7. TRANSPORT SERVICES and TCP/IP
Copyrights on Figures

• Figures presented in these notes with the following captions are extracted from the textbook and copyrighted by the publisher as indicated

• DISCLAIMER
  – Whilst every attempt is made to maintain the accuracy and correctness of these notes, the author, Dr. Roberto Togneri, makes no warranty or guarantee or promise express or implied concerning the content.
Application Needs

• End-to-End applications
  – Establish a connection to another application on another host
  – Transfer data in full-duplex mode reliably
    • In sequence; No duplicates; No losses
  – Terminate connection when finished
  – Examples
    • File Transfer; Interactive logins

• Multimedia applications
  – Transfer data with constant delay
    • single direction (broadcasting)
  – Transfer data with constant and minimum delay
    • half-duplex / full-duplex audio/video communication
  – Examples
    • Internet telephone; Video/Audio broadcasting

• Client-Server applications
  – Remote Procedure Call (RPC)
    • Client sends request to server
    • Server passes request arguments to procedure
    • Server returns reply to client
  – Examples
    • World-Wide-Web
Need for Internet Transport Service

- Characteristics of Internet
  - Unreliable, connectionless, and no QoS
    - Datagrams can get lost; Datagrams can arrive out of order; Duplicate datagrams

- Transmission Control Protocol (TCP)
  - RFC 793, 1122 (implementation), 1323 (extensions)
  - End-to-End applications
  - Services Provided
    - Connection establishment / termination
    - Application addressing
    - Reliable data transfer / Flow control
    - Crash Recovery / Congestion / Re-transmission Control
  - Byte Streaming Data Protocol
    - TCP controls size of IP datagrams used; block data transfers not directly supported

- User Datagram Protocol (UDP)
  - RFC 768
  - Minimal services; allows user to tailor UDP to specific applications
  - Services Provided
    - Application addressing
  - Blocked Data Protocol
    - User can control size of IP datagram used in transmission / reception of data
Internet Transport Service

- Transport Services
  - Addressing
  - Connection Establishment and Termination
  - Data Transfer (Flow Control)

- Protocol: TCP
  - Default Transport protocol
  - Congestion control mechanisms

- API: Berkeley Sockets
  - Lowest level programming interface
  - Provides access to both TCP and UDP

- Transport PDU (TPDU) or segment
  - [ TCP header | user data ]
  - TCP segment used for control & management
  - May contain 0 user data

- Encapsulation
  - [ IP header | TCP header | user data ]
  - [ IP header | UDP header | user data ]
TCP Format

- Source / Destination Port [16]
  - Identifier for source and destination transport users
  - Multiple users can be defined for the same host
- Sequence Number [32] (SEQ)
  - Used for TCP flow control
  - Sequence number of the first data octet in the data segment
  - Sequence numbers for data octets not segments

![Figure 17.14 (dcc5e)](image-url)
TCP Format

- Acknowledgment Number [32] (ACKN)
  - Allows *piggybacked* acknowledgments
    - ACKs are contained in the return data packets (at no cost!)
    - If predominantly one-way communication ACK-only (no data) TCP segments generated
  - Used by TCP receiving entity to indicate to the TCP sender that the next data octet the TCP receiving entity needs has a sequence number of ACKN

- Data Offset [4]
  - Number of 32-bit words in the header

- Reserved [6]

- Flags [6]
  - URG: If 1 urgent pointer field significant
  - ACK: If 1 segment contains a valid acknowledgment
  - PSH: If 1 segment contains PUSH data
  - RST: If 1 the connection has to be reset
  - SYN: Used to establish a connection
  - FIN: Used to terminate a connection

- Window [16] (WIN)
  - Size in octets for the flow control sliding window / credit allocation protocol
  - Used by the receiving TCP entity to indicate to the TCP sender that it can send data octets with sequence numbers from ACKN to ACKN + (WIN-1)
  - If WIN=0 then TCP sender can not send any more data
TCP Format

– Checksum [16]
  • Same checksum algorithm as IP
  • Checksum is computed over:
    – TCP header (excluding Checksum field)
    – TCP segment data
    – TCP segment length [16]
    – IP source address [32]
    – IP destination address [32]
    – IP protocol [8]

– Urgent Pointer [16]
  • Points to the location of the URGENT data within the TCP segment
  • A single TCP segment can contain both URGENT and normal data

– Options [variable]
  • Maximum Segment Size (MSS)
    – Maximum size of data payload negotiated at connection establishment
      • End TCP entities exchange their preferred MSS and the smallest value is adopted
    – Usually MSS = MTU-headers to avoid IP fragmentation (e.g. 1500-[20+20] for Ethernet)
    – Default MSS = 536 bytes
  • Window Scale Factor
  • Timestamp

