Computer Networks: UDP and TCP

Ankit Singla

Several slides adapted from Jennifer Rexford, Scott Shenker, Laurent Vanbever
Computer networks

Part 3: Transport

#1 What do we need in the transport layer?

#2 How do we build reliable transport?

#3 How does the Internet’s transport work?

#4 Sockets: the application ↔ transport interface
What do we need in the transport layer?

Data delivery to the correct application
  IP just points towards next protocol

Files or byte-streams abstractions for applications
  Network deals with packets
  Transport layer needs to translate between them

Reliable transfer (if needed)

Not overloading the receiver

Not overloading the network
How does Internet transport provide these?

Data delivery to the correct application

Demultiplexing using application identifiers (ports)

Files or byte-streams abstractions for applications

Do segmentation and reassembly

Reliable transfer (if needed): ACKs, checksums

Not overloading the receiver: “Flow control” via receiver window

Not overloading the network: “Congestion control” via sender window
UDP: “User Datagram Protocol”

Data delivery to the correct application

Demultiplexing using application identifiers (ports)

checksums (optionally)

No-frills, simple extension of IP
This is all UDP is

<table>
<thead>
<tr>
<th>SRC port</th>
<th>DST port</th>
</tr>
</thead>
<tbody>
<tr>
<td>checksum</td>
<td>length</td>
</tr>
</tbody>
</table>

DATA
Why would anyone want UDP?

Avoids overhead and delays of ordered, reliable delivery
No delay for connection establishment
Finer control over what data is sent and when
No connection state, buffers, timers, leading to greater scalability
Small per-packet overhead, as headers are minimal

DNS  Gaming  VoIP
TCP: “Transmission Control Protocol”

Data delivery to the correct application
  Demultiplexing using application identifiers (ports)

Files or byte-streams abstractions for applications
  Do segmentation and reassembly

Reliable transfer (if needed): ACKs, checksums

Not overloading the receiver: “Flow control” via receiver window

Not overloading the network: “Congestion control” via sender window
In the Internet, reliability is ensured by the end hosts, not by the network.
Reliability is a **L4 function**, implemented just above the network layer.

**goals**

Keep the network simple, dumb

make it relatively “easy” to build and operate a network

Keep applications as network “unaware” as possible

a developer should focus on its app, not on the network

**design**

Implement reliability in-between, in the networking stack

relieve the burden from both the app and the network
Reliability is a L4 function, implemented just above the network layer.

Layer

Application

L4 Transport reliable end-to-end delivery (if needed)

L3 Network global best-effort delivery

Link

Physical
Recall that L3 provides best-effort service, with no guarantees.

- L4: Transport - reliable end-to-end delivery (if needed)
- L3: Network - global best-effort delivery
TCP: Reliable, in-order delivery

TCP is a connection-oriented, reliable, bytestream transport service.

What UDP provides, plus:

- Retransmission of lost and corrupted packets
- Flow control (to not overflow receiver)
- Congestion control (to not overload network)
- “Connection” set-up & tear-down
Connections (or sessions)

Reliability requires keeping state
- Sender: packets sent but not ACKed, and related timers
- Receiver: non-contiguous packets

Each bytestream is called a connection or session
- Each with their own connection state
- State is in hosts, not network!
What transport protocols do not provide

Delay and/or bandwidth guarantees
  This cannot be offered by transport
  Would require support at IP level

Sessions that survive change-of-IP-address
  This is an artifact of current implementations
  As we shall see….

There’s now Multipath TCP, implemented in your iPhones to address this issue!
Transmission Control Protocol (TCP)

Reliable, in-order delivery *(previously, but quick review)*

- Ensures byte stream (eventually) arrives intact
- In the presence of corruption and loss

Connection oriented *(today)*

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

Full duplex stream-of-bytes service *(today)*

- Sends and receives a stream of bytes, not messages

Flow control *(previously, but quick review)*

- Ensures that sender doesn’t overwhelm receiver

Congestion control *(next lecture)*

- Dynamic adaptation to network path’s capacity
Basic Components of Reliability

ACKs
- Can’t be reliable without knowing whether data has arrived
- TCP uses byte sequence numbers to identify payloads

Checksums
- Can’t be reliable without knowing whether data is corrupted
- TCP does checksum over TCP and pseudoheader

Timeouts and retransmissions
- Can’t be reliable without retransmitting lost/corrupted data
- TCP retransmits based on timeouts and duplicate ACKs
- Timeout 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)

Single timer set after each payload is ACKed

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

Various tricks related to “fast retransmit”

- Using duplicate ACKs to trigger retransmission
## TCP Header

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>Flags</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

The TCP header includes fields such as source and destination ports, sequence number, acknowledgment, header length, flags, advertised window, checksum, urgent pointer, options, and data. Each field serves a specific purpose in managing data transmission over a network.
TCP Header

These should be familiar

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source port</td>
<td></td>
</tr>
<tr>
<td>Destination port</td>
<td></td>
</tr>
<tr>
<td>Sequence number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>Advertised window</td>
<td></td>
</tr>
<tr>
<td>Flags</td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>0</td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
</tr>
<tr>
<td>Urgent pointer</td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>
Segments and Sequence Numbers
TCP “Stream of Bytes” Service...

