connection management



- Six bits of flags, can be set together (es. ACK+SYN etc.):
  - URG: Urgent pointer is valid
  - ACK: ACKing data
  - PSH: Forces the data passage to the application (e.g. telnet)
  - RST: reset connection
  - SYN: synchronize seq. No. in opening connection
  - FIN: closing connection

TCP Header



Number of bytes, starting and including the one in the ACK field that the receiver can accept; implements flow control. 16 bits, the maximum value for rwnd is 65535 byte, unless the *window scaling option* is enabled (more later on)

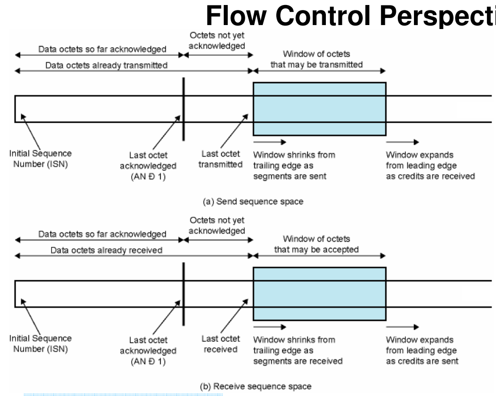
The receiver window drives throughtput

- Throughput is given by W/RTT
- Maximum data per RTT is max RWND:
  - 16-bit rwnd = 64kB max
- Given RTT=100ms the following windows are required to exploit the relative channels

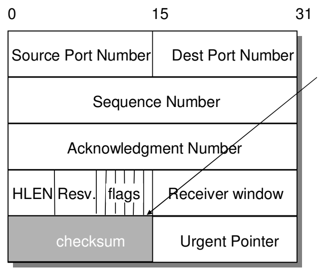
Channel (capacity)		bandwidth x delay
T1	(1.5Mbps)	18kB
Ethernet	(10Mbps)	122kB
T3	(45Mbps)	549kB
FastEth	(100Mbps)	1.2MB
STS-3	(155Mbps)	1.8MB
STS-12	(622Mbps)	7.4MB
STS-24	(1.2Gbps)	14.8MB

- These limits can be overcame using the window scale option

Sending and Receiving Flow Control Perspectives

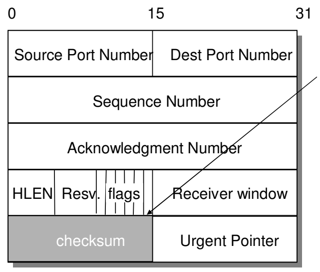


TCP header



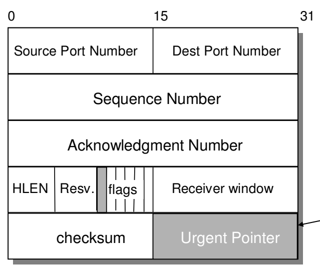
Checksum is compulsory and is computed on header and data plus the pseudo-header including IP address and protocol type.
This is a layering violation, but a useful one!

TCP header



- Checksum algorithm
  - align header, data and pseudo-header to 16 bits
  - sum every line in 1s complement algebra
  - ther result is a 32 bit number that is divided in two 16 bits parts
  - sum in 1s complement the two parts including the overflow
  - The result is the checksum inserted in the header

Intestazione TCP



It's the pointer to what is the “urgent data” in the data field (e.g. ctrl-C in a telnet session).

It's expresed as offset wrt the seq. no.

Valid only if URG is set

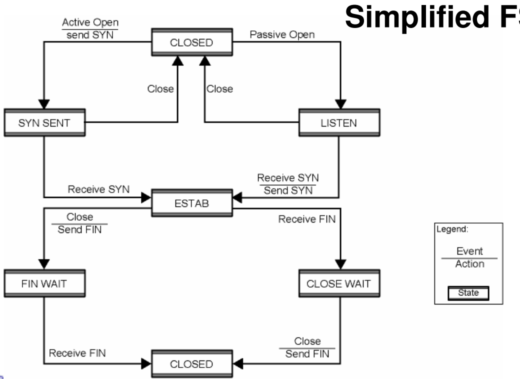
TCP options

- It's an extension to the header, used to add features to the protocol, many options exist
- Comes before data and it's in multiple of four bytes
- Most used are:
  - MSS (Maximum Segment Size), sent in the SYN segment to define the "optimal" size of segments to be received, not negotiated; default is 536 byte
  - Timestamping of packet to improve RTT calculation (more when talking about RTT estimation)
  - SACK for selective ACKs (more later on discussing congestion control)

TCP options

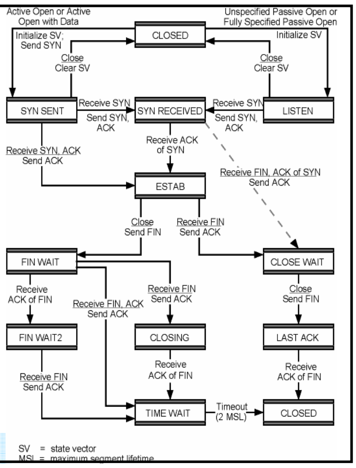
- Window scale
  - Included in SYN segment
  - Window field gives credit allocation in octets
  - With Window Scale value in Window field multiplied by 2^F
    - F is the value of window scale option
- Sack-permitted
  - Selective acknowledgement allowed
- Sack
  - Receiver can inform sender of all segments received successfully
  - Sender retransmit segments not receivedSACK, to enable
- Both must be issued for successful negotiation
- Result is not many connections use it, and usefulness still under debate