– Data [variable]
  • The segment of application data
TCP Addressing

- **Transport User Addressing**
  - User-defined End-to-End communication
    - User arbitrarily defines the identity of the source and destination processes by using PORT numbers
    - Example:
      
      **Source:** Port 1153  Host 130.95.111.1  
      **Destination:** Port 1188  Host 130.95.208.5
  
  - Client-Server communication
    - User must know the destination PORT number of the server → *Addressing Problem*
    - The source PORT number is usually assigned arbitrarily

- **Addressing Problem Solutions**
  - Hard-coded addresses
    - Proprietary client-server programs using a reserved or predefined PORT
  
  - Well-known addresses
    - Listed in services database file
      - *Example:* telnet port = 23, www port = 80, ftp port = 21
  
  - Consult a PORT name server
    - Equivalent to DNS for Internet addresses
    - NIS (YP) of services information
Multiplexing

• **Upward multiplexing**
  – Multiple transport connections on one network connection
  – X.25 / Frame Relay / ATM networks
    • Carrier pricing on a per network connection basis
    • Use upward multiplexing to allow multiple users to use the same network connection and share the cost
  – Internet networks
    • Multiple servers, user applications, etc. can be run on the same host with one connection
  – TCP Source and Destination Ports

• **Downward multiplexing**
  – One transport connection uses multiple network connections
  – Satellite networks
    • Point-to-point window flow control forces under-utilisation of network capacity → low-bandwidth
    • High-bandwidth transport connection can open multiple low-bandwidth network connections
  – TCP does not define downward multiplexing
Connection Establishment

- Connection establishment protocol
  - Each user knows of the other’s existence
    - “they shake hands and introduce themselves”
  - Users negotiate data transfer parameters by each sending a special SYN packet
    - Exchange initial sequence numbers
    - Agree on maximum window and segment size
    - Agree on QoS parameters
  - Triggers allocation of resources
    - Buffer space
    - Connection identifier and handler

- 2-way handshake: Symmetric Operation
  - End user A does **Active Open**
    - A sends a SYN packet to B to indicate it wants to communicate
  - End user B does **Active Open**
    - B sends a SYN packet to A to indicate it wants to communicate
  - Problems?
    - Recipient user of the SYN must be in a state to accept a communication → ready and waiting
    - Difficult to implement client-server interactions which use an asymmetric connection establishment paradigm (server *listens* for client *connection*)
Connection Establishment

- 2-way handshake: Asymmetric Operation
  - End user A does Passive Open
    - A enters a listening state whereby it waits for B to call it when it's ready to communicate
  - End user B does Active Open
    - B sends SYN packet to A to indicate it wants to start communicating
  - End user A acknowledges its ready
    - A sends a SYN packet to B to indicate it is ready to start communicating
  - Features
    - Problems: Agreement as to which end user listens and which end user connects
    - Client-server connection establishment paradigm:
      Server enters a listening state and waits for any client user to call it for service
Connection Establishment

- Problems with 2-way handshake over an unreliable Internet
  - SYN packets may get lost
    Solution? Timeout and re-transmit

- Delayed Duplicate / Obsolete SYN packets may arrive
  Solution? Three-Way Handshake
  - A sends SYN to B
  - B sends SYN to A and also acknowledges A’s SYN
  - A sends ACK to B to acknowledge B’s SYN

- Data segment from a previous connection may arrive
  Solution? Random initial sequence numbers
  - Initial sequence numbers for connection chosen so that any previous connection data segments will have an invalid sequence number
Connection Establishment

Examples of 2-way handshake problems

- A initiates a connection
- B accepts and acknowledges
- A begins transmission
- Connection closed
- New connection opened
- Obsolete segment SN = 2 is accepted; valid segment SN = 2 is discarded as duplicate

Figure 17.10 (dcc5e)

Figure 17.11 (dcc5e)
Connection Termination

• Connection Termination Protocol
  – One (or both) of the end users begins to terminate connection
    • Sends a special FIN packet after the last data to close the data transfer to end-user
    • Has to still receive data from other end user until both agree to terminate connection
  – Other end user reciprocates
    • Finishes transmitting data and then sends a FIN
    • Connection is closed when both end-users have sent the FIN packet

• Problems on the Internet
  – FIN packets can get lost
    Solution? Timeout and re-transmit
  – Delayed Duplicate / Obsolete FIN packets may arrive
    Solution? Three-Way Handshake
Berkeley Sockets API

