Contents

Transport Layer **3**

Transport Services and Protocols **3**

Internet Transport-Layer Protocols . . . 3

Multiplexing/Demultiplexing **4**

Addresses . . . 4

How Demultiplexing Works . . . 4

Connection Orientated Demux . . . 4

User Datagram Protocol (UDP) **5**

Header . . . 5

Principles of Reliable Data Transfer **6**

RDT . . . 6

rdt1.0: Reliable Transfer Over a Reliable Channel . . . 6

rdt2.0 . . . 7

rdt2.1: Discussion . . . 8

rdt2.2: NAK-free Protocol . . . 8

rdt3.0: Channels with Errors and Loss . . . 8

rdt3.0 . . . 9

Pipelined Protocols **9**

Overview . . . 9

Go-back-N . . . 9

Sender . . . 9

Receiver . . . 10

Selective Repeat . . . 10

Sender . . . 10

Receiver . . . 10

TCP (Overview) **11**

Header 11

Seq Numbers & ACKs 12

Round Trip Time (RTT) 12

Average RTT 12

Variations in RTT (DevRTT) 13

Reliable Data Transfer 14

Sender Events 14

ACK Generation 15

Fast Retransmit 15

Flow Control 15

Connection Management **16**

TCP 3-way Handshake 16

Opening 16

Closing 17

SYN Flood Attack (DoS) 17

SYN Cookies 17

Principles of Congestion Control **17**

Causes/Costs 18

Scenario 1 18

Scenario 2 18

Scenario 3 19

Approaches Towards congestion Control 19

ATM ABR Congestion Control 19

TCP Congestion Control Details 20

TCP **21**

Slow Start 21

Detecting and Reacting to Loss 21

Switching from Slow Start to CA 21

Securing 21

Transport Layer

- Transport Layer Services
- Multiplexing and Demultiplexing
- Connectionless Transport: UDP
- Connection-Oriented Transport: TCP
- TCP Congestion Control

Transport Services and Protocols

- Provide *logical communication* between app processes running on different hosts
  - Header
  - Intermediate routers don't run transport layer (Only physical, data link and network)
  - End points notice if there's any missing information
- Transport protocols run in end systems
  - Send side: breaks app messages into *segments*, passes to network layer
    * Packet loss would use up lots of bandwidth
    * Maximum Transfer Unit (MTU)
  - Rcv side: reassembles *segments* into messages, passes to app layer
- More than one transport protocol available to apps
  - Internet: TCP and UDP

Internet Transport-Layer Protocols

- TCP - Reliable, in-order delivery
  - Congestion Control
  - Flow Control
  - File transmission
  - Connection Setup
- UPD - Unreliable, unordered delivery
  - No frills extension of "best effort" IP
  - Video calls
- Services not available
  - Delay guarantees
  - Bandwidth guarantees

Multiplexing/Demultiplexing

- Multiplexing at sender:
  - Handle data from multiple sockets, add transport header (later used for demultiplexing)
- Demultiplexing at receiver
  - User header info to deliver received segments to correct socket

Addresses

- MAC Address
  - Physical Layer
  - Wired into wireless card
- IP Address
  - Network Layer
- Socket Address (Port number)
  - Software Address
  - Differentiate which packet goes to which process
  - In TCP, negotiated in setup
  - In UDP, must already know

How Demultiplexing Works

- Host receives IP datagrams
  - Each datagram has source IP address, destination IP address
  - Each datagram carries one transport layer segment
  - Each segment has source, destination port number
- Host uses *IP address* & *port numbers* to direct segment to appriopriate socket

Connection Orientated Demux

- TCP socket identified by 4-tuple (You need this data to find the right socket in a machine)
  - Source IP address
  - Source port numbers
  - Dest IP address

  - Dest port number
- Receiver uses all four values to direct segment to appropriate socket
  - Demultiplexing
- Server host may support many simultaneous TCP sockets
  - Each socket identifier by its own 4-tuple
- Web servers have different sockets for each connecting client
  - Non-persistent HTTP will have a different socket for each request
    * A nonpersistent connection is the one that is closed after the server sends the requested object to the client
    * In other words, the connection is used exactly for one request and one response

User Datagram Protocol (UDP)

- "No frills", "bare bones" Internet transport protocol
- "Best effort" service, UDP segments may be
  - Lost
  - Delivered out-of-order to app
- Connectionless
  - Speed
  - Each UDP segment handled independently of others
- Users
  - Streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP

Header

- No connection establishment
  - Which can add delay
- Simple - No connection state
  - At sender, receiver
- Small header size
- No congestion control
  - UDP can blast away as fast as desired
  - Can pump as much data into the network as possible

