CS4450

Computer Networks: Architecture and Protocols

Lecture 20
Reliable Transport and TCP

Rachit Agarwal
Goal of Today’s Lecture

• Continue our understanding of reliable transport conceptually

• Understanding TCP will become infinitely easier
  • TCP involves lots of detailed mechanisms
  • Knowing WHY TCP uses these mechanisms is most important
Let's start with recapping last lecture
Recap: Best Effort Service (L3)

- Packets can be **lost**
- Packets can be **corrupted**
- Packets can be **reordered**
- Packets can be **delayed**
- Packets can be **duplicated**
- ...

**How can you possible make anything work with such a service model?**
Recap: Four Goals for Reliable Transfer

• Correctness
  • As defined

• “Fairness”
  • Every flow must get a fair share of network resources

• Flow Performance
  • Latency, jitter, etc.

• Utilization
  • Would like to maximize bandwidth utilization
  • If network has bandwidth available, flows should be able to use it!
Recap: Complete Correctness Condition

A transport mechanism is “reliable” if and only if
(a) It resends all dropped or corrupted packets
(b) It attempts to make progress
Recap: Solution v1

• Send every packet as often and fast as possible...

• Not correct
  • if condition not satisfied: Transport must attempt to make progress
  • No way to check whether the packet was dropped or corrupted
    • So, must continue sending the same packet
  • Showed why we need receiver feedback
Recap: Solution v2

• Resend packet until you get an ACK
  • And receiver sends per-packet ACKs until data finally stops

• Correct
• Fair
• Good but suboptimal performance
• Suboptimal utilization
  • A specific kind of under-utilization:
    • The source is unnecessarily sending the same packet
  • Showed why we must wait for an ACK after sending a packet
    • But how long shall we wait for an ACK?
    • Indeed, the ACK may be lost as well
Recap: Solution v3

• Send packet
  • But now, **set a timer**
• receiver sends per-packet ACKs
• If sender receives ACK, done
• If no ACK when timer expires, resend

• Correct
• Fair
• Good but suboptimal performance
• Suboptimal utilization
  • **A different kind of under-utilization**
    • source is not “work conserving”: could send, but is not
• What to do while waiting?
  • Send more packets
  • How many?
Window-based Algorithms

• Very simple concept
  • Send $W$ packets
  • When one gets ACK’ed send the next packet in line

• We want to set $W$ such that:
  • if I am sending at rate = link bandwidth, then
  • the ACK of the first packet arrives
  • exactly when I just finish sending the last of my $W$ packets
  • (assuming same transmission time for data and ACK packets)

• Lets me send as fast as the path can deliver…
Recall that **Bandwidth Delay Product**

- \( \text{BDP} = \text{bandwidth} \times \text{propagation delay} \)

- \( B \times \text{RTT} \) is merely \( 2 \times \text{BDP} \)

**Window sizing rule:**

- Total bits in flight is roughly the amount of data that fits into forward and reverse “pipes”
  - Here pipe is complete path, not single link...
  - This is not “detail”, this is a fundamental concept...
Where Are We?

• Figured out correctness condition:
  • Always resend lost/corrupted packets
  • Always try to make progress (but can give up entirely)

• Figured out single packet case:
  • Send packet, set timer, resend if no ACK when timer expires

• Some progress towards multiple packet case:
  • Allow many packets (W) in flight at once
  • And know what the ideal window size is
    • RTT x B / Packet size

• What’s left to design?
Three Design Considerations

• Nature of feedback
  • What should ACKs tell us when we have many packets in flight

• Detection of loss

• Response to loss
ACK Individual Packets

The receiver sends ACK for each individual packet that it receives

Example:

• Assume that packet 5 is lost, but no others

• Stream of ACKs will be
  • 1
  • 2
  • 3
  • 4
  • 6
  • 7
  • 8
  • ...