State Diagram for TCP connections Simplified FSM



TCP Entity State Diagram

Full FSM



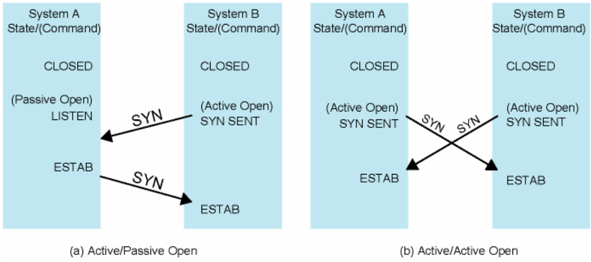
Operation with Unreliable Network Service

- Internet using IP
- …
- Segments may get lost
- Segments may arrive out of order
- … we know this all but
- **What are the consequences on a reliable transport layer?**

Problems

- Connection establishment
- Connection termination
- Ordered Delivery
- Retransmission strategy
- Duplication detection
- Flow control
- Crash recovery

Connection Establishment Scenarios



(a) Active/Passive Open

(b) Active/Active Open

Rearely used

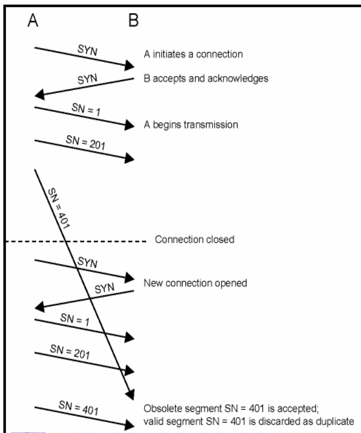
What if a Server is not listening?

- Reject with RST (Reset)
- Queue request until a matching open can be issued
  - complex
  - delay, client timeouts
- Notify the Application Protocol (AP) of pending request
  - May replace passive open with accept
  - Client must be able to "understand"

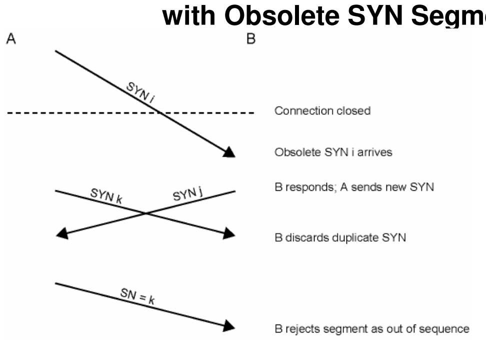
Connection Establishment

- Two way handshake … doesn't work
  - A send SYN, B replies with SYN
  - Lost SYN handled by re-transmission
    - Can lead to duplicate SYNs
  - Ignore duplicate SYNs once connected
- Lost or delayed data segments can cause connection problems
  - Segment from old connections
  - Start segment numbers far removed from previous connection
    - Use SYN i
    - Need ACK to include i
- Solved using Three Way Handshake

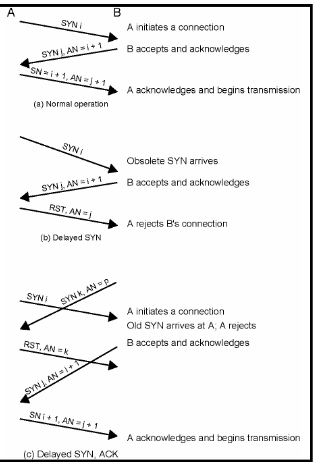
Two-Way Handshake Problem with Obsolete Data Segment



Two-Way Handshake Problem with Obsolete SYN Segments



Examples of Three-Way Handshake



Termination

- Can be from one side only of from both sides
- Abrupt termination
- By mutual agreement
- Graceful termination provided
  - Close wait state must accept incoming data until FIN received

Side Initiating Termination

- AP issue a Close request
- Transport entity sends FIN, requesting termination
- Connection placed in FIN WAIT state
  - Continue to accept data and deliver data to user
  - Not send any more data
- When FIN received, inform user and close connection

Side Not Initiating Termination

- FIN received
- Inform AP, which place connection in CLOSE WAIT state
  - Continue to accept data from AP and transmit it
- AP issues CLOSE primitive
- Transport entity sends FIN
- Connection closed

- All outstanding data is transmitted from both sides
- Both sides agree to terminate

Connection Termination

- Entity in CLOSE WAIT state sends last data segment, followed by FIN
- FIN arrives before last data segment
- Receiver accepts FIN
  - Closes connection
  - Loses last data segment
- Associate sequence number with FIN
- Receiver waits for all segments before FIN sequence number
- Loss of segments and obsolete segments
  - Must explicitly ACK FIN

Graceful Close

- Composition of the two half close
  - Send FIN i and receive AN i
  - Receive FIN j and send AN j
- Wait twice maximum expected segment lifetime
- Guarantees that all data in both directions is correctly sent
- Ensures proper freeing of logical resources on both sides

- Is slow and requires cooperation ...