• Berkeley Sockets Paradigm
  – Passive Open (Server)
    • sock = socket(AF_INET, SOCK_STREAM, …)
    • bind(sock, struct sockaddr *name, …)
    • listen(sock, num)
    • msgsock = accept(sock,…)
  – Active Open (Client)
    • sock = socket(AF_INET, SOCK_STREAM, …)
    • connect (sock, struct sockaddr *to, …)
  – Active Close (either user)
    • close(msgsock) or ^C
  – Passive Close (other user)
    • read(…) / write(…) immediately return with 0, then do a close(msgsock)

• Connection Identification
  – Unique connection identified by the pair (portA, portB) for end-users A and B
  – Server PORT (portA) is specified in the by the client in the connect (…) call
  – Client PORT (portB) arbitrarily assigned by system
  – More than one connection possible using the same portA on host A from different clients on the same (or different) host B
    • Example: (portA, 1333) (portA, 1334) (portA, 1335)
TCP Protocol: Connection Establishment

• Client sends SYN TPDU
  – Flags: SYN = 1
    • this is a SYN TPDU with parameters for establishing connection
  – SEQ = x
    • initial sequence number for client to server data
    • number is selected randomly
  – MSS Option and maybe Data

• Server responds with SYN+ACK TPDU
  – Flags: SYN = 1, ACK = 1
    • this is a SYN TPDU with a valid ACKN
  – SEQ = y
    • initial sequence number for server to client data
  – ACKN = x + 1
    • server expects the next data octet from client to have a sequence number of x + 1
      \[\rightarrow\] effectively acknowledges the client SYN
  – MSS Option and maybe Data
TCP Protocol: Connection Establishment

- Client responds with ACK TPDU
  - Flags: ACK = 1
    - this is a normal data segment with a valid ACKN
  - ACKN = y + 1
    - client expects the next data octet from server to have a sequence number of y + 1
      → effectively acknowledges the server SYN
  - Can include first segment of user data
    - with SEQ = x + 1

Figure 6.5 (cne3e)
TCP Protocol: Connection Establishment

• Re-transmission Timers
  – Client SYN TPDU
    • Re-transmit if server does not ACK
  – Server SYN TPDU
    • Re-transmit if client does not ACK

• No Server process listening?
  – Client/Server entity will keep re-transmitting SYN TPDU's
  – Solution? Server host transport entity sends a RST TPDU
    • Flags: RST = 1

• Obsolete (SYN / Data) TPDU’s
  – SEQ and ACKN values will be incorrect (unexpected) and a RST TPDU will be sent to the originating entity by whichever entity detects the discrepancy
    • Flags: RST = 1
    • ACKN = SEQ of TPDU causing the reset response
TCP Protocol: Connection Termination

- End user X sends a FIN TPDU
  - Flags: FIN = 1
  - SEQ = x
    - Sequence number of next data octet to be sent to Y
  - FIN TPDU usually has no data
- End entity Y sends ACK TPDU
  - Flags: ACK = 1
    - this is a FIN TPDU with a valid ACKN
  - ACKN = x + 1
    - Acknowledges the FIN TPDU from X
- End user Y sends a FIN TPDU
  - Flags: FIN = 1
  - SEQ = y
- End entity X sends ACK TPDU
  - Flags: ACK = 1
  - ACKN = y + 1
- 4 exchanges not 3!
  - Entity Y may send a FIN TPDU sometime after receiving the FIN TPDU from X, so a combined (FIN+ACK) is not possible
TCP State Diagram

Figure 17.12 (dcc5e)
TCP Protocol: Connection Termination

- Re-transmission Timers
  - Either end user sends FIN TPDU (enter FIN_WAIT state) by a close()
    - Re-transmit if other user does not acknowledge
  - Transport entity (not user) will acknowledge
    - Indefinite re-transmissions avoided
    - If host down, then a ICMP destination unreachable is used

- TCP State Diagram Comments
  - What is FIN_WAIT2?
    - User executing close() enters FIN_WAIT2 when other entity acknowledges
    - One direction of the connection is closed.
  - What is TIME_WAIT?
    - User in FIN_WAIT2 receives the FIN and responds with an ACK
    - The connection is closed, so why wait?
      - Other end entity may not receive ACK and may re-transmit the FIN
      - If connection CLOSED and new connection established with same PORT numbers then re-transmitted FIN will force a RST TPDU response or may even close the connection!
  - ABORT condition
    - If in FIN_WAIT2 and FIN never arrives connection is released after timer expires
    - If FIN then arrives an RST TPDU is sent
Crash Recovery

- **Normal ACK operation**
  - A sends segment to B
  - B acknowledges segment when it has processed it (e.g. written to disk)

- **Host B crashes problem**
  - If B acknowledges(A) then crashes(C) the segment is not processed
  - If B processes data(W) then crashes(C) the segment is not acknowledged and a duplicate is sent (since B is rebooted it won’t recognise a duplicate!)