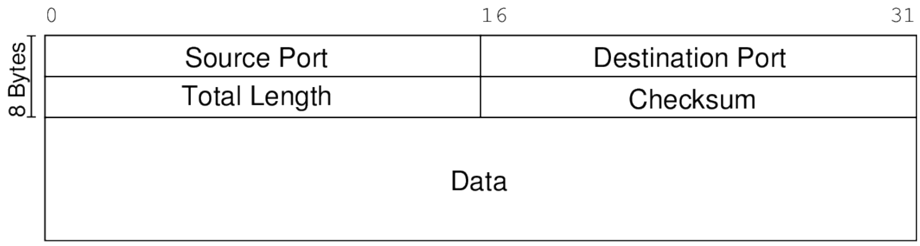

Figure 1: UDP segment format

Principles of Reliable Data Transfer

- Important in application, transport, link layers
- Characteristics of unreliable channel will determine complexity of *reliable data tranfer* protocol (rdt)

1. `rdt_send()`: called from above (e.g. by app), Passed data to deliver to receiver upper layer
2. `udt_send()`: called by rdt, to transfer packet over unreliable channel to receiver
    - Passes through unreliable channel
    - Bits might get corrupted, packets could get lost, etc.
3. `rdt_rcv()`: called when packet arrives on rcv-side of channel
4. `deliver_data()`: called by **rdt** to deliver data to upper

RDT

- We will incrementally develop sender, receiver sides of reliable data transfer protocol
- Consider only unidirectional data transfer
    - But control info will flow in both directions!
- Use finite state machines (FSM) to specify sender, receiver

rdt1.0: Reliable Transfer Over a Reliable Channel

- Underlying channel perfectly reliable
    - No bit errors
    - No loss of packets
- Separate FSMs for sender and receiver

    - Sender sends data into underlying channel
    - Receiver reads data from underlying channel

rdt2.0

Channel with Bit Errors

- Underlying channel may flip bits in packet
    - Checksum to detect bit errors
    - Cyclic redundancy checking (CRC) tells us bits were flipped but not which ones were flipped
- Q: How to recover from erors?
    - Acknowledgements (ACKs)
    - Receiver explicitly tells sender that pkt received OK
- Negative Acknowledgements (NAKs)
    - Receiver explicitly tells sender that pkt had errors
        * Sender retransmits pkt on receipt of NAK
- New mechanism in rdt2.0 (beyond rdt1.0)
    - Error detection
    - Feedback
        * Control msgs (ACK, NAK) from receiver to sender

Fatal Flaw

- What happens if ACK/NAK corrupted
    - Sender does not know what happened to receiver!
    - Cannot just retransmit
        * Possible duplicate
- Handling Duplicates
    - Sender retranmits current pkt if ACK/NAK corrupted
    - Sender add sequence number to each packet
    - Receiver discards (does not deliver up) duplicate pkt
- Stop and Wait
    - Sender sends one packet, then waits for receiver response

rdt2.1: Discussion

- Sender
  - seq # added to pkt
  - Two seq #'s (0, 1) will suffice - why?
  - Must check if received ACK/NAK corrupted
  - Twice as many states
    * State must "remember" whether "expected" pkt should have seq # of 0 or 1
- Receiver
  - Must check if received packet is duplicate
  - Note: receiver cannot tell if its last ACK/NAK received OK at sender

rdt2.2: NAK-free Protocol

- Same functionality as rdt2.1
  - Using ACKs only
- Instead of NAK, receiver sends ACK for last pkt receiver OK
  - Receiver must explicitly include seq # of pkt being ACKed
- Duplicate ACK at sender results in same action as NAK
  - Retransmit current pkt

rdt3.0: Channels with Errors and Loss

- New Assumption
  - Underlying channel can also lose packets (data, ACKs)
  - Checksum, seq #, ACKs, retransmissions will be of help
    * But not enough
- Approach
  - Retransmits if no ACK received in this time
  - If pkt (or ACK) just delayed (not lost)
    * Retransmission will be duplicate, but seq #'s already handle this
    * Receiver must specify seq # of pkt being ACKed
  - Requires countdown timer

rdt3.0

- rdt3.0 performance stinks
  - e.g. ! Gbps link (R), 15 ms prop. delay, 8000 bit packet (L)
- Effective throughput of 267 kbps over a 1 Gbps link

Pipelined Protocols

- Pipelining: sender allows multiple “in-flight”, yet-to-be-acknowledged pkts
  - Range of sequence numbers must be increased
  - Buffering at sender and/or receiver
- Two generic forms of pipelined protocols
  - Go-back-N
  - Selective Repeat