Application @ Host A

Application @ Host B
... Provided Using TCP “Segments”

Transmitted when:
- Segment full (Max Segment Size)
- Not full, but times out
TCP Segment

<table>
<thead>
<tr>
<th>IP Data</th>
<th>IP Hdr</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP Data (segment)</td>
<td>TCP Hdr</td>
</tr>
</tbody>
</table>

**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 ≥ 20 bytes long

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

ISN (initial sequence number)
Sequence Numbers

ISN (initial sequence number)

ACK sequence number = next expected byte
ACKing and Sequence Numbers

Sender sends packet

- Data starts with sequence number $X$
- Packet contains $B$ bytes
  - $X, X+1, X+2, \ldots X+B-1$

Upon receipt of packet, 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
Normal Pattern

Sender: seqno=X, length=B
Receiver: ACK=X+B

Sender: seqno=X+B, length=B
Receiver: ACK=X+2B

Sender: seqno=X+2B, length=B
...

Seqno of next packet is same as last ACK field
### TCP Header

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Sequence number**
- **Acknowledgment**
- **Advertised window**
- **Flags**
- **Checksum**
- **Urgent pointer**
- **Options (variable)**

### Starting byte offset of data carried in this segment

- **HdrLen**: 0
- **Data**
TCP Header

- Source port
- Destination port
- Sequence number
- Acknowledgment
- Advertised window
- Flags
- Checksum
- Urgent pointer
- HdrLen
- Options (variable)
- Data

“What Byte is Next”
## TCP Header

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>Flags</td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

How much the sender can send
Sliding Window Flow Control

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 $\text{bytes/sec}$)

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

Receiver only advertises more space when it has consumed old arriving data

In original design, sole protocol mechanism controlling sender’s rate!

What’s missing?
Implementing Sliding Window

Both sender & receiver maintain a **window**
- Sender: not yet ACKed
- Receiver: not yet delivered to application

**Left edge** of window:
- Sender: beginning of *unacknowledged* data
- Receiver: beginning of *undelivered* data

For the sender:
- Window size = maximum amount of data in flight

For the receiver:
- Window size = maximum amount of undelivered data
Sliding Window

Allow a larger amount of data “in flight”

- Allow sender to get ahead of the receiver
- … though not too far ahead

![Diagram of Sliding Window](image)

- Sending process
- Receiving process
- TCP
- Last byte ACKed
- Last byte written
- Last byte can send
- Sender Window
- Receiver Window
- Next byte needed
- Last byte received
Sliding Window

When sender receives a new ACK, send window advances (slides forward)
When sender receives a new ACK, send window advances (slides forward)
Sliding Window

When the receiving process consumes data, receive window advances.
Sliding Window

When the receiving process consumes data, receive window advances.

Receiving process

- Last byte read
- Next byte needed
- Last byte received
Sliding Window Summary

Sender: window advances when new data ACKed

Receiver: window advances as receiving process consumes data

Receiver advertises to the sender where the receiver window currently ends (“righthand edge”)

• Sender agrees not to exceed this amount
• It makes sure by setting its own window size to a value that can’t send beyond the receiver’s righthand edge
## TCP Header: What’s left?

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence number</th>
<th>Acknowledgment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Advertised window</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Options (variable)</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **“Must Be Zero”**: 6 reserved bits
- **Number of 4-byte words in TCP header**: 5 ⇒ no options
- **Used with URG flag to indicate urgent data** (not discussed further)
TCP Header: What’s left?

We’ll come to this one shortly
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

- Drawn from a pseudo random number generator

To establish a connection, hosts exchange ISNs
Host A sends a **SYN** (open; “synchronize sequence numbers”)
Host B returns a SYN acknowledgment (**SYN ACK**)
Host A sends an **ACK** to acknowledge the SYN ACK

Each host tells its ISN to the other
TCP Header

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

<table>
<thead>
<tr>
<th>A’s port</th>
<th>B’s port</th>
</tr>
</thead>
<tbody>
<tr>
<td>A’s Initial Sequence Number</td>
<td></td>
</tr>
<tr>
<td>(Irrelevant since ACK not set)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Flags</td>
<td>Advertised window</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>A’s port</th>
<th>B’s port</th>
</tr>
</thead>
<tbody>
<tr>
<td>A’s Initial Sequence Number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACK = B’s ISN plus 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flags</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Advertised window</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Urgent pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A tells B it's likewise okay to start sending