- **Host B crashes solution**
  - **Solution 1:** If host crashes all connections are aborted and restarted
    - Not a good solution for distributed computing file servers that crash (e.g. NFS)
  - **Solution 2:** Sending host A adopts a strategy to deal with host B crashes
    - 1: Host A always retransmits a segment
    - 2: Host A never retransmits a segment
    - 3: Host A only retransmits segments which have been acknowledged (S0)
    - 4: Host A only retransmits segments which have not been acknowledged (S1)

![Figure 6.18 (cne3e)](image-url)
Flow Control: Sliding-Window Protocol

• Sliding Window Protocol
  – Sender A and receiver B agree on fixed-size flow control window, WIN
  – Sequence of operations
    • B sends ACKN to A, then A can send from octet ACKN to octet ACKN + (WIN-1)
  – Purpose of Flow Control
    • B sends ACKN such that receiver buffer will not overflow if A immediately sends up to octet number ACKN + (WIN-1)
  – Purpose of Acknowledgments
    • B sends ACKN for all data received so that A will not think data was lost and re-transmit

• Problems with unreliable network
  – Conflicting flow control and acknowledgment requirements
    • B has a receive buffer of size WIN and X < WIN octets of data have been received:
      – If B sends ACKN = X+1 to acknowledge all received data, then A can send up to X + WIN more data, but then B receiver will overflow since (X + WIN) > buffer size of WIN
      – If B does not send any ACKN to avoid overflowing buffer, then A will time-out and re-transmit the data!
  – Problem: Acknowledgments are tied with Flow Control
  – Solution: Separate acknowledgments from flow control (credit-allocation)
Flow Control: Credit-Allocation Scheme

- Adopts a variable-length window
  - Sender A and receiver B do not need to agree on any fixed window size
  - Sequence of operations:
    - B sends (ACKN, WIN):
      - ACKN = all data received by B
      - WIN = such that ACKN + (WIN-1) = free buffer space remaining
      - A can send from ACKN to ACKN+(WIN-1) octets
    - Separates Flow Control & Acknowledgments
      - If B has a receive buffer of size BUF and X < BUF octets of data have been received, and B sends (ACKN=X+1,WIN=BUF-X):
        - A knows that all X data octets were received (ACKN=X+1) and will not need to re-transmit data (acknowledgment OK)
        - A can transmit an additional (BUF-X) packets without overflowing receive buffer on B (flow control OK)
Flow Control: Credit-Allocation Scheme

(a) Send sequence space

(b) Receive sequence space

Figure 17.6 (dcc5e)
TCP Protocol: Flow Control

- TCP uses credit-allocation flow control (ACK, WIN fields)

By specifying a WIN=0 the transport receiver can stop the sender and wait for user receiver to read some data from a full buffer.

Figure 6.29 (cne3e)
TCP Protocol: Data Transfers

- **Reliable data using unreliable IP**
  - Out-of-order arrival
    - Receiver stores all received segments and sends ACKN to force re-transmit of lost segments
  - Duplicate segments
    - Receiver ignores segments with SEQ less than currently expected in-order SEQ
  - Lost segments
    - Sender uses a re-transmission timer on last data segment sent that has not been acknowledged

- **PUSH data**
  - Sending user specifies Flag: $PSH = 1$ to force TCP to send segment immediately
  - Useful for interactive data communications or to send control data

- **URGENT data**
  - Flag: $URG = 1$ and Urgent Pointer used to identify “out-of-band” data in segment that needs urgent (e.g. SIGURG interrupt) handling at the receiver.
  - Can be used in combination with PUSH
  - Used for control or real-time data
TCP Implementation Policies

• Send Policy
  – When should TCP send a segment?
    • Wait until TCP segment data size is MSS or half the receiver’s buffer size (which can be estimated from the receiver’s past WIN advertisements) whichever is smaller
    • Send immediately if PUSH data, subject to Nagle’s algorithm: only send the next batch of PUSH segments if current segment has been acknowledged
      – avoids sending small-byte segments by aggregating data and sending only as fast as receiver is processing (e.g. TELNET)
    • Wait for a timer to expire then send all there is

• Deliver Policy
  – When should TCP deliver data to user?
    • Can deliver in-order data in any size since TCP is a byte-stream protocol
    • Delivers as much in-order data as is available when user asks for data (i.e. read(…))
    • May block user until sufficient in-order data is available (i.e. read(…) blocks for data delivery)
    • Immediate delivery of URGENT data by signalling/interrupt mechanism
TCP Implementation Policies

• Accept Policy
  – Options for accepting segments
    • **In-order**: Accept only segments that arrive in order; out of order segments are discarded (c.f. go-back N)
    • **In-window**: Accept all segments that are within the receive window (c.f. selective repeat)
  – TCP uses an in-window policy

• Acknowledgment Policy
  – Options for sending acknowledgments:
    • **Immediate**: As soon as data is received it should be acknowledged even if this means using an empty segment
    • **Cumulative**: Wait for outbound data to piggyback acknowledgment. Use timer to avoid long waits and possible sender re-transmissions
  – TCP uses a Cumulative policy:
    • Delay acknowledgments and wait at most 500 msec for any outbound data to piggyback acknowledgments, otherwise use an empty segment.