ACK Individual Packets

- **Nature of feedback**: simple - the receiver ACKs each packet

- **Loss detection**: simple - ACKs tell the fate of each packet to the source

- **Response to loss**: *moderate*:
  - + Retransmit the packet for which ACK not received
  - + Reordering not a problem
  - + Simple window algorithm
    - W independent single packet algorithms
    - When one finishes grab next packet

- - Loss of ACK packet requires a retransmission
Full Information Feedback

• List all packets that have been received
  • Give highest cumulative ACK plus any additional packets

Same Example (suppose packet 5 gets lost):

• Same story, except that the “hole” is explicit in each ACK

• Stream of ACKs will be
  • Up to 1
  • Up to 2
  • Up to 3
  • Up to 4
  • Up to 4, plus 6
  • Up to 4, plus 6,7
  • Up to 4, plus 6,7,8
  • ...
  • ...
Full Information Feedback

- **Nature of feedback:** complex - feedback may have high overheads
  - If packets 1, 5, 6, ..., 100 received: ACK(1, 5, 6, ... ,100)

- **Loss detection:** simple - the source still knows fate of each packet

- **Response to loss:** simple:
  - + Retransmit the packet for which ACK not received
  - + Reordering not a problem
  - + Simple window algorithm

- **- Loss of ACK does not necessarily requires a retransmission**
  - The next ACK will tell that the packet was indeed received
  - Resilient form of individual ACKs
Cumulative ACK

• Individual ACKs can get lost, and require unnecessary retransmission

• Full information feedback can handle lost ACKs but has high overheads

• Cumulative ACKs: a sweet spot between the two

• Just the first part of full information feedback

• ACK the highest sequence number for all previously received packets
Cumulative ACKs (same example; say packet 5 lost)

Full information feedback:

• Stream of ACKs will be
  • Up to 1
  • Up to 2
  • Up to 3
  • Up to 4
  • Up to 4, plus 6
  • Up to 4, plus 6,7
  • Up to 4, plus 6,7,8
  • ...

Tells “which” packet arrived, and which packet did not

Cumulative ACKs:

• Stream of ACKs will be
  • Up to 1
  • Up to 2
  • Up to 3
  • Up to 4
  • Up to 4
  • Up to 4
  • Up to 4
  • ...

Tells “some” packet arrived, and which packet did not
Loss With Cumulative ACKs (cont’d)

• Duplicate ACKs are a sign of loss
  • The lack of ACK progress means 5 hasn’t been delivered
  • Stream of duplicate ACKs means some packets are being delivered (one for each subsequent packet)

• Response to loss is trickier... When shall the source retransmit packet 5?
  • Packet may be delayed (so, source should wait)
  • Packet may be reordered (so, source should wait)
  • Or, packet may be dropped (source should immediately retransmit)
  • Impossible to know which one is the case
    • Life lesson: be optimistic!
    • Until optimism starts hurting

• Solution: retransmit after k duplicate ACKs
  • for some value of k, depending on how optimistic you feel!
Cumulative ACKs (how is reordering handled; large k)

Receiver events:

• Packet 1 received
• Packet 2 received
• Packet 3 received
• Packet 4 received
• Packet 6 received
• Packet 7 received
• Packet 5 received
• Packet 8 received
• ...

Cumulative ACKs:

• Up to 1
• Up to 2
• Up to 3
• Up to 4
• Up to 4
• Up to 4
• Up to 7
• Up to 8
• ...

Cumulative ACKs naturally handle packet reordering
(Packet delays are similar to reordering)
Cumulative ACKs (confusion with duplication)

- Produce duplicate ACKs
  - Could be confused for loss with cumulative ACKs
  - But duplication is rare...

<table>
<thead>
<tr>
<th>Source events:</th>
<th>Receiver events:</th>
<th>Cumulative ACKs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet 1 sent</td>
<td>Packet 1 received</td>
<td>Up to 1</td>
</tr>
<tr>
<td>Packet 2 sent</td>
<td>Packet 2 received</td>
<td>Up to 2</td>
</tr>
<tr>
<td>Packet 3 sent</td>
<td><strong>Packet 4 received</strong></td>
<td>Up to 2</td>
</tr>
<tr>
<td>Packet 4 sent</td>
<td>Packet 5 received</td>
<td>Up to 2</td>
</tr>
<tr>
<td>Packet 5 sent</td>
<td>Packet 6 received</td>
<td>Up to 2</td>
</tr>
<tr>
<td>Packet 6 sent</td>
<td><strong>Packet 3 received</strong></td>
<td>Up to 6</td>
</tr>
<tr>
<td><strong>Packet 3 resent</strong></td>
<td>Packet 3 received</td>
<td><strong>Up to 6</strong></td>
</tr>
<tr>
<td>Packet 7 sent</td>
<td>Packet 7 received</td>
<td>Up to 7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Possible Design For Reliable Transport

- Cumulative ACKs

- Window based, with retransmissions after
  - Timeout
  - k subsequent ACKs

- This is correct, high-performant and high-utilization
  - At least as much as we can efficiently

- How about fairness?
Fairness? (Come back to later)

• The question of fairness comes up when:
  • Senders want to send data at rate higher than bandwidth
  • There will be packet loss!