Failure Recovery

- After restart all state info is lost
- Connection is half open
  - Side that did not crash still thinks it is connected
- Close connection using persistence timer
  - Wait for ACK for (time out) * (number of retries)
  - When expired, close connection and inform user
- Send RST i in response to any i segment arriving
- User must decide whether to reconnect
  - Problems with lost or duplicate data

Ordered Delivery

- Segments may arrive out of order
- Number segments sequentially
- TCP numbers each octet sequentially
- Segments are numbered by the first octet number in the segment

Retransmission Strategy

- Segment damaged in transit
- Segment fails to arrive
- Transmitter does not know of failure
- Receiver must acknowledge successful receipt
- Use cumulative acknowledgement
- Time out waiting for ACK triggers re-transmission

Timer Value

- Fixed timer
  - Based on understanding of network behavior
  - Can not adapt to changing network conditions
  - Too small leads to unnecessary re-transmissions
  - Too large and the response to lost segments is slow
  - Should be a bit longer than round trip time
- Adaptive scheme
  - May not ACK immediately
  - Can not distinguish between ACK of original segment and re-transmitted segment
  - Conditions may change suddenly

Duplication Detection

- If ACK lost, segment is re-transmitted
- Receiver must recognize duplicates
- Duplicate received prior to closing connection
  - Receiver assumes ACK lost and ACKs duplicate
  - Sender must not get confused with multiple ACKs
  - Sequence number space large enough to not cycle within maximum life of segment
- Duplicate received after closing connection
  - Discard

Flow Control

- Credit allocation
- Problem: if AN=i, W=0, the window closes and never reopens!!
- Receiver sends AN=i, W=j to reopen
  - but if this is lost the sender thinks window is closed, while the receiver thinks it is open
- Use window timer
  - If timer expires, send something
  - Could be re-transmission of previous segment

Data Transport

- Full duplex
- Timely
  - Associate timeout with data submitted for transmission
  - If data not delivered within timeout, user notified of service failure and connection abruptly terminates
- Ordered
- Labelled
  - Establish connection only if security designations match
  - If precedence levels do not match higher level used
- Flow controlled
- Error controlled
  - Simple checksum
  - Delivers data free of errors within probabilities supported by checksum

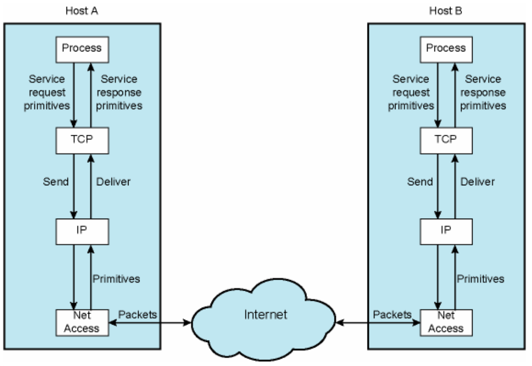
Special Capabilities

- Data stream push
  - TCP decides when enough data available to form segment
  - Push flag requires transmission of all outstanding data up to and including that labelled
  - Receiver will deliver data in same way
- Urgent data signalling
  - Tells destination user that significant or "urgent" data is in stream
  - Destination user determines appropriate action
- Error Reporting
  - TCP will report service failure due to internet conditions TCP cannot compensate

TCP Service Primitives

- Services defined in terms of primitives and parameters
- Primitive specifies function to be performed
- Parameters pass data and control information
- These defines the so-called socket programming

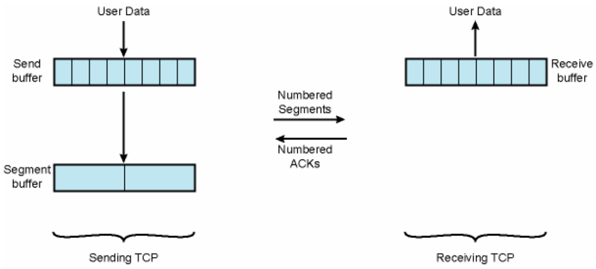
Use of TCP and IP Service Primitives



Basic Operation

- Data transmitted in segments
  - TCP header and portion of user data
  - Some segments carry no data
    - For connection management
- Data passed to TCP by user in sequence of Send primitives
- Buffered in send buffer
- TCP assembles data from buffer into segment and transmits
- Segment transmitted by IP service
- Delivered to destination TCP entity
- Strips off header and places data in receive buffer
- TCP notifies its user by Deliver primitive that data are available

Basic TCP Operation



Items Passed to IP

- TCP can pass some parameters down to IP
  - Precedence
  - Normal delay/low delay
  - Normal throughput/high throughput
  - Normal reliability/high reliability
  - Security

TCP Mechanisms (1)

- Connection establishment
  - Three way handshake
  - Between pairs of ports
  - One port can connect to multiple destinations
- Data transfer
  - Logical stream of octets
  - Octets numbered modulo 2^{32}
  - Flow control by credit allocation of number of octets
  - Data buffered at transmitter and receiver

Implementation Policy Options

- Send
- Deliver
- Accept
- Retransmit
- Acknowledge

Send

- If no push or close TCP entity transmits at its own convenience
- Data buffered at transmit buffer
- May construct segment per data batch
- May wait for certain amount of data

Deliver

- In absence of push, deliver data at own convenience
- May deliver as each in order segment received
- May buffer data from more than one segment

