Last week on

Communication Networks
Border Gateway Protocol
policies and more

BGP Policies
Follow the Money

Protocol
How does it work?

Problems
security, performance, ...
BGP sessions come in two flavors
external BGP (eBGP) sessions connect border routers in different ASes
iBGP sessions are used to disseminate externally-learned routes internally
BGP UPDATEs carry an IP prefix together with a set of attributes

- **IP prefix**
- **Attributes**

- Describe route properties used in route selection/exportation decisions
- are either local (only seen on iBGP)
- or global (seen on iBGP and eBGP)
<table>
<thead>
<tr>
<th>Attributes</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEXT-HOP</td>
<td>egress point identification</td>
</tr>
<tr>
<td>AS-PATH</td>
<td>loop avoidance</td>
</tr>
<tr>
<td></td>
<td>outbound traffic control</td>
</tr>
<tr>
<td></td>
<td>inbound traffic control</td>
</tr>
<tr>
<td>LOCAL-PREF</td>
<td>outbound traffic control</td>
</tr>
<tr>
<td>MED</td>
<td>inbound traffic control</td>
</tr>
</tbody>
</table>
Prefer routes…

with higher LOCAL-PREF

with shorter AS-PATH length

with lower MED

learned via eBGP instead of iBGP

with lower IGP metric to the next-hop

with smaller egress IP address (tie-break)
Border Gateway Protocol

policies and more

BGP Policies
Follow the Money

Protocol
How does it work?

Problems
security, performance, ...
This week on

Communication Networks
We’re continuing our journey up the layers, and will start looking at the transport layer.
But first…

Let's finish BGP
BGP suffers from many rampant problems

*(switch to last week's slides)*

<table>
<thead>
<tr>
<th>Problems</th>
<th>Reachability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Security</td>
</tr>
<tr>
<td></td>
<td>Convergence</td>
</tr>
<tr>
<td></td>
<td>Performance</td>
</tr>
<tr>
<td></td>
<td>Anomalies</td>
</tr>
<tr>
<td></td>
<td>Relevance</td>
</tr>
</tbody>
</table>
BGP security has been improving lately 🎉
let's quickly see how

Problems
Reachability
Security Solutions: Origin and Path Validation
Convergence
Performance
Anomalies
Relevance
BGP Security Today  Yesterday

• Applying best common practices (BCPs)
  – Securing the session (authentication, encryption)
  – Filtering routes by prefix and AS path
  – Packet filters to block unexpected control traffic

• This is not good enough
  – Depends on vigilant application of BCPs
  – Doesn’t address fundamental problems
    • Can’t tell who owns the IP address block
    • Can’t tell if the AS path is bogus or invalid
    • Can’t be sure the data packets follow the chosen route
Today, BGP is *slowly* becoming more secure thanks to cryptography.

Plain BGP
"web of trust"

**today**

Route Origin Validation
using Resource Public Key Infrastructure (RPKI)

Path Validation
using BGPSec
Route Origin Validation using Resource Public Key Infrastructure (RPKI)

Plain BGP "web of trust"

today

Path Validation using BGPSec
RPKI enables to validate the *origin* of a BGP route by *certifying* IP prefixes allocations.

RPKI is a database storing Route Origin Authorization (ROAs). ROAs map prefix space (130.0.0.0/8–32) to an origin AS.

Routers consult this database to verify BGP messages. BGP messages are *not* changed, RPKI works "out-of-band".

RPKI has been standardized in 2012 (RFC 6480). Today, RPKI can validate ~28% of the IPv4 prefixes.
Let's look back at an example, first without RPKI

Source: Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"
Without RPKI, a more-specific attack by AS34109 successfully manages to attract the traffic.
Let's assume now that AS 29997 registers (204.16.254.0/24–32, 29997) as a new ROA.

Source: Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"
Using the RPKI, greenhost.nl sees that AS34109 is *not* a valid origin for 204.16.254.40/32

Source: Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"
This announcement is said to be INVALID

Source: Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"
Now what if AS34109 announce AS29997 as the origin?

Source: Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"
Here greenhost.nl receives 2 valid RPKI routes: one via NTT and another one via 34109

the “1-hop hijack” defeats the RPKI

(This exact situation is hypothetical, but this type of attack has been seen in the wild, See [Schlamp, Carle, Biersack 2013])

Source: Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"
As the route via 34109 has a shorter path, it is preferred... **the attack works again!**

---

**Source:** Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"
We see that RPKI does not protect against all attacks

Source: Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"
Route Origin Validation using Resource Public Key Infrastructure (RPKI)

Plain BGP
"web of trust"

today

Path Validation using BGPSec
Secure BGP

Origin + Path Authentication using cryptographic signatures

Public Key Signature: Anyone who knows v's public key can verify that the message was sent by v.
Secure BGP

Origin + Path Authentication using cryptographic signatures

Public Key Signature: Anyone who knows v’s public key can verify that the message was sent by v.
Secure BGP

Origin + Path Authentication using cryptographic signatures

Anyone who knows v's public key can verify that the message was sent by v.
S-BGP Secure Version of BGP