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

Client (initiator)  
“Active open”  

Server  
“Passive open”  

connect()  
SYN, SeqNum = x  
SYN + ACK, SeqNum = y, Ack = x + 1  
ACK, Ack = y + 1  
listen()  
accept()
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 & 2988) use default of 3 seconds
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 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, One Side at a Time

Finish (**FIN**) to close and receive remaining bytes
- **FIN** occupies one octet in the sequence space
Other host ack’s the octet to confirm
Closes A’s side of the connection, but not B’s
- Until B likewise sends a **FIN**
- Which A then acks

Timeout:
Avoid reincarnation
B will retransmit **FIN**
if **ACK** is lost
Normal Termination, Both Together

Same as before, but B sets \textbf{FIN} with their ack of A's \textbf{FIN}
A sends a RESET (RST) to B
  • E.g., because app. process on A crashed
That’s it
  • B does not ack the RST
  • Thus, 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

Data, ACK exchanges are in here
Reliability: TCP Retransmission
Timeouts and Retransmissions

Reliability requires retransmitting lost data

Involves setting timer and retransmitting on timeout

TCP resets timer whenever new data is ACKed

Retx of packet containing “next byte” when timer goes off
Example

Arriving ACK expects 100
Sender sends packets 100, 200, 300, 400, 500
  • Timer set for 100
Arriving ACK expects 300
  • Timer set for 300
Timer goes off
  • Packet 300 is resent
Arriving ACK expects 600
  • Packet 600 sent
  • Timer set for 600
Setting the Timeout Value

Too long ⇒ inefficient

Too short ⇒ duplicate packets
RTT Estimation

Use exponential averaging of RTT samples

\[ \text{SampleRTT} = \text{AckRcvdTime} - \text{SendPacketTime} \]

\[ \text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT} \]

\[ 0 < \alpha \leq 1 \]
Exponential Averaging Example

\[ \text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT} \]

Assume RTT is constant → \( \text{SampleRTT} = \text{RTT} \)
Problem: Ambiguous Measurements

How do we differentiate between the real ACK, and ACK of the retransmitted packet?
Measure \textit{SampleRTT} only for original transmissions

- Do not use retransmits for any measurements
- Computes \textit{EstimatedRTT} using $\alpha = 0.875$

Timeout value (RTO) $= 2 \times \text{EstimatedRTT}$

Use exponential backoff for repeated retransmissions

- Every time RTO timer expires, set RTO $\leftarrow 2 \times$ RTO
  - (Up to maximum $\geq 60$ sec)
- Every time new measurement comes in (= successful original transmission), collapse RTO back to $2 \times \text{EstimatedRTT}$
Figure 5: Performance of an RFC793 retransmit timer

The packets are indicated by a dot. A dashed line connects them to make the sequence easier to follow. The solid line shows the behavior of a retransmit timer computed according to the rules of RFC793.

The parameter $\beta$ accounts for RTT variation (see [5], section 5). The suggested $\beta = 2$ can adapt to loads of at most 30%. Above this point, a connection will respond to load increases by retransmitting packets that have only been delayed in transit. This forces the network to do useless work, wasting bandwidth on duplicates of packets that will eventually be delivered, at a time when it's known to be having trouble with useful work. I.e., this is the network equivalent of pouring gasoline on a fire.

We developed a cheap method for estimating variation (see appendix A) and the resulting retransmit timer essentially eliminates spurious retransmissions. A pleasant side effect of estimating $\beta$ rather than using a fixed value is that low load as well as high load performance improves, particularly over high delay paths such as satellite links (figures 5 and 6).

Another timer mistake is in the backoff after a retransmit: If a packet has to be retransmitted more than once, how should the retransmits be spaced? For a transport endpoint embedded in a network of unknown topology and with an unknown, unknowable and constantly changing population of competing conversations, only one scheme has any hope of working—exponential backoff—but a proof of this is beyond the scope of this paper.

We are far from the first to recognize that transport needs to estimate both mean and variation. See, for example, [6]. But we do think our estimator is simpler than most.