Accept

- Segments may arrive out of order
- In order
  - Only accept segments in order
  - Discard out of order segments
- In windows
  - Accept all segments within receive window

Retransmit

- TCP maintains queue of segments transmitted but not acknowledged
- TCP will retransmit if not ACKed in given time
  - First only
  - Batch
  - Individual

Acknowledgement

- Cumulative
  - Always ACK all the data received in order, allows for quasi-selective repeat without (almost) any overhead
- Immediate
  - send one ACK per packet
- Delayed
  - send ACKs with delay to allow data piggibacking or every 2 segments received

Silly Window Syndrome

- The unnecessary splitting of the Tx window in many small segments due to protocol operation
- Caused either by
  - the receiver, solved by simple logic
  - the sender, solved by Nagle's Algorithm – RFC 896
- If not prevented it is a normal phenomenon and "kills" TCP performance

Receiver

- Avoid setting rcwnd < MSS
  1. Try pushing data to the application in large chunks, this is a matter of socket management and process speeds
  2. If buffer space is < MSS ➔ rcwnd=0

Sender: Nagle's Algrithm

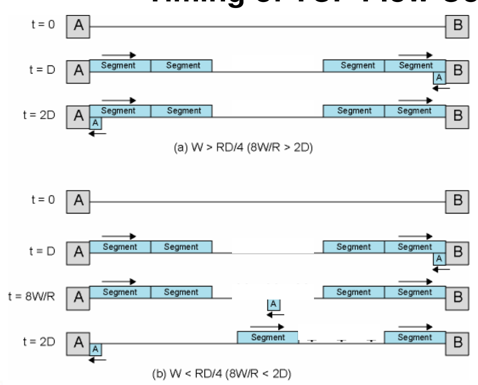
```
if there is new data to send
   if the window size and available data is >= MSS
          send complete MSS size segment now
   else
       if there is unconfirmed data still in the pipe
          enqueue data in the buffer until an ack is received
       else send data immediately
```

- Again it has to do with socket managemnt
- Works well for telnet or file transfers
- Interacts badly with delayed ACK on other applications (X, Web, …)

Effect of Window Size (reprise)

- W = TCP window size (octets)
- R = Data rate (bps) at TCP source
- D = End-to-End delay (seconds)
- After TCP source begins transmitting, it takes D seconds for first octet to arrive, and D seconds for acknowledgement to return
- TCP source should transmit 2RD bits, or RD/4 octets to "fill the pipe"

Timing of TCP Flow Control

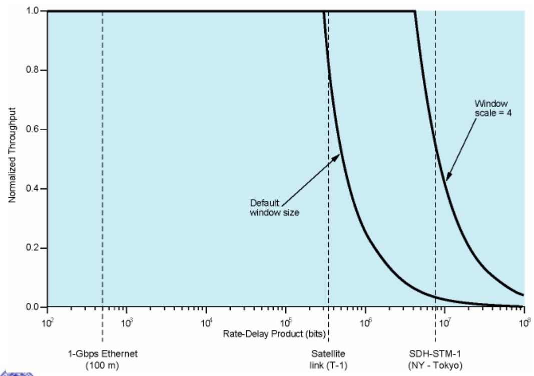


Normalized Throughput S

$$S = \begin{cases} 1 & W > RD/4 \\ \dfrac{4W}{RD} & W < RD/4 \end{cases}$$

Where are stored the W-RD/4 excessive bytes?

TCP Flow Control Performance



Complicating Factors

- Multiple TCP connections multiplexed over same network interface
  - Reducing R and efficiency
- For multi-hop connections, D is sum of delays across each network plus delays at each router
- If source data rate R exceeds data rate on a hop, that hop will be bottleneck
- Lost segments retransmitted, reducing throughput
  - Impact depends on retransmission policy

Retransmission Strategy

- TCP relies on positive acknowledgements
  - Retransmission on timeout or duplicated ACKs
- No explicit negative acknowledgement
- Retransmission required when:
  - Segment arrives damaged
    - Checksum error
    - Receiver discards
  - Segment fails to arrive

Timers

- Timer (a single one per each TCP send process) initialized with each segment as it is sent
- If timer expires before acknowledgement, sender must retransmit
- Value of retransmission timer is key
  - Too small: many unnecessary retransmissions, wasting network bandwidth
  - Too large: delay in handling lost segment

Two Strategies

- Timer should be longer than round-trip delay
- Delay is variable

- Strategies:
- **Fixed timer**
- ***Adaptive***

Problems with Adaptive Scheme

- Peer TCP entity may accumulate acknowledgements and not acknowledge immediately
- For retransmitted segments, can't tell whether acknowledgement is response to original transmission or retransmission
- Network conditions may change suddenly

Average Round-Trip Time (ARTT)

- Take average of observed round-trip times over number of segments
- If average accurately predicts future delays, resulting retransmission timer will yield good performance

$$\mathrm{ARTT}(K+1) = \frac{1}{K+1}\sum_{i=1}^{K+1}\mathrm{RTT}(i)$$

- Use this formula to avoid recalculating sum every time

$$\mathrm{ARTT}(K+1) = \frac{K}{K+1}\mathrm{ARTT}(K) + \frac{1}{K+1}\mathrm{RTT}(K+1)$$

RFC 793 Exponential Averaging

- Smoothed Round-Trip Time (SRTT)