• Window Update Policy
  – When should TCP advertise the window size?
    • Do not send a update on outbound data if WIN is too small. Send a window update only if receive buffer is half-empty or can handle MSS bytes, whichever is smaller.
TCP Implementation Policies

• Redundant ACKN / WIN updates to cope with lost ACKN / WIN
  – The current ACKN (next expected sequence number) and WIN (available credit allocation) of the incoming data are advertised in all outgoing data segments
  • All data segments will contain a valid acknowledgment (Flag: ACK = 1)

• Re-transmission Policy
  – Options for retransmitting segments:
    • **First-only**: One re-transmission timer for entire queue. If ACK is received, remove the appropriate segment(s) from the queue and reset the timer. If timer expires re-transmit the lowest SEQ segment not yet acknowledged.
      – Cheap to implement (one timer)
      – Minimum re-transmissions but long delay if all segments lost
      – Makes most sense with in-window accept policy
    
    • **Batch**: As for First-only except that when timer expires all unacknowledged segments that have been transmitted are re-transmitted
      – Cheap to implement (one timer)
      – Low delay since all segments transmitted but wasteful if only one segment is lost
      – Makes most sense with in-order accept policy
    
    • **Individual**: Maintain one timer for each outstanding segment in the queue, destroy timer when ACK arrives and re-transmit only that segment if timer expires
      – Expensive to implement (multiple timers)
      – Efficient since transmits only segments that are lost as soon as possible
  – TCP uses a **First-only** policy
TCP High-Speed Network Problems

• **Problem:** 32-bit SEQ too small
  – $2^{32} = 4$ GBytes
  – How long does it take for SEQ to wrap around if transmission rate is $\mu$ bps?
    • $T_w = 34 \times 10^9 / \mu$
  – Typical TCP wrap-around times ($T_w$):
    • Ethernet (10 Mbps): 57 minutes
    • Fast Ethernet (100 Mbps): 6 minutes
    • STS-24 (1.2 Gbps): 28 seconds
  – Maximum Segment Lifetime (MSL) $\approx 60$ sec (allow for 2 MSL = 120 sec)
    • There are problems with STS-24

• **Solution:** Second extension to TCP
  – Define 64-bit segment identification by [32-bit Timestamp | 32-bit SEQN]
    • Since timestamp always increases a wrap-around SEQ will be detected and rejected.
    • 32-bit Timestamp is a TCP option which is included in all data segments, originally proposed in the first extension to TCP for estimating the RTT
TCP High-Speed Network Problems

• **Problem:** 16-bit WIN too small
  – $2^{16} = 64$ KBytes
  – Bandwidth-Delay product, $R\mu$
    • $R$ is the Round-Trip Time (RTT) in seconds; $\mu$ is the data rate in bps;
      $W$ is the window size in bits ($W = WIN \times 8$)
    • The sender will have $R\mu$ bits of unacknowledged data before the first ACK arrives
      – It takes $R$ seconds for the ACK to arrive, in this time $R\mu$ bits of data will have been sent
    • If $W > R\mu$
      – Maximum Normalised Throughput, $U = 1$
    • If $W < R\mu$
      – Degraded Normalised Throughput, $U = W / (R\mu)$

• **Solution:** Third extension to TCP
  – When connection established a scaling factor, $SC$, is exchanged and the window size is then calculated as ($WIN \times SC$)
    • Window size is expressed in chunks of size $SC$
    • $SC$ is exchanged in the TCP options together with the standard MSS option during connection establishment
TCP High-Speed Network Problems