• Adjust $W$ based on losses...

• In a way that flows receive same shares

• Short version:
  • Loss: cut $W$ by 2
  • Successful receipt of window: $W$ increased by 1
Overview of Reliable Transport

• Window based self control separate concerns
  • Size of W
  • Nature of feedback
  • Response to loss

• Can design each aspect relatively independently

• Can be correct, fair, high-performant and high-utilization

• All of these are important concerns
  • But correctness is most fundamental

• Design must start with correctness
  • Can then “engineer” its performance with various hacks
  • These hacks can be “fun”, but don’t let them distract you
What Have We Done so far?

• Started from first principles
  • Correctness condition for reliable transport

• ... to understanding **why feedback from receiver is necessary** (sol-v1)

• ... to understanding **why timers may be needed** (sol-v2)

• ... to understanding **why window-based design may be needed** (sol-v3)

• ... to understanding **why cumulative ACKs may be a good idea**
  • Very close to modern TCP

• You are now ready to learn TCP
Let's learn TCP
Transport layer

- Transport layer offer a “pipe” abstraction to applications
- Data goes in one end of the pipe and emerges from other
- Pipes are between processes, not hosts
- There are two basic pipe abstractions
Two Pipe Abstractions

- **Unreliable packet** delivery (UDP)
  - Unreliable (application responsible for resending)
  - Messages limited to single packet

- **Reliable byte stream** delivery
  - Bytes inserted into pipe by sender
  - They emerge, in order at receiver (to the app)

- What features must transport protocol implement to support these abstractions?
**UDP (Datagram Messaging Service)**

- Sources send packets

- **Destinations do nothing**, but receive packets

- If packets delayed/reordered/lost:
  - Meh!
  - Let application handle packet loss (or be oblivious to drops)
  - If application needs reliable delivery, it must use reliable transport

- Discarding corrupted packets (optional)

- Nothing else!

- A minimal extension of IP
TCP (Reliable, In Order Delivery)

- Source send **segments**
- Destinations send ACKs
- Source retransmits lost and/or corrupted **segments**
- Sources perform **Flow control** (to not overflow receiver)
- Sources perform **Congestion control** (to not overload network)
- Source and destination participate in “Connection” set-up and tear-down
Connections (Or Sessions)

• Reliability requires keeping state
  • Sender: packets sent but not yet ACKed, and related timers
  • Receiver: packets that arrived out-of-order

• Each byte stream is called a **connection** or **session**
  • Each with their own connection state
  • State is in hosts, not network
Ports

• Separate 16-bit port address space for UDP, TCP

• “Well known” ports (0-1023)
  • Agreement on which services run on these ports
  • e.g., ssh:22, http:80
  • Client (app) knows appropriate port on sender
  • Services can listen on well-known ports
Multiplexing and Demultiplexing

- Host receives IP datagrams
  - Each datagram has source and destination IP address
  - Each segment has source and destination port number
- Host uses IP address and port numbers to direct the segment to appropriate socket

<table>
<thead>
<tr>
<th>Source Port #</th>
<th>Dest Port #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Other Header Fields</td>
</tr>
<tr>
<td></td>
<td>Application Data (Message)</td>
</tr>
</tbody>
</table>
IP Packet Structure