$\mathrm{SRTT}(K+1) = \alpha * \mathrm{SRTT}(K) + (1-\alpha) * \mathrm{RTT}(K+1)$

- Gives greater weight to more recent values as shown by expansion of above:

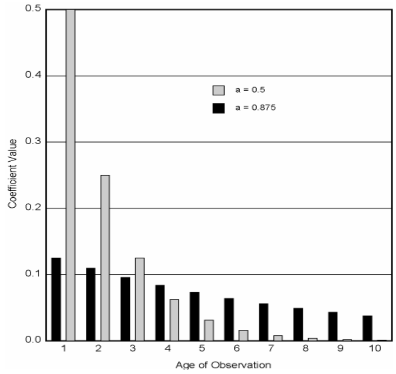
$\mathrm{SRTT}(K+1) = (1-\alpha)\mathrm{RTT}(K+1) + \alpha(1-\alpha)\mathrm{RTT}(K) + \alpha^2(1-\alpha)\mathrm{RTT}(K-1) + \ldots + \alpha^K(1-\alpha)\mathrm{RTT}(1)$

- α and $1-\alpha$ < 1 so successive terms get smaller
- E.g. = 0.8

$\mathrm{SRTT}(K+1) = 0.2\ \mathrm{RTT}(K+1) + 0.16\ \mathrm{RTT}(K) + 0.128\ \mathrm{RTT}(K-1) + \ldots$

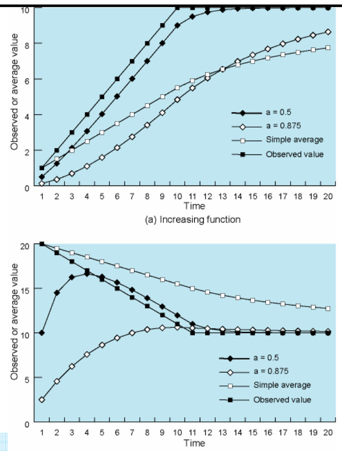
- Smaller values of α give greater weight to recent values

Exponential Smoothing Coefficients



Behavior of exp. averaging with varying RTT values

- Actual RTT increases or decreases
- Curves show the actual value and the values obtained with different averaging techniques



(a) Increasing function

(b) Decreasing function

RFC 793 Retransmission Timeout

- Equation above used in RFC 793 to estimate current round-trip time
- Retransmission timer set somewhat greater
- Could use a constant value:

$$RTO(K+1) = SRTT(K+1) + \Delta$$

- RTO is retransmission timer (or retransmission timeout)
- Δ is a constant
- Δ not proportional to SRTT
  - Large values of SRTT, Δ relatively small
  - Fluctuations in RTT result in unnecessary retransmissions
  - Small values of SRTT, Δ is relatively large
  - Unnecessary delays in retransmitting lost segments

RFC 793 Retransmission Timeout

- Use of timer value is proportional to SRTT, within limits

```
RTO(K+1)=MIN(UBOUND,MAX(LBOUND,β*SRTT(K+1)))
```

- UBOUND and LBOUND pre chosen fixed upper and lower bounds on timer value and β is a constant
- RFC 793 does not recommend values but gives "example values"
  - α between 0.8 and 0.9 and β between 1.3 and 2.0

Modern Retransmission Timeout

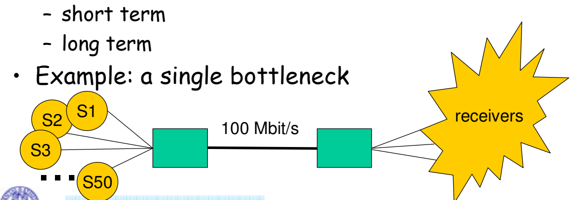
- Use of timer value is proportional to SRTT, within limits

```
RTO(K+1)=MIN(UBOUND,MAX(LBOUND,β*SRTT(K+1)))
```

- UBOUND and LBOUND pre chosen fixed upper and lower bounds on timer value and β is a constant
- RFC 793 does not recommend values but gives "example values"
  - α between 0.8 and 0.9 and β between 1.3 and 2.0
- **More on timeout setting while discussing congestion control & "modern" RTO setting**

TCP Congestion Control

- Retransmissions are useless if the network is congested ... but what is congestion?
- **Congestion**: a state of a network when the offered load (or traffic) η is larger than the network capacity C
  - normalize $\rho = \eta/C$ so that $\rho >= 1$ means congestion
  - short term
  - long term
- Example: a single bottleneck



TCP Congestion Control Alternative Options

- Dynamic routing can alleviate congestion by spreading load more evenly
- But only effective for unbalanced loads and brief surges in traffic
- Indeed IP routing is dynamic only in face of failures or topology changes
- Load-dependant, or QoS routing is a topic discussed, researched-on and tested since 30 years, but never implemented

TCP Congestion Control Alternative Options

- Congestion can only be controlled by limiting total amount of data entering network
  - **I.E. making $\rho < 1$**
- ICMP source Quench message is crude and not effective … and really not implements in hosts
- RSVP may help but not widely implemented
- No other Connection Admission Control Techniques available in the Internet