• **Address attestations**, using **RPKI**
  – Claim the right to originate a prefix
  – Signed and distributed out-of-band
  – Checked through delegation chain from ICANN

• **Route attestations**
  – Distributed as an attribute in BGP update message
  – Signed by each AS as route traverses the network

• **S-BGP can validate**
  – AS path indicates the order ASes were traversed
  – No intermediate ASes were added or removed
S-BGP Deployment Challenges

- Complete, accurate registries of prefix “owner”
- Public Key Infrastructure
  - To know the public key for any given AS
- Cryptographic operations
  - E.g., digital signatures on BGP messages
- Need to perform operations quickly
  - To avoid delaying response to routing changes
- Difficulty of incremental deployment
  - Hard to have a “flag day” to deploy S-BGP
BGP suffers from many rampant problems

<table>
<thead>
<tr>
<th>Problems</th>
<th>Reachability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Security</td>
</tr>
<tr>
<td></td>
<td>Convergence</td>
</tr>
<tr>
<td></td>
<td>Performance</td>
</tr>
<tr>
<td></td>
<td>Anomalies</td>
</tr>
<tr>
<td></td>
<td>Relevance</td>
</tr>
</tbody>
</table>
That's it!

for the network layer, and for now…
We’re continuing our journey up the layers, and will start looking at the transport layer.
What do we need in the Transport layer?

Functionality implemented in network
- Keep minimal (easy to build, broadly applicable)

Functionality implemented in the application
- Keep minimal (easy to write)
- Restricted to application-specific functionality

Functionality implemented in the “network stack”
- The shared networking code on the host
- This relieves burden from both app and network
- The transport layer is a key component here
What do we need in the Transport layer?

Application layer
- Communication for specific applications
- e.g., HyperText Transfer Protocol (HTTP), File Transfer Protocol (FTP)

Network layer
- Global communication between hosts
- Hides details of the link technology
- e.g., Internet Protocol (IP)
What Problems Should Be Solved Here?

Data delivering, to the *correct* application

- IP just points towards next protocol
- *Transport needs to demultiplex incoming data (ports)*

Files or bytestreams abstractions for the applications

- Network deals with packets
- *Transport layer needs to translate between them*

Reliable transfer *(if needed)*

Not overloading the receiver

Not overloading the network
What Is Needed to Address These?

Demultiplexing: identifier for application process
- Going from host-to-host (IP) to process-to-process

Translating between bytestreams and packets:
- Do segmentation and reassembly

Reliability: ACKs and all that stuff

Corruption: Checksum

Not overloading receiver: “Flow Control”
- Limit data in receiver’s buffer

Not overloading network: “Congestion Control”
UDP: Datagram messaging service

UDP provides a connectionless, unreliable transport service.

- No-frills extension of “best-effort” IP
- UDP provides only two services to the App layer
  - Multiplexing/Demultiplexing among processes
  - Discarding corrupted packets (optional)
TCP: Reliable, in-order delivery

TCP provides 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: noncontiguous 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
- Requires support at IP level (*and let’s not go there*)

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

- Each datagram has source and destination IP address,
- Each segment has source and destination port number

Host uses IP addresses 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></td>
</tr>
<tr>
<td>Other header fields</td>
<td></td>
</tr>
<tr>
<td>Application data</td>
<td></td>
</tr>
<tr>
<td>(message)</td>
<td></td>
</tr>
<tr>
<td>4-bit Version</td>
<td>4-bit Header Length</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit Identification</td>
<td>3-bit Flags</td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
<td>8-bit Protocol</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (if any)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Payload</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>16-bit Identification</td>
<td>3-bit Flags</td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
<td>8-bit Protocol</td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td></td>
</tr>
<tr>
<td>Payload</td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Format</td>
</tr>
<tr>
<td>----------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Type of Service (TOS)</td>
<td>8-bit</td>
</tr>
<tr>
<td>Total Length (Bytes)</td>
<td>16-bit</td>
</tr>
<tr>
<td>Identification</td>
<td>16-bit</td>
</tr>
<tr>
<td>Flags</td>
<td>3-bit</td>
</tr>
<tr>
<td>Fragment Offset</td>
<td>13-bit</td>
</tr>
<tr>
<td>Time to Live (TTL)</td>
<td>8-bit</td>
</tr>
<tr>
<td>Source IP Address</td>
<td>32-bit</td>
</tr>
<tr>
<td>Destination IP Address</td>
<td>32-bit</td>
</tr>
</tbody>
</table>

