Communication Networks
Spring 2020

Laurent Vanbever
nsg.ee.ethz.ch

ETH Zürich (D-ITET)
April 6 2020

Materials inspired from Scott Shenker, Jennifer Rexford, and Sharon Goldberg
Last week on

Communication Networks
Border Gateway Protocol

policies and more

1. BGP Policies
   Follow the Money

2. Protocol
   How does it work?

3. Problems
   security, performance, …
Border Gateway Protocol
policies and more

1  BGP Policies
   Follow the Money

Protocol
How does it work?

Problems
security, performance, ...
The Internet topology is shaped according to *business* relationships.
There are 2 main business relationships today:

- customer/provider
- peer/peer

Many less important ones (siblings, backups,...)
There are 2 main business relationships today:

- customer/provider
- peer/peer
Customers pay providers to get Internet connectivity
Peers don’t pay each other for connectivity, they do it *out of common interest*

DT and ATT exchange *tons* of traffic. they save money by directly connecting to each other
Business relationships conditions

route selection

For a destination $p$, prefer routes coming from

- customers over
- peers over
- providers

route type
Business relationships conditions

route exportation

send to

customer peer provider

customer

from peer

provider
Routes coming from customers are propagated to everyone else.

From peer to:
- Customer ✓
- Peer ✓
- Provider ✓

Send to:
- Customer ✓
- Peer ✓
- Provider ✓
Routes coming from peers and providers are only propagated to customers

<table>
<thead>
<tr>
<th></th>
<th>send to</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>customer</td>
</tr>
<tr>
<td>from</td>
<td></td>
</tr>
<tr>
<td>customer</td>
<td>✓</td>
</tr>
<tr>
<td>peer</td>
<td>✓</td>
</tr>
<tr>
<td>provider</td>
<td>✓</td>
</tr>
</tbody>
</table>
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)
Follow the Money

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, now looking at the transport layer.
But first…

Let's finish BGP
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>
The NEXT-HOP is a global attribute which indicates where to send the traffic next.
The NEXT-HOP is set when the route enters/exits an AS, it does not change within the AS.
For externally-learned route, this means that the NEXT-HOP is the IP address of the neighbor's eBGP router, here 10.0.0.1
For this router, reaching 10.0.0.1 is not a problem as it is directly connected to the corresponding subnet (10.0.0.0/30)
That router is *not* directly to the NEXT-HOP subnet (10.0.0.0/30) and does not know how to reach it, it will therefore drop the BGP route…
One solution is for the external router to redistribute the prefixes attached to the external interfaces into the IGP.

announce 10.0.0.0/30 in OSPF
Another solution is for the border router to rewrite the NEXT-HOP before sending it over iBGP, usually to its loopback address.
Of course, **loopback address** need to be reachable network-wide. Typically, each router advertise its loopback (as a /32) in the IGP
This is the infamous **next-hop-self policy**

```
82.130.64.0/18
NEXT-HOP: 10.0.0.1
```

```
82.130.64.0/18
NEXT-HOP: 40.0.0.1
```

```
82.130.64.0/18
NEXT-HOP: 11.0.0.1
```

```
82.130.64.0/18
NEXT-HOP: 40.0.0.2
```
The advantage of next-hop self is to spare the need to advertise *each* prefix attached to an external link in the IGP.

*one* NEXT-HOP, 40.0.0.1, is used to reach routes announced by AS 40, 41, 42, 43.
BGP suffers from many rampant problems

Problems

- Reachability
- Security
- Convergence
- Performance
- Anomalies
- Relevance
<table>
<thead>
<tr>
<th>Problems</th>
<th>Reachability</th>
<th>covered last week</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Security</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Convergence</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Performance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Anomalies</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Relevance</td>
<td></td>
</tr>
</tbody>
</table>
Many **security** considerations are simply **absent** from BGP specifications

ASes can advertise any prefixes even if they don’t own them!

ASes can arbitrarily modify route content *e.g.*, change the content of the AS-PATH

ASes can forward traffic along different paths than the advertised one
BGP (lack of) security

#1 BGP does not validate the origin of advertisements

#2 BGP does not validate the content of advertisements
BGP (lack of) security

#1 BGP does not validate the origin of advertisements

#2 BGP does not validate the content of advertisements
Prefix Hijacking

- **Blackhole**: data traffic is discarded
- **Snooping**: data traffic is inspected, then redirected
- **Impersonation**: traffic sent to bogus destinations
Hijacking is Hard to Debug

• The victim AS doesn’t see the problem
  – Picks its own route, might not learn the bogus route

• May not cause loss of connectivity
  – Snooping, with minor performance degradation

• Or, loss of connectivity is isolated
  – E.g., only for sources in parts of the Internet

• Diagnosing prefix hijacking
  – Analyzing updates from many vantage points
  – Launching traceroute from many vantage points
• Originating a more-specific prefix
  – Every AS picks the bogus route for that prefix
  – Traffic follows the longest matching prefix
BGP (lack of) security

#1 BGP does not validate the origin of advertisements

#2 BGP does not validate the content of advertisements
Bogus AS Paths

- Remove ASes from the AS path
  - E.g., turn “701 3715 88” into “701 88”

- Motivations
  - Attract sources that normally try to avoid AS 3715
  - Help AS 88 look like it is closer to the Internet’s core

- Who can tell that this AS path is a lie?
  - Maybe AS 88 does connect to AS 701 directly
Bogus AS Paths