Overview

- Go-back-N (GBN)
  - Sender can have up to `N` unacked packets in the pipeline
  - Receiver can send cumulative acks
  - Sender has timer for oldest acked packet
    * When timer expires, retransit all unacked packets
- Selective Repeat (SR)
  - Sender can have up to `N` unacked packets in pipeline
    * Receiver sends individual ack for each packet
  - Sender maintains timer for each unacked packet
    * When timer expires, retransmit only that unacked packet
  - Less bandwidth used in selective repeat, may be preferable over GBN

Go-back-N

Sender

- k-bit seq # in pkt header
  - “Window” of up to `N` consequtive unacked pkts allowed
- Timer for oldest in-flight pkt
  - Timeout(n): Retransmit packet `n` and all higher seq # pkts that have been previously sent in window

Receiver

- ACK-only: always send ACK for correctly-received pkt with highest in-order seq #
  - Need only remember **expected seq num**
- Out-of-order pkt
  - Discard: no receiver buffer!
    * Re-ACK pkt with highest in-order seq #

Selective Repeat

- Receiver individually acknowledges all correctly received pkts
  - Buffers pkts, as needed, for eventual in-order delivery to upper layer
- Sender only resends pkts for which ACK not received
- Sender window
  - `N` consecutive seq #'s
  - Limits seq #s of sent, unACKed pkts

Sender

- Data from above
  - If next available seq # in window, send pkt
- Timeout(n)
  - Resend pkt `n`, restart timer
- ACK(n) in [sendbase, sendbase+N]
  - Mark pkt `n` as received
  - If `n` smallest unACKed pkt (seq # == send-base), advance window base to next unACKed seq #

Receiver

- Pkt `n` in [rscbase, rcvbase+N-1]
  - Send ACK(n)
  - Out-of-order: buffer
  - In-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt
- Pkt `n` in [rcvbase-N, rcvbase-1]

  - Ack(n) - previously acknowledged pkt
- Otherwise
  - Ignore

TCP (Overview)

- Point-to-point
  - One sender, one receiver
- Reliable, in order byte stream
  - No "message boundaries"
  - Streaming protocol
- Pipelined
  - TCP congestion and flow control set window size
- Full duplex data
  - Bi-directional data flow in same connection
  - MSS: maximum segment size
- Connection-orientated
  - Handshaking (exchange of control msgs) initialise sender, receiver state before data exchange
- Flow controlled

Header

- Control Flags
  - URG: Urgent data (generally not used)
  - ACK: ACK # valid
  - PSH: Push data now (generally not used)
  - RST, SYN, FIN: connection estab (setup, teardown commands)
- Checksum: Internet Checksum
- Sequence/Acknowledgement number: Counting by bytes of data (not segments)
- Receive window: # bytes rcvr willing to accept

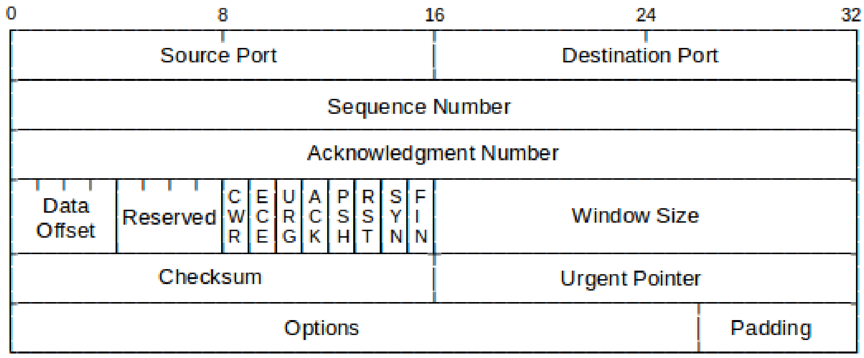

Figure 2: TCP Header

Seq Numbers & ACKs

- Sequence Numbers
  - Byte stream "number" of first byte in segment's data
- Acknowledgements
  - Seq # of next byte expected from other side

Round Trip Time (RTT)

- How to set TCP timeout value to recover from lost segments?
  - Longer than RTT - but RTT varies
- How to estimate RTT?
  - SampleRTT: measured time from segment transmission until ACK receipt
    * Ignore retransmissions
  - SampleRTT will vary due to congestion and load at routers
    * Want EstimatedRTT - smooth
      · Average several recent measurements, not just current SampleRTT

Average RTT

$EstimatedRTT = (1 - \alpha) * EstimatedRTT + \alpha * SampleRTT$

- Exponential weighted moving average
- Influence of past sample decreases expontentially fast
- Typical value: $\alpha = 0.125$

Pkt #	SampleRTT	EstimatedRTT $\alpha = 0.125$
20		40
21	30	39
22	80	44
23	40	43
24	90	49
25	50	?