See [8]. Several authors have shown that backoffs 'slower' than exponential are stable given finite populations and knowledge of the global traffic. However, [17] shows that nothing slower than exponential behavior will work in the general case. To feed your intuition, consider that an IP gateway has essentially the same behavior as the 'ether' in an ALOHA net or Ethernet. Justifying exponential retransmit backoff is the same as justifying exponential retransmit backoff.
Karn/Partridge Algorithm

Measure $SampleRTT$ only for original transmissions

- Do not use retransmits for any measurements
- Computes $EstimatedRTT$ using $\alpha = 0.875$

Timeout value (RTO) $= 2 \times EstimatedRTT$

Use exponential backoff for repeated retransmissions

- Every time RTO timer expires, set $RTO \leftarrow 2 \times RTO$
  - (Up to maximum $\geq 60$ sec)
- Every time new measurement comes in (= successful original transmission), collapse RTO back to $2 \times EstimatedRTT$

Does not account for variance
The Jacobson/Karels improvement

\[ \text{Deviation} = | \text{SampleRTT} - \text{EstimatedRTT} | \]

\[ \text{EstimatedDeviation: exponential average of Deviation} \]

\[ \text{RTO} = \text{EstimatedRTT} + 4 \times \text{EstimatedDeviation} \]
With Jacobson/Karels

Figure 6: Performance of a Mean+Variance retransmit timer

---

If the timers are in good shape, it is possible to state with some confidence that a timeout indicates a lost packet and not a broken timer. At this point, something can be done about (3). Packets get lost for two reasons: they are damaged in transit, or the network is congested and somewhere on the path there was insufficient buffer capacity. On most network paths, loss due to damage is rare ($\ll 1\%$) so it is probable that a packet loss is due to congestion in the network.
In practice ...

(2.4) Whenever RTO is computed, if it is less than 1 second, then the RTO SHOULD be rounded up to 1 second.

200 milliseconds in Linux. Incurring a timeout is expensive!

Relevance of these algorithms is reduced by use of duplicate ACKs
Loss with cumulative ACKs

Sender sends packets with 100B and seqnos:

- 100, 200, 300, 400, 500, 600, 700, 800, 900, …

Assume the fifth packet (seqno 500) is lost, but no others

Stream of ACKs will be:

- 200, 300, 400, 500, 500, 500, 500, …
Loss with cumulative ACKs

“Duplicate ACKs” are a sign of an isolated loss

- The lack of ACK progress means 500 hasn’t been delivered
- Stream of ACKs means some packets are being delivered

Therefore, could trigger resend upon receiving $k$ duplicate ACKs

- TCP uses $k=3$

We will revisit this in congestion control
Congestion control
TCP Congestion Control
Because of traffic burstiness and lack of BW reservation, congestion is inevitable.

If many packets arrive within a short period of time, the node cannot keep up anymore.
Congestion is harmful
average packet arrival rate $a$ [packet/sec]

transmission rate of outgoing link $R$ [bit/sec]

fixed packets length $L$ [bit]

average bits arrival rate $La$ [bit/sec]

traffic intensity $La/R$
When the traffic intensity is $>1$, the queue will increase without bound, and so does the queuing delay

Golden rule

Design your queuing system,
so that it operates far from that point
When the *traffic intensity is \( \leq 1 \),
queueing delay depends on the burst size.
Congestion is not a new problem

The Internet almost died of congestion in 1986
throughput collapsed from 32 Kbps to… 40 bps

Van Jacobson saved us with Congestion Control
his solution went right into BSD

Recent resurgence of research interest after brief lag
new methods (ML), context (Data centers), requirements
The Internet almost died of congestion in 1986
throughput collapsed from 32 Kbps to... 40 bps
On connection, nodes send full window of packets.

Upon timer expiration, retransmit packet immediately.

Sending rate only limited by flow control.

Window-sized burst of packets.
Increase in network load results in a **decrease** of useful work done.

Sudden load increased the round-trip time (RTT) faster than the hosts’ measurements of it.

As RTT exceeds the maximum retransmission interval, hosts begin to retransmit packets.

Hosts are sending each packet several times, eventually some copies arrive at the destination.

This phenomenon is known as **congestion collapse**.
Knee point after which

throughput increases slowly
delay increases quickly

Cliff point after which

throughput decreases quickly
delay tends to infinity

Knee

cliff
congestion collapse

2C CONSERVATION EQUILIBRIUM: ROUND TRIP TIMING

This burst of packets often puts the connection into a persistent failure mode of continuous retransmissions (figures 3 and 4). The dashed line shows the 20 KBps bandwidth available for this connection. Only 35% of this bandwidth was used; the rest was wasted on retransmits. Almost everything is available bandwidth. Nothing in this trace resembles desirable behavior.

Ideal behavior on this graph would be a relatively smooth line of dots extending diagonally from the lower left to the upper right. The slope of this line would equal the 'Desirable' behavior on this graph would be a relatively smooth line of dots extending diagonally from the lower left to the upper right. The slope of this line would equal the

Packet Sequence Number (KB)

Ideal behavior

Packet seq number (indexed by KB)

Send time (s)

Startup behavior of TCP without Slow-start
Van Jacobson saved us with **Congestion Control**

his solution went right into BSD
Congestion control aims at solving three problems:

1. **bandwidth estimation**
   - How to adjust the bandwidth of a single flow to the bottleneck bandwidth?
   - Could be 1 Mbps or 1 Gbps…

2. **bandwidth adaptation**
   - How to adjust the bandwidth of a single flow to variation of the bottleneck bandwidth?