- 4-bit Version
- 4-bit Header Length
- 8-bit Type of Service (TOS)
- 16-bit Total Length (Bytes)
- 16-bit Identification
- 3-bit Flags
- 13-bit Fragment Offset
- 8-bit Time to Live (TTL)
- 8-bit Protocol
- 16-bit Header Checksum
- 32-bit Source IP Address
- 32-bit Destination IP Address
- Options (if any)
- Payload
### IP Packet Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-bit Version</td>
<td>Version of the IP protocol version.</td>
</tr>
<tr>
<td>4-bit Header Length</td>
<td>Length of the header in 32-bit words.</td>
</tr>
<tr>
<td>8-bit Type of Service</td>
<td>Type of Service (TOS)</td>
</tr>
<tr>
<td>16-bit Total Length</td>
<td>Total length of the packet in 32-bit words</td>
</tr>
<tr>
<td>16-bit Identification</td>
<td>Identification number for fragment tracking</td>
</tr>
<tr>
<td>3-bit Flags</td>
<td>Flags for fragment handling</td>
</tr>
<tr>
<td>13-bit Fragment Offset</td>
<td>Offset of fragment in the total packet</td>
</tr>
<tr>
<td>8-bit Time to Live</td>
<td>Time to Live (TTL)</td>
</tr>
<tr>
<td>16-bit Header Checksum</td>
<td>Checksum of the header</td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td>Source IP address</td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td>Destination IP address</td>
</tr>
<tr>
<td>16-bit Source Port</td>
<td>Source port of the transport protocol</td>
</tr>
<tr>
<td>16-bit Dest Port</td>
<td>Destination port of the transport protocol</td>
</tr>
</tbody>
</table>

More Transport Header Fields...

Payload
Transmission Control Protocol (TCP)

- Reliable, in-order delivery
  - Ensures byte stream (eventually) arrives intact
  - In the presence of corruption, delays, reordering, loss

- Connection oriented
  - Explicit set-up and tear-down of TCP session

- Full duplex stream of **byte service**
  - Sends and receives stream of bytes, not messages

- Flow control
  - Ensures the sender does not overwhelm the receiver

- Congestion control
  - Dynamic adaptation to network path’s capacity
From design to implementation: major notation change

• Previously we focused on packets
  • Packets had numbers
  • ACKs referred to those numbers
  • Window sizes expressed in terms of # of packets

• TCP focuses on bytes, thus
  • Packets identified by the bytes they carry
  • ACKs refer to the bytes received
  • Window size expressed in terms of # of bytes
Basic Components of Reliability

• ACKs
  • TCP uses byte sequence numbers to identify payloads
  • ACKs referred to those numbers

• Timeouts and retransmissions
  • Can’t be reliable without retransmitting lost/corrupted data
  • TCP retransmits based on timeouts and duplicate ACKs
  • Timeouts based on estimate of RTT
Other TCP Design Decisions

- Sliding window flow control
  - Allow $W$ contiguous bytes to be in flight

- Cumulative Acknowledgements
  - Selective ACKs (full information) also supported (ignore)

- Set timer after each payload is ACK’ed
  - Timer is effectively for the “next expected payload”
  - When the timer goes off, resend that payload and wait
    - And double timeout period

- Various tricks related to “fast retransmit”
TCP Header

These should be familiar
Segments and Sequence Numbers
TCP “Stream of Bytes” Service

Application @ Host A

Application @ Host B
TCP “Stream of Bytes” Service

Application @ Host A

Segment sent when
1) Segment full (Max Segment Size)
2) Not full, but times out

TCP Data

Application @ Host B
TCP Segment

- IP Packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes with Ethernet

- TCP Packet
  - IP packet with a TCP header and data inside
  - TCP header $\geq$ 20 bytes long

- TCP Segment
  - No more than MSS (Maximum Segment Size) bytes
  - E.g., upto 1460 consecutive bytes from the stream
  - MSS = MTU - IP header - TCP header
Sequence Numbers

Host A

Initial Sequence Number (ISN)

Sequence number = 1st byte in segment = ISN + k

K bytes

TCP Data

TCP Hdr

Host B

TCP Data

TCP Hdr

ACK Sequence number = next expected byte = seqno + length(data)
ACKing and Sequence Numbers

• Sender sends segments (byte stream)
  • Data starts with sequence number X
  • Packet contains B bytes
    • X, X+1, X+2, ..., X+B-1

• Upon receipt of a segment, receiver sends an ACK
  • If all data prior to X already received:
    • ACK acknowledges X+B (because that is next expected byte)
  • If highest contiguous byte received is smaller value Y
    • ACK acknowledges Y+1
    • Even if this has been ACKed before
TCP Header

- Source Port
- Destination Port
- Sequence Number
- Acknowledgement
- Header Length (HdrLen)
- Checksum
- Options (variable)
- Data
- Urgent Pointer
- Advertised Window
- Flags