- What happens if α is too small (say very close 0)
  - May lead to under or over estimation of RTT for a long time
- What happens if α is too large (say very close 1)
  - Transient fluctuations/changes in network load affects EstimatedRTT and makes it unstable when it should not
    - Also leads to under or over estimation of RTT

Variations in RTT (DevRTT)

- Timeout Interval: `EstimatedRTT` plus "safety margin"
- Estimate SampleRTT deviation from EstimatedRTT

$$DevRTT = (1 - \beta * DevRTT + \beta \mid SampleRTT - EstimatedRTT \mid)$$

(typically, $\beta = 0.25$)

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

- For TCP, initial value of TimeoutInterval = 1

Pkt #{}	Sample RTT	EstimatedRTT $\alpha = 0.125$	DevRTT $\beta = 0.25$
20		40	10
21	30	39	10
22	80	44	18
23	40	43	14
24	90	49	20
25	50	54	25

Reliable Data Transfer

- TCP creates rdt service on top of IP's unreliable service
  - Pipelined Segments
  - Cumulative ACKs
  - Single Retransmission Timer
- Retransmissions triggered by
  - Duplicate ACKs
- Let us initially consider simplified TCP sender
  - Ignore Duplicate ACKs
  - Ignore Flow Control & Congestion Control

Sender Events

- Data rcvd from app
  - Create segment with seq #
- Start timer if not already running
  - Think of timer as for oldest unacked segment
    * Expiration interval: TimeOutInterval
- Timeout
  - Retransmit segment that caused timeout
    * Restart timer
- ACK rcvd
  - If ACK acknowledges previously unacked segments
    * Update what is known to be ACKed
    * Start timer if there are still unacked segments

ACK Generation

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

Fast Retransmit

- Time-out period often relatively long
  - Long delay before resending lost packet
- Detect lost segments via duplicate ACKs
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs
- TCP Fast Retransmit
  - If sender receives 3 duplicate ACKs (i.e., on the 4^{th} ACK) from same data - "Triple Duplicate ACKs"
    * Resend unacked segment with smallest sequence number
    * Number of duplicates to wait for before retransmission is configurable
  - Likely that unacked segment lost, so do not wait for timeout

Flow Control

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

- Receiver "advertises" free buffer space by including *rwnd* (receive window) value in TCP header of recaiver-to-sender segments

- RcvBuffer size set via socket options
  - Typical default is 4096 bytes
- Many operaying systems auto adjust the RcvBuffer

```
rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]
```

- Sender limits amount of unacked ("in-flight") data to receiver's *rwnd* value
  - Guarantees receive buffer will not overflow

Connection Management

- Before exchanging data, sender/receiver "handshake"
  - Agree to establish connection (each knowing the other willing to establish a connection)
  - Agree on connection parameters

TCP 3-way Handshake

Opening

1. Client State: $LISTEN \rightarrow SYNSENT$
   - $SYNbit = 1$
   - $Seq = x$
2. Server State: $LISTEN \rightarrow SYNRCVD$
   - $SYNBit = 1$
   - $Seq = y$
   - $ACKbit = 1$
   - $ACKnum = x + 1$
3. Client State: $SYNSEND \rightarrow ESTAB$
   - $ACKbit = 1$
   - $ACKnum = y + 1$
   - Can piggyback user data at this point
4. Server State: $SYNRCVD \rightarrow ESTAB$

Closing

1. Client State: $ESTAB \rightarrow FIN\_WAIT_1$
   - $FINbit = 1$
   - $Seq = x$
2. Server State: $ESTAB \rightarrow CLOSE\_WAIT$
   - ACKbit=1
   - ACKnum=x+1
3. Client State $FIN\_WAIT_1 \rightarrow FIN\_WAIT_2$
4. Server State: $CLOSE\_WAIT \rightarrow LAST\_ACK$

SYN Flood Attack (DoS)

- Attacker sends large number of TCP SYN segments
  - Without completing the third handshake step

SYN Cookies

- Server does not know if the SYN segment is coming from a legitimate user
- Server creates an initial sequence number (ISN) or "cookie" from the hash of
  - Src IP addr & Port
  - Dest IP addr & Port
  - Timestamp
- Server sends SYNACK
  - Maintains no state info about corresponding to the SYN
- A legitimate client will return an ACK segment
  - Use the cookie information (ISN+1) in the ACK