3. **fairness**
   - How to share bandwidth “fairly” among flows, without overloading the network
Congestion control differs from flow control
both are provided by TCP though

Flow control
prevents one fast sender from
overloading a slow receiver

Congestion control
prevents a set of senders from
overloading the network
TCP solves both using two distinct windows

Flow control prevents one fast sender from overloading a slow receiver

solved using a receiving window

Congestion control prevents a set of senders from overloading the network

solved using a “congestion” window
The sender adapts its sending rate based on these two windows:

- **Receiving Window (RWND)**: How many bytes can be sent without overflowing the receiver buffer? Based on the receiver input.

- **Congestion Window (CWND)**: How many bytes can be sent without overflowing the routers? Based on network conditions.

- **Sender Window**: $\text{minimum}(\text{CWND, RWND})$
The 2 key mechanisms of Congestion Control

detecting congestion

reacting to congestion
The 2 key mechanisms of Congestion Control

detecting congestion

reacting to congestion
There are essentially three ways to detect congestion:

**Approach #1**
- Network could tell the source
- but signal itself could be lost

**Approach #2**
- Measure packet delay
- but signal is noisy
- delay often varies considerably

**Approach #3**
- Measure packet loss
- fail-safe signal that TCP already has to detect
Packet dropping is the best solution

delay- and signaling-based methods are hard & risky

Approach #3

Measure packet loss

fail-safe signal that TCP already has to detect
Detecting losses can be done using ACKs or timeouts, the two signals differ in their degree of severity.

- **duplicated ACKs**
  - *mild congestion signal*
  - packets are still making it

- **timeout**
  - *severe congestion signal*
  - multiple consequent losses
The 2 key mechanisms of Congestion Control

detecting congestion

reacting to congestion
TCP’s approach is to *gently increase* when not congested and to *rapidly decrease* when congested.

**question**

What *increase/decrease function* should we use?

*it depends on the problem we are solving...*
Remember that Congestion Control aims at solving three problems

#1 bandwidth estimation
How to adjust the bandwidth of a single flow to the bottleneck bandwidth?

could be 1 Mbps or 1 Gbps…

#2 bandwidth adaptation
How to adjust the bandwidth of a single flow to variation of the bottleneck bandwidth?

#3 fairness
How to share bandwidth “fairly” among flows, without overloading the network
How to adjust the bandwidth of a single flow to the bottleneck bandwidth? Could be 1 Mbps or 1 Gbps…
The goal here is to quickly get a first-order estimate of the available bandwidth

**Intuition**
Start slow but rapidly increase until a packet drop occurs

**Increase policy**
- \( cwnd = 1 \) initially
- \( cwnd += 1 \) upon receipt of an ACK
This increase phase, known as slow start, corresponds to an... exponential increase of CWND!

slow start is called like this only because of starting point
The problem with slow start is that it can result in a full window of packet losses

Example
Assume that CWND is just enough to “fill the pipe”
After one RTT, CWND has doubled
All the excess packets are now dropped

Solution
We need a more gentle adjustment algorithm
once we have a rough estimate of the bandwidth
How to adjust the bandwidth of a single flow to variation of the bottleneck bandwidth?
The goal here is to track the available bandwidth, and oscillate around its current value.

Two possible variations

- **Multiplicative Increase or Decrease**
  
  \[ cwnd = a \times cwnd \]

- **Additive Increase or Decrease**
  
  \[ cwnd = b + cwnd \]

... leading to four alternative designs.
<table>
<thead>
<tr>
<th></th>
<th>increase behavior</th>
<th>decrease behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIAD</td>
<td>gentle</td>
<td>gentle</td>
</tr>
<tr>
<td>AIMD</td>
<td>gentle</td>
<td>aggressive</td>
</tr>
<tr>
<td>MIAD</td>
<td>aggressive</td>
<td>gentle</td>
</tr>
<tr>
<td>MIMD</td>
<td>aggressive</td>
<td>aggressive</td>
</tr>
</tbody>
</table>
To select one scheme, we need to consider the 3rd problem: **fairness**

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Increase Behavior</th>
<th>Decrease Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIAD</td>
<td>Gentle</td>
<td>Gentle</td>
</tr>
<tr>
<td>AIMD</td>
<td>Gentle</td>
<td>Aggressive</td>
</tr>
<tr>
<td>MIAD</td>
<td>Aggressive</td>
<td>Gentle</td>
</tr>
<tr>
<td>MIMD</td>
<td>Aggressive</td>
<td>Aggressive</td>
</tr>
</tbody>
</table>
#3 fairness

How to share bandwidth “fairly” among flows, without overloading the network
TCP notion of fairness: 2 identical flows should end up with the same bandwidth
Consider first a single flow between A and B and AIMD.