- Effect of SC parameter on U

Figure 10.4 (hsn1e)
TCP Timer Management

• Retransmission Timer
  – Segment retransmitted if timer expires before the ACK is received
    • ACK will take at least RTT from when the segment is sent, possibly more
  – Retransmission timer should be set to just over RTT
    • If too short → too many unnecessary retransmissions
    • If too long → too much delay before necessary retransmission
  – At the link layer RTT is easy to use
    • The RTT is simply twice the link propagation delay
    • The value for RTT is constant and accurate
  – At the Internet layer RTT is volatile
    • The RTT depends on the condition the network (i.e. congestion), status of the routers from source to destination, packet size, etc.
    • The TCP acknowledge policy allows cumulative acknowledgments, so an immediate acknowledgment should not be expected
  – Observed RTT for each segment: First extension to TCP
    • 32-bit Timestamp TCP option (Second extension) used for estimation of RTT
      – Sender includes its current timestamp for each segment sent
      – Receiver echoes the same timestamp in return ACK segment
      – Sender estimates RTT by taking the difference between the timestamp of when return segment is received and timestamp value stored in return ACK segment
TCP Timer Management

• Standard algorithm for setting retransmission timer
  – $RTT(i)$: RTT observed for the $i^{th}$ transmitted segment
  – $SRTT(K)$: Smoothed RTT for $RTT(i) : i = 1,2,\ldots, K$
    • $SRTT(K) = \alpha.SRTT(K-1) + (1-\alpha).RTT(K)$ (exponential averaging)
  – $RTO(K)$: Re-transmission timer value for the $K^{th}$ segment
    • $RTO(K) = \text{MIN}(\text{UBOUND, MAX}(\text{LBOUND}, \beta.SRTT(K)))$

• Typical Values
  • $\alpha = 0.875$
  • $\beta = 2$
  • Initialisation: $RTT(0) = 0$, $RTO(0) = 3$ sec

• Problem with Standard Algorithm
  – Problem 1: If segment is re-transmitted the return ACK may be in response to either the original segment or the re-transmitted segment
    • Will drastically change observed RTT if the wrong segment is used
    • Cause: Router congestion (queues full) will cause packets to be dropped and retransmissions to increase
  – Problem 2: Assumes little variation in $RTT(i)$
    • Typically under-estimates $RTO(K)$ with large fluctuations in $RTT(i)$
    • Cause: TCP cumulative acknowledgment policy, network congestion, IP packet size (packet transmission time)
TCP Timer Management

• **Solution 1: Karn’s Algorithm**
  – Disable calculation of $RTT(i)$ and $SRTT(K)$ for retransmitted segments
  – Use exponential backoff to set $RTO(K, r)$ for $r^{th}$ re-transmission of segment $K$
    • $RTO(K, r) = 2.RTO(K, r-1)$ (double timer value)
  – Resume calculation of $RTT(i)$ and $SRTT(K)$ for the next successfully transmitted segment

• **Solution 2: Jacobson’s Algorithm**
  – $SRTT(K)$: Smoothed RTT for $RTT(i) : i = 1,2,\ldots,K$
    • $SRTT(K) = \alpha . SRTT(K-1) + (1 - \alpha) . RTT(K)$
  – $SERR(K)$: Estimation Error for $SRTT(K)$
    • $SERR(K) = RTT(K) - SRTT(K-1)$
  – $SDEV(K)$: Smoothed Mean Deviation of $SRTT(K)$ ($\approx$ Standard Deviation)
    • $SDEV(K) = \gamma . SDEV(K-1) + (1 - \gamma) . |SERR(K)|$
  – $RTO(K)$: Re-transmission timer value for the $K^{th}$ segment
    • $RTO(K) = \beta . SRTT(K) + f . SDEV(K)$
  – Typical Values
    • $\alpha = 0.875 ; \gamma = 0.75$
    • $\beta = 1, f = 4$

• **Implementation Issues**
  – Use parameter values of the form $m(1/2)^n$ ($m, n$ integers) to avoid floating-point arithmetic
  – Observed $RTT(i)$ highly dependent on clock granularity of operating system
TCP Timers

• Persistence Timer
  – All segments have been acknowledged but sender is blocked because WIN = 0
    • Receiver has sent an updated WIN in an ACK segment but it got lost
      → sender and receiver are now in deadlock
  – When persistence timer expires sender transmits a dummy segment (probe) to
    the receiver
    • Receiver will respond by an ACK which includes the correct WIN value

• Keep-Alive Timer
  – All segments have been acknowledged but the connection has been idle
  – When keep-alive timer expires a probe is sent to check that the connection is
    alive
    • One or other side may have “died” leaving the other side waiting indefinitely
  – If probe fails the connection is terminated
  – Features
    • Adds overhead and may terminate an otherwise valid connection
    • Good for security and releasing resources which are no longer needed

• Other Timers
  – TIME_WAIT Timer: Waits for 2 MSL when TCP enters the TIME_WAIT state
  – FIN_WAIT2 Timer: Waits for other end to close their half of the connection
TCP Congestion Control