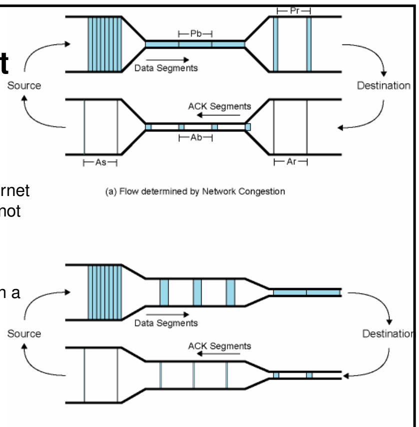
TCP Congestion Control is Difficult

- IP is connectionless and stateless, with no provision for detecting or controlling congestion
- TCP only provides end-to-end flow control
- No cooperative, distributed algorithm to bind together various TCP entities
- No cooperation between IP and TCP
  - When links/routers are congested IP drops packets
  - TCP retransmit them … **increasing the load!!!**

TCP Flow and Congestion Control

- The rate at which a TCP entity can transmit is determined by rate of incoming ACKs to previous segments with new credit
- Rate of Ack arrival determined by round-trip path between source and destination
- Bottleneck may be destination or internet
- Sender cannot tell which
- Only the internet bottleneck can be due to congestion

TCP Segment Pacing

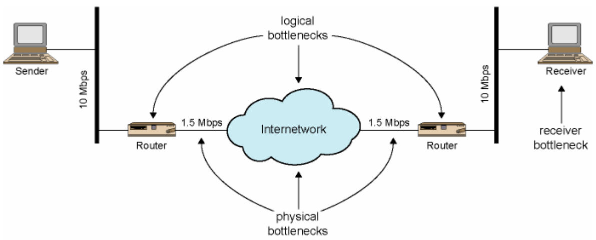


(a) Flow determined by Network Congestion

(b) Flow determined by Destination System

- Congestion in the Internet and at the receiver cannot be distinguished
- Both related with transmission window
- TCP handles both with a single window, which creates a lot of complexities

Context of TCP Flow and Congestion Control



Retransmission Timer Management

- Congestion affects RTT → a technique to efficiently and reliably compute RTO is needed
- Three Techniques to calculate retransmission timer (RTO):
- RTT Variance Estimation
- Exponential RTO Backoff
- Karn's Algorithm

RTT Variance Estimation (Jacobson's Algorithm)

- 3 sources of high variance in RTT
- If data rate relative low, then transmission delay will be relatively large, with larger variance due to variance in packet size
- Load may change abruptly due to other sources
- Peer may not acknowledge segments immediately

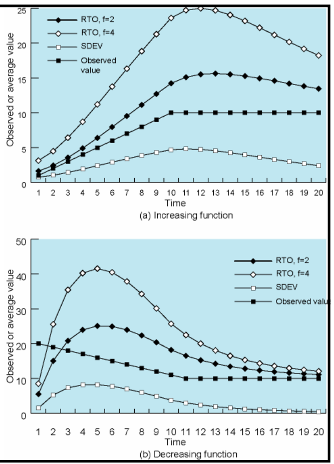
Jacobson's Algorithm

- $SRTT(K + 1) = (1 - g) \times SRTT(K) + g \times RTT(K + 1)$
- $SERR(K + 1) = RTT(K + 1) - SRTT(K)$
- $SDEV(K + 1) = (1 - h) \times SDEV(K) + h \times |SERR(K + 1)|$
- $RTO(K + 1) = SRTT(K + 1) + f \times SDEV(K + 1)$
- $g = 0.125$
- $h = 0.25$
- $f = 2$ or $f = 4$ (most current implementations use $f = 4$)

Jacobson's RTO Calculation

Figures show the behavior when RTT increase from 0 to 10 or decreases from 20 to 10

The most used implementation grossly overestimates since has a large wheight on variance



Two Other Factors

- Jacobson's algorithm can significantly improve TCP performance, but:
- What RTO to use for retransmitted segments?
  - ANSWER: exponential RTO backoff algorithm
- Which round-trip samples to use as input to Jacobson's algorithm?
  - ANSWER: Karn's algorithm

Exponential RTO Backoff

- Increase RTO each time the **same segment** is retransmitted – backoff process
- Multiply RTO by constant:
  - **RTO = q × RTO**
- q = 2 → binary exponential backoff

Which Round-trip Samples?

- If an ack is received for retransmitted segment, there are 2 possibilities:
  - Ack is for first transmission
  - Ack is for second transmission
- TCP source cannot distinguish 2 cases
- No valid way to calculate RTT:
  - From first transmission to ack, or
  - From second transmission to ack?

Karn's Algorithm

- Do not use measured RTT to update SRTT and SDEV
- Calculate backoff RTO when a retransmission occurs
- Use backoff RTO for segments until an ack arrives for a segment that has not been retransmitted
- Then use Jacobson's algorithm to calculate RTO

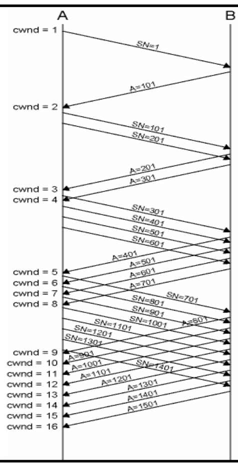
Window Management

- Slow start
- Congestion Avoidance
- Dynamic window sizing on congestion
- Fast retransmit
- Fast recovery
- Limited transmit