**Without congestion:** CWND increases by one packet every RTT.

**Upon congestion:** CWND decreases by a factor 2.
Rate (pkts/RTT)

Backlog in router (pkts)
Congested if > 20
Now let’s look at two senders A and B sharing a link

without congestion  CWND increases by one packet every RTT
upon congestion    CWND decreases by a factor 2
We can analyze the system behavior using a system trajectory plot.
The system is efficient if the capacity is fully used, defining an efficiency line where $a + b = 1$
The goal of congestion control is to bring the system as close as possible to this line, and stay there.
A’s throughput

B’s throughput

congestion
under-utilization
The system is fair whenever A and B have equal throughput, defining a *fairness line* where $a = b$.
A's throughput

B's throughput

--

fairness line

B gets more than A

1
A gets more than B

A’s throughput

B’s throughput

fairness line
A's throughput

B's throughput

inefficient & unfair

fairness line

efficiency line

.2

.5
A's throughput

B's throughput

congested

efficiency line

fairness line

.5

.7
A's throughput vs B's throughput graph shows the trade-off between efficiency and fairness. The 'efficient & unfair' point at (0.3, 1) suggests that for low efficiency, the throughput is highly unfair. The 'fairness line' and 'efficiency line' indicate the boundary between fair and efficient and efficient but unfair performance, respectively.
A's throughput

B's throughput

Efficient & fair

Fairness line

Efficiency line
<table>
<thead>
<tr>
<th></th>
<th>increase behavior</th>
<th>decrease behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIAD</td>
<td>gentle</td>
<td>gentle</td>
</tr>
<tr>
<td>AIMD</td>
<td>gentle</td>
<td>aggressive</td>
</tr>
<tr>
<td>MIAD</td>
<td>aggressive</td>
<td>gentle</td>
</tr>
<tr>
<td>MIMD</td>
<td>aggressive</td>
<td>aggressive</td>
</tr>
</tbody>
</table>
A small algebra-geometry recap

Multiplying both $X$ and $Y$ by some $\alpha$ moves them on a line through the origin.
A small algebra-geometry recap

Adding some $\alpha$ to both $X$ and $Y$ moves them on a line of slope 45°
<table>
<thead>
<tr>
<th></th>
<th>increase behavior</th>
<th>decrease behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIAD</td>
<td>gentle</td>
<td>gentle</td>
</tr>
<tr>
<td>AIMD</td>
<td>gentle</td>
<td>aggressive</td>
</tr>
<tr>
<td>MIAD</td>
<td>aggressive</td>
<td>gentle</td>
</tr>
<tr>
<td>MIMD</td>
<td>aggressive</td>
<td>aggressive</td>
</tr>
</tbody>
</table>
AIAD does not converge to fairness, nor efficiency:
the system fluctuates between two states

Adding a constant:
move along 45 deg
AIAD does not converge to fairness, nor efficiency:
the system fluctuates between two states
<table>
<thead>
<tr>
<th></th>
<th>increase behavior</th>
<th>decrease behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIAD</td>
<td>gentle</td>
<td>gentle</td>
</tr>
<tr>
<td>AIMD</td>
<td>gentle</td>
<td>aggressive</td>
</tr>
<tr>
<td>MIAD</td>
<td>aggressive</td>
<td>gentle</td>
</tr>
<tr>
<td>MIMD</td>
<td>aggressive</td>
<td>aggressive</td>
</tr>
</tbody>
</table>
MIMD does not converge to fairness, nor efficiency:
the system fluctuates along an equi-fairness line
<table>
<thead>
<tr>
<th></th>
<th>Increase Behavior</th>
<th>Decrease Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIAD</td>
<td>gentle</td>
<td>gentle</td>
</tr>
<tr>
<td>AIMD</td>
<td>gentle</td>
<td>aggressive</td>
</tr>
<tr>
<td>MIAD</td>
<td>aggressive</td>
<td>gentle</td>
</tr>
<tr>
<td>MIMD</td>
<td>aggressive</td>
<td>aggressive</td>
</tr>
</tbody>
</table>
MIAD converges to a totally unfair allocation, favoring the flow with a greater rate at the beginning.
If flows start along the fairness line, MIAD fluctuates along it, yet deviating from it at the slightest change.
<table>
<thead>
<tr>
<th>Model</th>
<th>Increase Behavior</th>
<th>Decrease Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIAD</td>
<td>gentle</td>
<td>gentle</td>
</tr>
<tr>
<td>AIMD</td>
<td>gentle</td>
<td>aggressive</td>
</tr>
<tr>
<td>MIAD</td>
<td>aggressive</td>
<td>gentle</td>
</tr>
<tr>
<td>MIMD</td>
<td>aggressive</td>
<td>aggressive</td>
</tr>
</tbody>
</table>
AIMD converges to fairness and efficiency, it then fluctuates around the optimum (in a stable way)
AIMD converges to fairness and efficiency, it then fluctuates around the optimum (in a stable way)