- Sender transmission rate
  - Governed by the rate at which ACKs are received
    - If per segment ACKs are received every $As$ seconds then $As$ segments will be transmitted on average
    - ACK arrival rate = MIN (receiver processing rate, overall network data rate)
  - Self-Clocking behaviour
    - The ACKs function as pacing signals
    - Automatically adjusts rate to cope with receiver flow control and network congestion effects

Figure 10.7 (hsn1e)
TCP Congestion Control

• Problem with Self-Clocking Behaviour on Network Congestion
  – Initial TCP connections have a WIN = MSS
    • Sender sends MSS segments as fast as it can
      – If enough new connections within the national or regional extent of a particular central router happen in a short interval of time → buffers temporarily full → network congestion
      – Network congestion is quick to occur but very slow to alleviate since the effects spread
    • Sender needs to slowly increase transmission rate to avoid the onset of congestion
  – Sender rate should be less than self-clocking rate when there is congestion
    • TCP sender needs to determine when there is congestion
      – IP provides no common mechanism for explicit notification of congestion
      – ICMP Source Quench is too crude; RSVP could help but is not yet widespread
      – Congestion detected by increased re-transmission rate and delayed ACKs

• Solution?
  – Actual transmission window for sender = MIN \((credit, cwnd)\)
    • credit: Receiver’s advertised window to prevent receiver buffer overflow
    • cwnd: Sender’s estimated congestion window to prevent network congestion
    • Window values in segments not octets (i.e. \(credit = \text{WIN} / \text{Segment Size}\))
TCP Congestion Control

• Slow-Start Algorithm
  – *RFC 2001*
  – A new TCP connection gradually increases transmit rate from 1 to full speed
    • Slow-Start is used by TCP sender to gauge network capacity and determine the optimum rate by dynamically adjusting $cwnd$
  – Calculating the value of $cwnd$
    • Initially set $cwnd = 1$ MSS and check whether segment is transmitted successfully
      – Segment is transmitted successfully if ACK arrives before re-transmission timer expires
    • **Exponential slow-start:** If transmission successful then $cwnd = 2 \cdot cwnd$ for each successful transmission of $cwnd$ segments ($cwnd$ incremented by 1 for each ACK received). This continues until $cwnd$ reaches a maximum value.
      – If $cwnd$ reaches credit then algorithm has done its job in determining that WIN does not create any network congestion
      – $cwnd$ grows exponentially up to $cwnd^{\text{max}}$ (upper bound on $cwnd$)
    • When re-transmission timer expires (e.g. connection goes “dead”):
      – Define a threshold $TH = cwnd / 2$
      – Set $cwnd = 1$ MSS and perform exponential slow-start until $cwnd = TH$
      – For $cwnd \geq TH$, increment $cwnd$ by 1 per RTT interval (linear slow-start) up to $cwnd^{\text{max}}$
TCP Congestion Control

- Example of Slow-Start Algorithm transmissions

Figure 10.10 (hsn1e)
TCP Congestion Control

- Example of Slow-Start Algorithm \(cwnd\) calculation

![Diagram showing TCP Congestion Control](image)

Figure 10.12 (hsn1e)
TCP Congestion Control

- **Fast Retransmit**
  - *RFC 2001*
  - Faster way for the sender to realise a segment has been lost
    - Normally the re-transmission timer expiry indicates a segment has been lost
    - RTO $>>$ RTT, if SDEV is large (RTT = min. time an ACK will take to arrive)
    - If sender waits RTO to retransmit segment receiver may have too many out-of-order packets (which cannot be removed until they are delivered to end-user in-order) and receive buffer may fill and subsequent packets dropped.
  - Relies on rule that the receiver ACKs all received segments when out-of-order
    - Receiver will ACK all out-of-order segments received by ACK’ing the most recent in-order segment (i.e. the lost segment is ACK’ed repeatedly)
      - With a full “pipe” of data due to a large sender window, a lost segment will create a “burst” of duplicate ACKs from the out-of-order segments still arriving from the “pipe”
      - Sender will then see a sequence of duplicate ACKs for the lost segment
    - Receiver will send a cumulative ACK for all in-order segments received when the lost segment is received
  - Sender realises a segment X is lost if duplicate ACK X segments are received
    - Typically 4 ACKs need to be received in case segment X has been delayed
    - Sender will re-transmit lost segment X when 4 ACKs (3 duplicates) for X are received
    - Normally results in segment X being retransmitted before RTO expiry
TCP Congestion Control