Slow Start

- awnd = MIN[credit, cwnd]
  - where
  - awnd = allowed window in segments
  - cwnd = congestion window in segments
  - credit = amount of unused credit granted in most recent ack (rcwn)
- cwnd = 1 for a new connection and increased by 1 for each ack received, up to a maximum
- Most implementations are not compliant and start from 2 to "counter" delayed ACKs

Effect of Slow Start



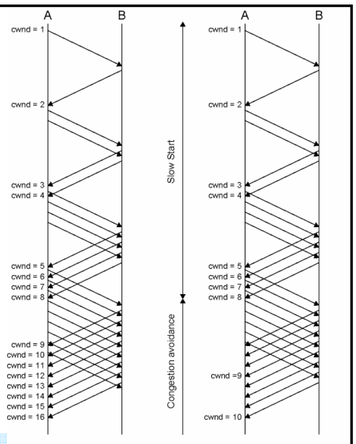
Dynamic Window Sizing on Congestion

- A lost segment indicates congestion
- Prudent to reset cwsd = 1 and begin slow start process
  - The first implementation of TCP with dynamic window
- Highly inefficient since throughput is related to the integral of the window in time
- The integral of "exponentials" -- indeed geometrics is small
- Timeouts are normally long w.r.t. RTT

Dynamic Window Sizing on Congestion

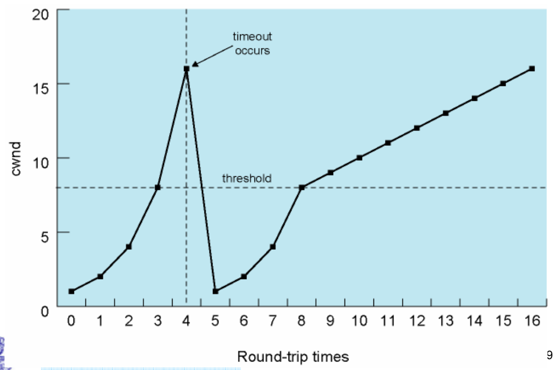
- One problem is "doubling" of the offered load each RTT → much better to have a "gentler" increase when the throughput is "reasonable"
- Linear window growth after a threshold
- Called "**congestion avoidance**"
- First introduced in 1988/1989 in a TCP version (BSD 4.3) called Tahoe

Slow Start and Congestion Avoidance



(a) Slow start, ending with a timeout (b) Slow start followed by congestion avoidance

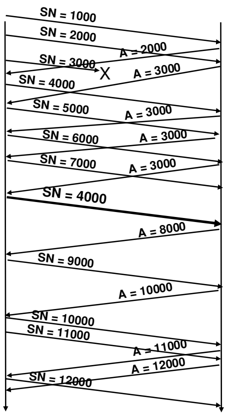
Window size during Slow Start and Congestion Avoidance



Fast Retransmit

- RTO is generally noticeably longer than actual RTT
- If a segment is lost, TCP may be slow to retransmit
- TCP rule: if a segment is received out of order, an ack must be issued immediately for the last in-order segment
- Fast Retransmit rule: if 4 acks received for same segment, highly likely it was lost, so retransmit immediately, rather than waiting for timeout

Fast Retransmit



Fast retransmit works only for med. to large windows (cwnd > 4)

In any case transmission is stopped for 1 RTT and then restart from scratch

Fast Recovery

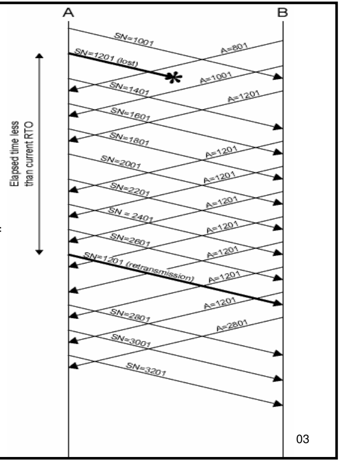
- When TCP retransmits a segment using Fast Retransmit, a segment was assumed lost
- Congestion avoidance measures are appropriate at this point
  - Slow-start/congestion avoidance procedure
- This may be unnecessarily conservative since multiple acks indicate segments are getting through
- **Fast Recovery: retransmit lost segment, cut cwnd in half, go into congestion avoidance**
- This avoids initial exponential slow-start

Fast Recovery

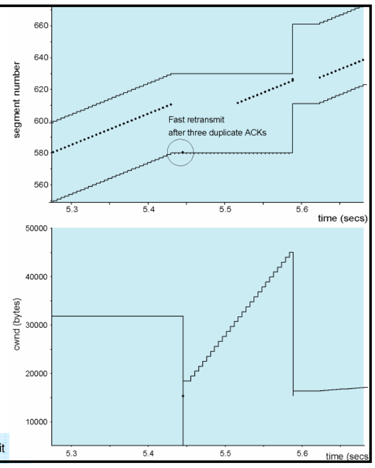
Window management is quite difficult with Fast Recovery

First implementations were wrong (TCP Reno) and went in timeout if more than one segment was lost in a window

The correct implementation (TCP NewReno) requires keeping additional state



Fast Recovery Example of window evolution (sinmplified)



Limited Transmit

- If congestion window at sender is small, fast retransmit may not get triggered,
  - e.g., cwnd = 3
- Under what circumstances does sender have small congestion window?
- Is the problem common?
- If the problem is common, why not reduce number of duplicate acks needed to trigger retransmit?