<table>
<thead>
<tr>
<th>Intuition</th>
<th>During increase, both flows gain bandwidth at the same rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>During decrease, the faster flow releases more</td>
</tr>
</tbody>
</table>
AIMD converges to fairness and efficiency, it then fluctuates around the optimum (in a stable way)
In practice, TCP implements AIMD

<table>
<thead>
<tr>
<th></th>
<th>increase</th>
<th>decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>behavior</td>
<td>behavior</td>
<td>behavior</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>gentle</th>
<th>aggressive</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIAD</td>
<td>gentle</td>
<td>gentle</td>
</tr>
<tr>
<td>AIMD</td>
<td>gentle</td>
<td>aggressive</td>
</tr>
<tr>
<td>MIAD</td>
<td>aggressive</td>
<td>gentle</td>
</tr>
<tr>
<td>MIMD</td>
<td>aggressive</td>
<td>aggressive</td>
</tr>
</tbody>
</table>
In practice, TCP implements AIMD

**Implementation**

After each ACK,

Increment cwnd by $1/cwnd$

linear increase of max. 1 per RTT

**Question**

When does a sender leave slow-start and start AIMD?

Introduce a slow start threshold, adapt it as function of congestion:

on timeout, $ssthresh = CWND/2$
TCP congestion control in less than 10 lines of code

Initially:
\[
\text{cwnd} = 1 \\
\text{ssthresh} = \text{infinite}
\]

New ACK received:
\[
\text{if} \ (\text{cwnd} < \text{ssthresh}): \\
\quad /* \text{Slow Start}*/ \\
\quad \text{cwnd} = \text{cwnd} + 1 \\
\text{else:} \\
\quad /* \text{Congestion Avoidance}*/ \\
\quad \text{cwnd} = \text{cwnd} + 1/\text{cwnd}
\]

Timeout:
\[
\quad /* \text{Multiplicative decrease} */ \\
\quad \text{ssthresh} = \text{cwnd}/2 \\
\quad \text{cwnd} = 1
\]
The congestion window of a TCP session typically undergoes multiple cycles of slow-start/AIMD.

![Graph showing TCP congestion window behavior](graph.png)
Going back all the way back to 0 upon timeout completely destroys throughput

**solution**

Avoid timeout expiration…

which are usually \(>500\text{ms}\)
Detecting losses can be done using **ACKs** or timeouts, the two signals differ in their degree of severity

- duplicated ACKs
  - mild congestion signal
  - packets are still making it

- timeout
  - severe congestion signal
  - multiple consequent losses
TCP automatically resends a segment after receiving 3 duplicates ACKs for it. This is known as a “fast retransmit.”
After a fast retransmit, TCP switches back to AIMD, without going all the way back to 0. This is known as “fast recovery.”
TCP congestion control (almost complete)

**Initially:**
- cwnd = 1
- ssthresh = infinite

**New ACK received:**
- if (cwnd < ssthresh):
  - /* Slow Start*/
  - cwnd = cwnd + 1
- else:
  - /* Congestion Avoidance */
  - cwnd = cwnd + 1/cwnd
  - dup_ack = 0

**Timeout:**
- /* Multiplicative decrease */
- ssthresh = cwnd/2
- cwnd = 1

**Duplicate ACKs received:**
- dup_ack ++;
- if (dup_ack >= 3):
  - /* Fast Recovery */
  - ssthresh = cwnd/2
  - cwnd = ssthresh
Initially:
  cwnd = 1
  ssthresh = infinite

New ACK received:
  if (cwnd < ssthresh):
    /* Slow Start*/
    cwnd = cwnd + 1
  else:
    /* Congestion Avoidance */
    cwnd = cwnd + 1/cwnd

  dup_ack = 0

Timeout:
  /* Multiplicative decrease */
  ssthresh = cwnd/2
  cwnd = 1

Duplicate ACKs received:
  dup_ack ++;
  if (dup_ack >= 3):
    /* Fast Recovery */
    ssthresh = cwnd/2
    cwnd = ssthresh
Congestion control makes TCP throughput look like a “sawtooth”
Let’s see this in action!