Starting byte offset of data carried in this segment
TCP Header

Acknowledgement gives sequence number just beyond highest sequence number received in order ("What byte is next")
TCP Header

- Source Port
- Destination Port
- Sequence Number
- Acknowledgement
- HdrLen
- Checksum
- Flags
- Urgent Pointer
- Advertised Window
- Options (variable)
- Data
Flow Control (Sliding Window)

• Advertised Window: $W$
  • Can send $W$ bytes beyond the next expected byte

• Receiver uses $W$ to prevent sender from overflowing buffer

• Limits number of bytes sender can have in flight
Filling the Pipe

• Simple example:
  • $W$ (in bytes), which we assume is constant
  • RTT (in sec), which we assume is constant
  • $B$ (in bytes/sec)

• How fast will data be transferred?

• If $W/RTT < B$, the transfer has speed $W/RTT$
• If $W/RTT > B$, the transfer has speed $B$
Advertised Window Limits Rate

• Sender can send no faster than $W/RTT$ bytes/sec

• In original TCP, that was the sole protocol mechanism controlling sender’s rate

• What’s missing?

• **Congestion control** about how to adjust $W$ to avoid network congestion (next lecture)
Any Questions?
Implementing Sliding Window

• Sender maintains a window
  • Data that has been sent out but not yet ACK’ed

• Left edge of window:
  • Beginning of unacknowledged data
  • Moves when data is ACKed

• Window size = maximum amount of data in flight

• Receiver sets this amount, based on its available buffer space
  • If it has not yet sent data up to the app, this might be small
TCP Header: What’s left?

Source Port   Destination Port

Sequence Number

Acknowledgement

HdrLen  0  Flags

Checksum   Advertised Window

Options (variable)

Data
TCP Connection Establishment and Initial Sequence Numbers
Initial Sequence Number (ISN)

- Sequence number for the very first byte
  - E.g., Why not just use ISN = 0?

- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get used again
  - ... small chance an old packet is still in flight

- TCP therefore requires changing ISN
  - Set from 32-bit clock that ticks every 4 microseconds
  - ... only wraps around once every 4.55 hours

- To establish a connection, hosts exchange ISNs
  - How does this help?
### Establishing a TCP Connection

- Three-way handshake to establish connection
  - Host A sends a **SYN** (open; “synchronize sequence numbers”) to host B
  - Host B returns a SYN acknowledgement (**SYN ACK**)
  - Host sends an **ACK** to acknowledge the SYN ACK

Each host tells its ISN to the other host.
TCP Header

Flags:
- SYN
- ACK
- FIN
- RST
- PSH
- URG

See /usr/include/netinet/tcp.h on Unix Systems
Step 1: A’s Initial SYN Packet

A tells B it wants to open a connection…
Step 2: B’s SYN-ACK Packet

B tells A it accepts and is ready to hear the next byte…

… upon receiving this packet, A can start sending data
Step 3: A’s ACK of the SYN-ACK

A tells B it’s likewise okay to start sending

... upon receiving this packet, B can start sending data
Timing Diagram: 3-Way Handshaking

Active Open
Client (initiator)

connect()  
SYN, SeqNum = x

SYN + ACK, SeqNum = y, Ack = x + 1

PASSIVE OPEN
Server
listen()

ACK, ACK = y+1

accept()
Note: TCP is Duplex

- A TCP connection between A and B can carry data in both directions
- Packets can both carry data and ACK data
- If the ACK flag is set, then it is ACKing data
- (details to follow ...)

Any Questions?
Done for today

Next lecture: Congestion control
Back up slides on UDP

(not needed for exams)
UDP: User Datagram Protocol

• Lightweight communication between processes
  • Avoid overhead and delays of ordered, reliable delivery
  • Send messages to and receive from a socket

• UDP described in RFC 768 - (1980)
  • IP plus port numbers to support (de)multiplexing
  • Optional error checking on the packet contents
    • Checksum field = 0 means “don’t verify checksum”
  • (local port, local IP, remote port, remote IP) ←→ socket

<table>
<thead>
<tr>
<th>Source Port #</th>
<th>Dest Port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checksum</td>
<td>Length</td>
</tr>
<tr>
<td>Application Data (Message)</td>
<td></td>
</tr>
</tbody>
</table>
Question

- Why do UDP packets carry sender’s port?
Popular Applications That Use UDP

- Some interactive streaming apps
  - Retransmitting lost/corrupted packets is often pointless — by the time the packet is transmitted, it’s too late
  - E.g., telephone calls, video conferencing, gaming
  - Modern streaming protocols using TCP (and HTTP)

- Simple query protocols like Domain Name System
  - Connection establishment overhead would double cost
  - Easier to have application retransmit if needed

“Address for bbc.co.uk?”