Principles of Congestion Control

- Informally:
  - "Too many sources sending too much data too fast for network to handle"
    * Different from flow control!
- Manifestations

  - Lost packets
    * Buffer overflow at routers
  - Long delays
    * Queuing in router buffers

Causes/Costs

Scenario 1

- Two senders, two receivers
- One router, infinite buffers
- Output link capacity: R
- No retransmission
- Maximum per-connection throughput: R/2
- Large queuing delays are experienced as the pkt arrival rate, λ_{in}, approaches capacity

Scenario 2

- One router, *finite* buffers
  - Pkts will be dropped
- Idealization: *perfect knowledge*
  - Sender sends only when router buffers available
- Idealization: *known loss*
  - Packets can be lost, dropped at router due to full buffers
  - Sender only resends if packet known to be lost
- Realistic: *duplicates*
  - Packets can be lost, dropped at router due to full buffers
  - Sender times out permaturely, sending *two* copies, both of which are delivered
- "Costs of Congestion"
  - More work (retrans) for given "goodput"
  - Unneeded retransmissions
    * Link carries multiple copies of pkt
      · Decreasing goodput

Scenario 3

- Four Senders (blue send/rcv and red send/rcv)
- Multihop Paths
- Timeout/Retransmit

Q: What happens as λ_{in} and λ'_{in} increase?

A: As red λ'_{in} increases, all arriving blue pkts at upper queue are dropped, blue throughput $\rightarrow 0$

- Another "cost" of congestion
  - When packet dropped, any upstream transmission capacity used for that packet is wasted

Approaches Towards congestion Control

- End-to-End Congestion Control
  - No explicit feedback from network
  - Congestion inferred from end-system observed loss, delay
  - Approach taken by TCP
- Network-Assisted Congestion Control
  - Routers provide feedback to end systems
  - Single bit indicating congestion (ATM - Asynchronous Transfer Mode)
  - Explicit rate for sender to send at

ATM ABR Congestion Control

ABR: Available Bit Rate

- "Elastic Service"
- If sender's path is underloaded
  - Sender should use available bandwidth
- If sender's path is congested
  - Sender throttled to minimum guaranteed rate

RM (Resource Management) Cells

- Sent by sender, interspersed with data cells

- Bits in RM cell set by switches (“network assisted”)
  - NI bit: no increase in rate (mild congestion)
  - CI bit: Congestion indicated
- RM cells returned to sender by receiver, with bits intact

Two-byte ER (explicit rate) field in RM cell

- Congested swith may lower ER value in cell

EFCI (explicit forward congestion indication) bit in data cells: set to 1 in congested switch

- If data cell preceding RM cell has EFCI set, receiver sets CI bit in returned RM cell

Additive Increase Multiplicative Decrease (AIMD)

- Sender increases transmission rate (window size)
  - Probing for usable bandwidth, until loss occurs
- Additive Increase
  - Incrase congestion window (*cwnd*) by 1 MSS every RTT until loss detected
- Multiplicative Decrease
  - Cut *cwnd* in half after loss

TCP Congestion Control Details

- Sender limits transmission

```
LastByteSent - LastByteAcked <= cwnd
```

- *cwnd* is dynamic, function of perceived network congestion

TCP sending rate

- Send *cwnd* bytes
- Wait RTT for ACKs
- Then send more bytes

TCP

Slow Start

- When connection begins, increase rate exponentially until first loss event
  - Initially *cwnd* = 1 MSS
  - Double *cwnd* every RTT
  - Done by incrementing *cwnd* for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast

Detecting and Reacting to Loss

- Loss indicated by timeout
  - *cwnd* set to 1 MSS
- Window then grows exponentially (as in slow start) to threshold (ssthresh)
  - Then grows linearly
- Loss indicated by 3 duplicate ACKs: TCP Reno
  - Dup ACKs indicate network capable of delivering some segments
  - *cwnd* is cut in half window then grows linearly
- TCP Tahoe always set *cwnd* to 1
  - Timeout or 3 duplicate ACKs

Switching from Slow Start to CA

Q: When should the exponential increase switch to linear?

A: When *cwnd* gets to 1/2 of its value before timeout

- Implementation
  - Variable ssthresh

Securing

- TCP and UDP
  - No encryption
- SSL

  - Provides encrypted TCP connection
  - Data integrity
- SSL is at app layer
  - Apps use SSL libraries which talk to TCP
- SSL socket API
  - Cleartext passwords sent into socket traverse internet encrypted