Two TCP flows competing — queuing, drops, rate adjustment
http://nrg.cs.ucl.ac.uk/mjh/gz01/lectures/twotcps.mp4

Two TCP flows competing — oscillations around an efficient and fair allocation
http://nrg.cs.ucl.ac.uk/mjh/gz01/lectures/chui-jain.mp4

Credit: Mark Handley, UCL
Different “flavors” of TCP have different behavior

See: http://intronetworks.cs.luc.edu/1/html/newtcps.html
The “Anternet”

The Regulation of Ant Colony Foraging Activity without Spatial Information

Balaji Prabhakar¹, Katherine N. Dektar², Deborah M. Gordon³*

[Bob Peterson via Wikimedia]
Where we are in the course ...

- Part 1: Overview & Principles
- Part 2: Applications
- Part 3: Transport
- Part 4: Algorithms
Computer networks

Part 3: Transport

#1 What do we need in the transport layer?

#2 How do we build reliable transport?

#3 How does the Internet’s transport work?

#4 Sockets: the application ⟷ transport interface
Where we are in the course ...

- Part 1: Overview & Principles
- Part 2: Applications
- Part 3: Transport
- Part 4: Algorithms

Sockets
Sockets and Ports

**Sockets**: an operating system abstraction

**Ports**: a networking abstraction

This is not a port on a switch (which is an interface)
Think of it as a *logical interface* on a host
Sockets

A socket is a software abstraction by which an application process exchanges network messages with the (transport layer in the) operating system

socketID = socket(…, socket.TYPE)
socketID.sendto(message, …)
socketID.recvfrom(…)

Two important types of sockets
UDP socket: TYPE is SOCK_DGRAM
TCP socket: TYPE is SOCK_STREAM
Ports

**Problem:** which app (socket) gets which packets

**Solution:** port as transport layer identifier (16 bits)

  Packet carries source/destination port numbers in transport header

OS stores mapping between sockets and ports

  Port: in packets
  Socket: in OS

Try: `lsof -Pn -i4`
More on 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, https:443
  Client (app) knows appropriate port on server
  Services can listen on well-known port

Ephemeral ports (most 1024-65535):
  Given to clients (at random)
Multiplexing and Demultiplexing

Host receives IP datagrams with …
source and destination IP address
source and destination port number

IP address, port used to deliver the segment to appropriate socket

<table>
<thead>
<tr>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
</tr>
<tr>
<td>other header fields</td>
</tr>
<tr>
<td>application data (message)</td>
</tr>
<tr>
<td>4-bit Version</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>16-bit Identification</td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
</tr>
<tr>
<td>Options (if any)</td>
</tr>
<tr>
<td>Payload</td>
</tr>
<tr>
<td>Bit Position</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>16-bit Identification</td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
</tr>
</tbody>
</table>

**Payload**
### IP Header

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version and Header Length</td>
<td>4 bytes</td>
<td>5</td>
</tr>
<tr>
<td>Type of Service (TOS)</td>
<td>8 bits</td>
<td></td>
</tr>
<tr>
<td>Total Length (Bytes)</td>
<td>16 bits</td>
<td></td>
</tr>
<tr>
<td>Identification</td>
<td>16 bits</td>
<td></td>
</tr>
<tr>
<td>Flags</td>
<td>3 bits</td>
<td></td>
</tr>
<tr>
<td>Fragment Offset</td>
<td>13 bits</td>
<td></td>
</tr>
<tr>
<td>Time to Live (TTL)</td>
<td>8 bits</td>
<td>127</td>
</tr>
<tr>
<td>Protocol</td>
<td>8 bits</td>
<td>6 (TCP)</td>
</tr>
<tr>
<td>Source IP Address</td>
<td>32 bits</td>
<td></td>
</tr>
<tr>
<td>Destination IP Address</td>
<td>32 bits</td>
<td></td>
</tr>
</tbody>
</table>

**Payload**

The IP header is followed by the **Payload**, which contains the actual data being transmitted. This data can vary widely depending on the application, such as data packets for file transfers, voice over IP (VoIP) calls, or video streams.
<table>
<thead>
<tr>
<th>8-bit Type of Service (TOS)</th>
<th>16-bit Total Length (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit Identification</td>
<td>3-bit Flags</td>
</tr>
<tr>
<td>13-bit Fragment Offset</td>
<td>16-bit Header Checksum</td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
<td>6 = TCP</td>
</tr>
<tr>
<td></td>
<td>17 = UDP</td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td>32-bit Destination IP Address</td>
</tr>
<tr>
<td>16-bit Source Port</td>
<td>16-bit Destination Port</td>
</tr>
<tr>
<td></td>
<td>More transport header fields ....</td>
</tr>
<tr>
<td></td>
<td>Payload</td>
</tr>
</tbody>
</table>
Connection Mappings

- For UDP ports (SOCK_DGRAM)
  - OS stores (local port, local IP address) $\mapsto$ socket

- For TCP ports (SOCK_STREAM)
  - OS stores (local port, local IP, remote port, remote IP) $\mapsto$ socket

- Why the difference?

- Implications for mobility

- Why do you need to include local IP?