• Add ASes to the path
  – E.g., turn “701 88” into “701 3715 88”

• Motivations
  – Trigger loop detection in AS 3715
    • Denial-of-service attack on AS 3715
    • Or, blocking unwanted traffic coming from AS 3715!
  – Make your AS look like it has richer connectivity

• Who can tell the AS path is a lie?
  – AS 3715 could, if it could see the route
  – AS 88 could, but would it really care?
Bogus AS Paths

• Adds AS hop(s) at the end of the path
  – E.g., turns “701 88” into “701 88 3”

• Motivations
  – Evade detection for a bogus route
  – E.g., by adding the legitimate AS to the end

• Hard to tell that the AS path is bogus...
  – Even if other ASes filter based on prefix ownership
Invalid Paths

• AS exports a route it shouldn’t
  – AS path is a valid sequence, but violated policy

• Example: customer misconfiguration
  – Exports routes from one provider to another

• Interacts with provider policy
  – Provider prefers customer routes
  – Directing all traffic through customer

• Main defense
  – Filtering routes based on prefixes and AS path
Missing/Inconsistent Routes

• Peers require consistent export
  – Prefix advertised at all peering points
  – Prefix advertised with same AS path length

• Reasons for violating the policy
  – Trick neighbor into “cold potato”
  – Configuration mistake

• Main defense
  – Analyzing BGP updates, or traffic,
  – ... for signs of inconsistency
BGP Security Today

• 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
BGP today 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)

Path Validation using BGPSec

Plain BGP "web of trust" today
RPKI enables to validate the origin of a BGP route by certifying IP prefixes allocations.

RPKI is a database storing Route Origin Authorization (ROA). ROAs map prefix space (130.0.0.0/8–32) to an origin Autonomous System (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](https://tools.ietf.org/html/rfc6480)). Today, RPKI can validate ~19% 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?"
Here, we see that the attack is successful.

Source: Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"
Let's assume now that AS 29997 registers (204.16.254.0/24–32, 29997) as a new ROA.
Using the RPKI, greenhost.nl sees that AS34109 is *not* a valid origin for 204.16.254.40/32
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?

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?"
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.
Plain BGP
"web of trust"
today
Path Validation
using BGPSec

Route Origin Validation
using Resource Public Key Infrastructure (RPKI)
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
S-BGP Secure Version of BGP

• **Address attestations**
  – 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
Problems

Reachability

Security

Convergence

Performance

Anomalies

Relevance

switch back to last week's slides
That's it!

for the network layer, and for now...
We’re continuing our journey up the layers, now 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 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>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>8-bit Time to Live (TTL)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Options (if any)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Payload</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Payload**
<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>8-bit Type of Service (TOS)</th>
<th>16-bit Total Length (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit Identification</td>
<td>3-bit Flags</td>
<td>13-bit Fragment Offset</td>
<td></td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
<td>6 = TCP</td>
<td>16-bit Header Checksum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>17 = UDP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit Source IP Address

32-bit Destination IP Address

Payload
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit Type of Service (TOS)</td>
<td></td>
</tr>
<tr>
<td>16-bit Total Length (Bytes)</td>
<td></td>
</tr>
<tr>
<td>16-bit Identification</td>
<td></td>
</tr>
<tr>
<td>3-bit Flags</td>
<td></td>
</tr>
<tr>
<td>13-bit Fragment Offset</td>
<td></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>16-bit Header Checksum</td>
<td></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>16-bit Source Port</td>
<td></td>
</tr>
<tr>
<td>16-bit Destination Port</td>
<td></td>
</tr>
<tr>
<td>Payload</td>
<td></td>
</tr>
</tbody>
</table>
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”)

<table>
<thead>
<tr>
<th>SRC port</th>
<th>DST port</th>
</tr>
</thead>
<tbody>
<tr>
<td>checksum</td>
<td>length</td>
</tr>
<tr>
<td>DATA</td>
<td></td>
</tr>
</tbody>
</table>
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 (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></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
Segments and Sequence Numbers
TCP “Stream of Bytes” Service…

Application @ Host A

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

Host A

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

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 **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 = 1\textsuperscript{st} 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>
</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>
</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>

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
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 \textit{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>
</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>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>HdrLen</th>
<th>Options (variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</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>

- "Must Be Zero" 6 bits reserved
- Number of 4-byte words in TCP header; 5 = no options

Data
**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>
</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>

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

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
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

<table>
<thead>
<tr>
<th>A’s port</th>
<th>B’s port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A’s Initial Sequence Number</th>
<th>(Irrelevant since ACK not set)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5=20B</td>
</tr>
</tbody>
</table>

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

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

B’s port | A’s port
---|---

<table>
<thead>
<tr>
<th>B’s Initial Sequence Number</th>
<th>ACK = A’s ISN plus 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>20B</td>
<td>Flags</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Advertised window</th>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Options (variable)</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

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

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>Flags:</th>
<th>A’s port</th>
<th>B’s port</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>URG</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**A’s Initial Sequence Number**

<table>
<thead>
<tr>
<th>B’s ISN plus 1</th>
<th>20B</th>
<th>Flags</th>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</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>

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

connect()

SYN, SeqNum = x

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

ACK, Ack = y + 1

Server

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

Connection now **closed**

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 (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
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
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

EstimatedRTT = \alpha \cdot \text{EstimatedRTT} + (1 - \alpha) \cdot \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, ...
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