Limited Transmit Algorithm

- Sender can transmit new segment when 3 conditions are met:
  - Two consecutive duplicate acks are received
  - Destination advertised window allows transmission of segment
  - Amount of outstanding data after sending is less than or equal to cwnd + 2 (i.e. the window was exactly 3)
- Rarely implemented, solves just a limited number of cases
- What about correlated losses?

SACK, RED/ECN and Throughput Modeling

TCP Throughput

- What is the throughput achievable by TCP?
- Integral of the window size in time
- Can we predict TCP throughput?
- What are the free parameters
  - Loss probability (is it independent from TCP itself?)
  - RTT, Number of connections, …
- Can we decouple flow from congestion control?
- Can we avoid dropping packets due to congestion?

SACK Option (RFC 2081)

- Negotiation at startup to verify if both ends are enabled
- "Holes" in the receiver buffer sent back to the sender as couple of pointers in ACK optional fields
- Can improve performance (not much!) with highly correlated losses
- Can sometimes lead to blocks and timeouts (implemntation bugs?)

Timestamp Option

- RFC 1323 (as windowscale)
- "Normal" TCP can only compute one RTT sample per window since the only timer is overwritten every new transmission
- With timestamp each segment is stamped with time, ACKs are stamped too
- There can be 1 RTT sample per segment
- RTT can be computed more precisely and RTO can be set more accurately
- Additionally can solve window wrap around problems

Explicit Congestion Notification (ECN)

- RFC 3168
- Routers alert end systems to growing congestion
  - End systems reduce offered load
  - With implicit congestion notification, TCP deduces congestion by noting increasing delays or dropped segments
- Benefits of ECN
  - Prevents unnecessary lost segments
    - Alert end systems before congestion causes dropped packets
    - Retransmissions which add to load avoided
  - Sources informed of congestion quickly and unambiguously
    - No need to wait for retransmit timeout or three duplicate ACKs
- Disadvantages
  - Changes to TCP and IP header
  - New information between TCP and IP
    - New parameters in IP service primitives

Changes Required for ECN

- Two new bits added to TCP header
  - TCP entity on hosts must recognize and set these bits
- TCP entities exchange ECN information with IP
- TCP entities enable ECN by negotiation at connection establishment time
- TCP entities respond to receipt of ECN information
- Two new bits added to IP header
  - IP entity on hosts and **routers** must recognize and set these
- IP entities in hosts exchange ECN information with TCP
- IP entities in routers must set ECN bits based on congestion

IP Header

- Prior to introduction of differentiated services IPv4 header included 8-bit Type of Service field
- IPv6 header included 8-bit traffic class field
- With DS, these fields reallocated
  - Leftmost 6 bits dedicated to DS field,
  - Rightmost 2 bits designated currently unused (CU)
- RFC 3260 renames CU bits as ECN field

The ECN field has the following interpretations:

Value	Label	Meaning
00	Not-ECT	Packet is not using ECN
01	ECT (1)	ECN-capable transport
10	ECT (0)	ECN-capable transport
11	CE	Congestion experienced

TCP Header

- To support ECN, two new flag bits added
- ECN-Echo (ECE) flag
  - Used by receiver to inform sender when CE packet has been received
- Congestion Window Reduced (CWR) flag
  - Used by sender to inform receiver that sender's congestion window has been reduced

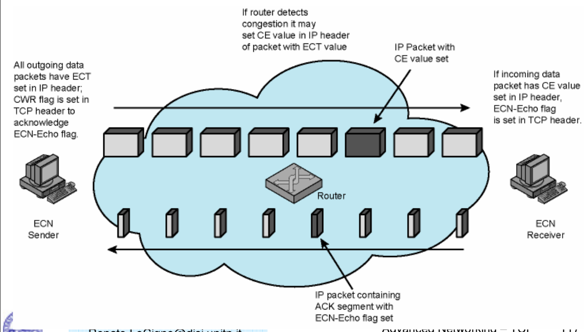
TCP Initialization

- TCP header bits used in connection establishment to enable end points to agree to use ECN
- A sends SYN segment to B with ECE and CWR set
  - A ECN-capable and prepared to use ECN as both sender and receiver
- If B prepared to use ECN, returns SYN-ACK segment with ECE set CWR not set
- If B not prepared to use ECN, returns SYN-ACK segment with ECE and CWR not set

Basic Operation

- TCP host sending data sets ECT code (10 or 01) in IP header of every data segment sent
- If sender receives TCP segment with ECE set, sender adjusts congestion window as for fast recovery from single lost segment
- Next data segment sent has CWR flag set
  - Tells receiver that it has reacted to congestion
- If router begins to experience congestion, may set CE code (11) in any packet with ECT code set
- When receiver receives packet with CE set it begins to set ECE flag on all acknowledgments (with or without data)
  - Continues to set ECE flag until it receives segment with CWR set

Basic ECN Operation



Open problems

- How to properly react to multiple ECN indications so that window is not reduced too much
  - make a single reduction per RTT
  - some "complicated" heuristics to achieve that
- When buffers are full, packets are lost in bursts
  - problem in general with TCP, also with ECN
- Can we manage buffers to avoid bursty losses?