Payload
<table>
<thead>
<tr>
<th>Field</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>4</td>
<td>4 = IPv4, 6 = IPv6</td>
</tr>
<tr>
<td>Internet Protocol</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Type of Service (TOS)</td>
<td>8-bit</td>
<td></td>
</tr>
<tr>
<td>Total Length (Bytes)</td>
<td>16-bit</td>
<td></td>
</tr>
<tr>
<td>Identification</td>
<td>16-bit</td>
<td></td>
</tr>
<tr>
<td>Flags</td>
<td>3-bit</td>
<td></td>
</tr>
<tr>
<td>Fragment Offset</td>
<td>13-bit</td>
<td></td>
</tr>
<tr>
<td>Time to Live (TTL)</td>
<td>8-bit</td>
<td></td>
</tr>
<tr>
<td>Protocol</td>
<td>8-bit</td>
<td>6 = TCP, 17 = UDP</td>
</tr>
<tr>
<td>Source IP Address</td>
<td>32-bit</td>
<td></td>
</tr>
<tr>
<td>Destination IP Address</td>
<td>32-bit</td>
<td></td>
</tr>
<tr>
<td>Source Port</td>
<td>16-bit</td>
<td></td>
</tr>
<tr>
<td>Destination Port</td>
<td>16-bit</td>
<td></td>
</tr>
</tbody>
</table>

More transport header fields ….

Payload
UDP
UDP: User Datagram Protocol

Lightweight communication between processes
- Avoid overhead and delays of ordered, reliable delivery
- Send messages to and receive them 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”)
Why Would Anyone Use UDP?

Finer control over what data is sent and when
- As soon as an application process writes into the socket
- … UDP will package the data and send the packet

No delay for connection establishment
- UDP just blasts away without any formal preliminaries
- … which avoids introducing any unnecessary delays

No connection state
- No allocation of buffers, sequence #s, timers …
- … making it easier to handle many active clients at once

Small packet header overhead
- UDP header is only 8 bytes
Popular Applications That Use UDP

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

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

"Address for bbc.co.uk?"
"212.58.224.131"
TCP
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 week)*
- 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

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></td>
<td></td>
</tr>
</tbody>
</table>

| Sequence number |                  |
|-----------------|                  |

| Acknowledgment  |                  |
|-----------------|                  |

<table>
<thead>
<tr>
<th>HdrLen</th>
<th>Flags</th>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Options (variable)</th>
</tr>
</thead>
</table>

### Data
**TCP Header**

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td>Acknowledgment</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>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

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

Application @ Host A

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

Segment sent when:
1. Segment full (Max Segment Size),
2. Not full, but times out
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 ≥ 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 number = 1st byte in segment = ISN + k
Sequence Numbers

Host A

ISN (initial sequence number)

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

TCP Data

TCP HDR

Host B

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

Sender sends packet
  - Data starts with sequence number X
  - Packet contains B bytes
    - X, X+1, X+2, …, 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>
</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>HdrLen</th>
<th>Flags</th>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Data
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
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 TCP design, that was the **sole** protocol mechanism controlling sender’s rate

What’s missing?
Implementing Sliding Window

Both sender & receiver maintain a window

- Sender: not yet ACK’ed
- 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 Summary

Sender: window advances when new data ack’d

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>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<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 bits reserved
- Number of 4-byte words in TCP header; 5 = no options
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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>HdrLen</th>
<th>Flags</th>
<th>Advertised window</th>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Used with **URG** flag to indicate urgent data (not discussed further)
TCP Header: What’s left?

<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>0</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>

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
- initially set from 32-bit clock that ticks every 4 microseconds
- now drawn from a pseudo random number generator (security)

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”)
- 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 host.
## TCP Header

<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>0</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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Flags</th>
<th>5=20B</th>
<th>A’s port</th>
<th>B’s port</th>
<th>A’s Initial Sequence Number</th>
<th>Advertised window</th>
<th>Checksum</th>
<th>Urgent pointer</th>
<th>Options (variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYN</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RST</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>URG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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

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) → Server

Active
Open

connect() → SYN, SeqNum = x

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

ACK, Ack = y + 1

Passive
Open

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
  - 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…
- 3-6 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 **FIN** with their ack of A’s **FIN**

**Timeout:**
Avoid reincarnation
Can retransmit
FIN ACK if ACK lost

Connection now closed
A sends a RESET (\textbf{RST}) to B
- E.g., because app. process on A \textbf{crashed}

\textbf{That’s it}
- B does \textbf{not} ack the \textbf{RST}
- Thus, \textbf{RST} is \textbf{not} delivered \textbf{reliably}
- And: any data in flight is \textbf{lost}
- But: if B sends anything more, will elicit another \textbf{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

Timeout too long → inefficient

Timeout 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 \( \rightarrow \) \text{SampleRTT} = \text{RTT}
Problem: Ambiguous Measurements

How do we differentiate between the real ACK, and ACK of the retransmitted packet?
Karn/Partridge Algorithm

Measure *SampleRTT* only for original transmissions
- Once a segment has been retransmitted, do not use it for any further 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 $RTO \leftarrow 2 \cdot RTO$
  - (Up to maximum $\geq 60$ sec)
- Every time new measurement comes in (= successful original transmission), collapse RTO back to $2 \times EstimatedRTT$
This is all very interesting, but.....

Implementations often use a coarse-grained timer
  • 500 msec is typical

So what?
  • Above algorithms are largely irrelevant
  • **Incurring a timeout is expensive**

So we rely on 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, ...
“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