“212.58.224.131”
Back up slides on TCP
(not needed for exams)
What if the SYN Packet Gets Lost?

• Suppose the SYN packet gets lost
  • Packet is lost inside the network, or
  • Server **discards** the packet (e.g., listen queue is full)

• Eventually, no SYN-ACK arrives
  • Sender sets a **timer** and **waits** for the SYN-ACK
  • ... and retransmits the SYN if needed

• How should the TCP sender set the timer?
  • Sender has **no idea** how far away the receiver is
  • Hard to guess a reasonable length of time to wait
  • Should (RFCs 1122 and 2988) use default of 3 seconds
    • Other implementations instead use 6 seconds
SYN Loss and Web Downloads

- User clicks on a hypertext link
  - Browser creates a socket and does a “connect”
  - The “connect” triggers the OS to transmit a SYN

- If the SYN is lost...
  - 3-4 seconds of delay: can be very long
  - User may become impatient
  - ... and click the hyperlink again, or click “reload”

- User triggers an “abort” of the “connect”
  - Browser creates a new socket and another “connect”
  - Essentially, forces a faster send of a new SYN packet!
  - Sometimes very effective, and the page comes quickly
Tearing Down the Connection
Normal Termination

- Finish (FIN) to close connections
  - FIN occupies one byte in the sequence space
- Other host ack’s the byte to confirm
- Closes A’s side of connection, but not B’s
  - Until B likewise sends a FIN
  - Which A then acks
  
Timeout:
- Avoid reincarnation
- Can retransmit FIN ACK if ACK lost

Connection now closed
Normal Termination, Both Together

- Same as before, but B sets FIN with their ack of A’s FIN

Timeout:
Avoid reincarnation
Can retransmit FIN+ACK if ACK lost
Connection now closed
Abrupt Termination

- A sends a RESET (RST) to B
  - E.g., because app. Process on A crashed
- That’s it
  - B does not ack the RST
  - This, RST is not delivered reliably
  - And, any data in flight is lost
  - But, if B sends anything more, will elicit another RST
TCP State Transitions

- CLOSED
  - Passive open /SYN
  - Send/SYN
  - Active open /SYN
  - Close

- LISTEN
  - Closed
  - SYN/SYN + ACK
  - SYN/SYN + ACK
  - SYN + ACK/ACK
  - Send/SYN

- SYN_RCVD
  - ACK
  - Close/FIN
  - SYN/SYN + ACK
  - SYN/SYN + ACK

- ESTABLISHED
  - ACK
  - SYN + ACK/ACK
  - FIN/ACK
  - FIN/ACK
  - Close/FIN

- FIN_WAIT_1
  - ACK
  - Close/FIN
  - FIN/ACK

- FIN_WAIT_2
  - ACK
  - FIN/ACK
  - FIN/ACK

- CLOSING
  - ACK
  - Timeout after two segment lifetimes

- CLOSE_WAIT
  - Close/FIN

- LAST_ACK
  - ACK

- TIME_WAIT
  - ACK

- CLOSED
  - ACK
A Simpler View of the Client Side

- **CLOSED**: SYN (Send)
- **SYN_SENT**: Rcv. SYN + ACK, Send ACK
- **ESTABLISHED**: Send FIN
- **FIN_WAIT_1**: Rcv. ACK, Send Nothing
- **FIN_WAIT_2**: Rcv. FIN, Send ACK
- **TIME_WAIT**:
TCP Header

- Source Port
- Destination Port
- Sequence Number
- Acknowledgement
- HdrLen
- Flags
- Checksum
- Urgent Pointer
- Advertised Window
- Options (variable)
- Data
TCP Header: What’s left?

“Must be Zero”
6 bits reserved

Number of 4-byte words in TCP header; 5 = no options
TCP Header: What’s left?

Used with URG flag to indicate urgent data (not discussed further)