Example of Fast Retransmit transmissions

Figure 10.13 (hsn1e)
TCP Congestion Control

- Fast Recovery
  - *RFC 2001*
  - When a segment is re-transmitted due to a Fast Retransmit (not a normal RTO expiry) alternate congestion avoidance procedures should be invoked
    - Behaviour of standard slow-start when there is a time-out is unnecessarily conservative since it assumes the cause was possible congestion so $cwnd$ is reset to 1
    - The fact that duplicate ACKs are getting through implies the network is not as congested as would be the case with an RTO expiry
  - Modified $cwnd$ calculation when 3rd duplicate ACK arrives:
    - Eliminate the exponential slow-start phase which resets $cwnd = 1$ when a time-out (i.e. fast retransmit) occurs by halving $cwnd$ and starting the linear slow-start immediately:
      - Define a threshold $TH = cwnd / 2$
      - (Fast) Retransmit missing segment, and then set $cwnd = TH + 3$
        - The 3 takes into account the 3 extra segments sent that caused the 3 duplicate ACKs
      - Increment $cwnd$ by 1 for each additional duplicate ACK (linear slow-start)
      - Reset $cwnd = TH$ when the cumulative ACK (that acknowledges the lost segment, and all other in-order transmitted segments) arrives
      - Continue the normal linear slow-start of $cwnd = TH$ up to $cwnd_{max}$
Wireless TCP Issues

- TCP Congestion Control creates wireless problems
  - Lost segments are assumed caused by a congested network → slow sender rate
  - On wireless network lost segments are due to noisy transmission medium, not congestion of full queues → increase data rate to cut-through noise

- Solution 1: Indirect TCP
  - Allow different congestion control mechanism for wired links and wireless links
    - Sender → wired ← R1 → wireless ← Receiver
    - Sender → R1: connection with normal congestion control
    - R1 → Receiver: connection with wireless congestion control
  - Destroys basic TCP flow control concept since TCP flow control no longer end-to-end but link-to-link
    - Sender flow control now to R1 not the destination!

- Solution 2: Modify network layer
  - Network agent subsumes TCP activity and provides transparent handling of wireless data transfers
    - Base station to mobile host losses → Agent uses a shorter timer to retransmit segment
    - Duplicate ACKs sent from mobile host due to lost segment → Agent re-transmits segments and “absorbs” duplicate ACKs
    - Mobile host to Base station losses → Agent detects out-of-order transmissions and uses a TCP option for a selective repeat of the missing segment or bytes
TCP over ATM

TCP Segment
TCP header
20 octets

IP Datagram
IP header
20 octets

CS PDU

ATM Cell
SDU = 0
48 octets
53 octets
SDU = 0

Protocol Stack
TCP
IP
AAL5
ATM

SDU = 0
SDU = 0
SDU = 1
TCP over ATM

• TCP over UBR
  – Unspecified Bit Rate (No congestion control feedback information provided)
  – Poor performance due to cascade effect of dropping a single ATM cell
    • All ATM cells which are part of the same IP datagram are useless once any one cell
      is discarded, but such cells are forwarded through the network regardless
  – Some solutions (requires modifying operation of ATM switch)
    • Partial Packet Discard (PPD)
      – If a cell is dropped all subsequent cells are also dropped
      – Easy to implement but on average half of the useless cells are still forwarded
    • Early Packet Discard (EPD)
      – Before switch buffer overflows all cells associated with an entire IP datagram are dropped
      – Similar behaviour to packet-switched networks
      – Unfair treatment: biased against short IP datagrams (these are more likely to be dropped
        since there are more of them); biased against connections that pass through multiple
        switches (more likely that a packet will be dropped at any one switch)

• TCP over ABR
  – Available Bit Rate (Congestion control feedback provided)
  – More difficult to analyse
UDP Protocol: Header Format

- UDP header and functions
  - RFC 768
  - Source / Destination Port [16]
  - Length [16]
    - Length of UDP header and user data, maximum size is 65,535 bytes
  - Checksum [16]
    - Same as TCP but optional (TCP checksum is mandatory)
      - If unused then checksum = 0
        (note that the checksum is all 1’s if the true checksum is 0 (1’s complement))
  - Data [variable]
UDP Protocol: Data Transfer

• Connectionless, Unreliable but minimal overheads
  – No need to establish/terminate connection
  – Lost UDP data is not re-transmitted
  – No congestion control, flow control and optional error control
  – User can send up to 64 kB in the one UDP datagram
    • IP fragmentation possible, fragmentation is bad → highly variable performance
    • TCP avoids fragmentation by using an MSS which corresponds to the MTU

• Uses?
  – One-off data transfers
    • No connection establishment/termination overheads
      – Remote Procedure Call (RPC)
      – Client-server Request/Response interactions
  – Some aspect of TCP functionality needs to be disabled or prevented
    • Real-time (multimedia) data → No re-transmissions and No slow-start