Communication Networks
Spring 2017

Laurent Vanbever
www.vanbever.eu

ETH Zürich (D-ITET)
May, 8 2017

Material inspired from Scott Shenker & Jennifer Rexford
Two weeks ago on Communication Networks
Routing Security
Routing security
attacks & mitigation

intra-domain routing

insider

inter-domain routing

in/outsider
Routing security
attacks & mitigation

- intra-domain routing
- inter-domain routing
- insider
- in/outsider
Most of the attacks on intra-domain routing aim at performing Denial-of-Service (DoS) or intercept traffic

<table>
<thead>
<tr>
<th>Interception</th>
<th>eavesdrop on/drop/modify/inject/delay traffic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>steer traffic along paths controlled by the attacker</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DoS</th>
<th>induce churn to overload the routers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>announce/withdraw at fast pace</td>
</tr>
<tr>
<td></td>
<td>floods the routers link-state database</td>
</tr>
<tr>
<td></td>
<td>inject thousands of prefixes</td>
</tr>
<tr>
<td></td>
<td>induce congestion/higher delay</td>
</tr>
<tr>
<td></td>
<td>steer traffic along fewer/low-throughput paths</td>
</tr>
<tr>
<td></td>
<td>prevent reachability</td>
</tr>
<tr>
<td></td>
<td>steer traffic along blackholes or loops</td>
</tr>
</tbody>
</table>
The solution is quite simple:  
**Rely on cryptography!**

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution 1 (light)</th>
<th>Solution 2 (heavy)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bogus advertisements can be injected</td>
<td>Use Cryptographic Authentication (header)</td>
<td>Encrypt the entire advertisement (header/payload)</td>
</tr>
<tr>
<td>Legitimate advertisements can be tampered with</td>
<td>integrity and authentication</td>
<td>integrity, authentication <em>and</em> confidentiality</td>
</tr>
</tbody>
</table>
Routing security
attacks & mitigation

intra-domain routing
insider

inter-domain routing
in/outsider
BGP (lack of) security:
problems & solutions

#1 BGP does not validate the origin of advertisements

#2 BGP does not validate the content of advertisements

#3 Proposed Enhancements

#4 What about the data plane?

#5 What’s the Internet to do anyway?
BGP (lack of) security: problems & solutions

#1  BGP does not validate the origin of advertisements

#2  BGP does not validate the content of advertisements

#3  Proposed Enhancements

#4  What about the data plane?

#5  What’s the Internet to do anyway?
Prefix Hijacking

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

- Originating a more-specific prefix
  - Every AS picks the bogus route for that prefix
  - Traffic follows the longest matching prefix
BGP (lack of) security:
problems & solutions

#1 BGP does not validate the origin of advertisements

#2 BGP does not validate the content of advertisements

#3 Proposed Enhancements

#4 What about the data plane?

#5 What’s the Internet to do anyway?
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 is 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
BGP (lack of) security:
problems & solutions

#1 BGP does not validate the origin of advertisements

#2 BGP does not validate the content of advertisements

#3 Proposed Enhancements

#4 What about the data plane?

#5 What’s the Internet to do anyway?
Secure BGP

Origin Authentication + cryptographic signatures

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

\[ a_1: (v, \text{Prefix}) \]

\[ m: (a_1, v, \text{Prefix}) \]
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
This week on Communication Networks
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 (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>4-bit Version</th>
<th>4-bit Header Length</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>8-bit Protocol</td>
<td>16-bit Header Checksum</td>
<td></td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (if any)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Payload
<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<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>Protocol</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>8-bit Time to Live (TTL)</th>
<th>6 = TCP</th>
<th>17 = UDP</th>
<th>16-bit Header Checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit Source IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td></td>
<td></td>
<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>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>17 = UDP</td>
<td></td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit Source Port</td>
<td>16-bit Destination Port</td>
<td></td>
<td></td>
</tr>
<tr>
<td>More transport header fields ....</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Payload</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Connection Mappings

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

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

Why the difference?

Implications for mobility

Why do you need to include local IP?
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”)

```
+---------+---------+
| SRC port| DST port|
+---------+---------+
| checksum| length  |
+---------+---------+
|         |         |
|         | DATA    |
+---------+---------+
```
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 \textit{(previously, but quick review)}
- Ensures byte stream (eventually) arrives intact
  - In the presence of \textit{corruption and loss}

Connection oriented \textit{(today)}
- Explicit set-up and tear-down of TCP session

Full duplex stream-of-bytes service \textit{(today)}
- Sends and receives a stream of bytes, not messages

Flow control \textit{(previously, but quick review)}
- Ensures that sender doesn’t overwhelm receiver

Congestion control \textit{(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>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source port</td>
<td>Destination port</td>
</tr>
<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>Flags</td>
<td></td>
</tr>
<tr>
<td>Advertised window</td>
<td></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>
# 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>0</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>

Data

*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

Host A

ISN (initial sequence number)

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

k bytes
Sequence Numbers

Host A

ISN (initial sequence number)

k

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

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

<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

Acknowledgment gives seqno just beyond highest seqno received in order

“What Byte is Next”
### 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>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HdrLen</th>
<th>Flags</th>
<th>Options (variable)</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>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
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 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**

Allow a larger amount of data “in flight”

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

![Diagram showing Sliding Window](image-url)
Sliding Window

For the sender, when receives an acknowledgment for new data, window advances (*slides* forward)

![Diagram of Sliding Window](image.png)
Sliding Window

For the sender, when receives an acknowledgment for new data, window advances (*slides* forward)
For the receiver, as the receiving process consumes data, the window slides forward.
Sliding Window

For the receiver, as the receiving process consumes data, the window slides forward.
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>

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

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

```
<table>
<thead>
<tr>
<th>HdrLen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</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></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>

Data

---

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></td>
<td></td>
</tr>
<tr>
<td>Sequence number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td><strong>HdrLen</strong></td>
<td><strong>Flags</strong></td>
</tr>
<tr>
<td>0</td>
<td><strong>Urgent pointer</strong></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>Sequence number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>HdrLen 0 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>

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

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

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

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

| A’s Initial Sequence Number |
| (Irrelevant since ACK not set) |

| Advertised window |
| 5=20B |

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

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

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

<table>
<thead>
<tr>
<th>Flags:</th>
<th>SYN</th>
<th>ACK</th>
</tr>
</thead>
<tbody>
<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>

- **B’s port**
- **A’s port**
- **B’s Initial Sequence Number**
- **ACK = A’s ISN plus 1**
- **20B**
- **0**
- **Flags**
- **Advertised window**
- **Checksum**
- **Urgent pointer**
- **Options (variable)**

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’s Initial Sequence Number

B’s ISN plus 1

Advertised window

Flags: SYN ACK FIN RST PSH URG

Checksum

Urgent pointer

Options (variable)

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):
- 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
Abrupt Termination

A sends a RESET (RST) to B
- E.g., because app. process on A crashed

That’s it
- B does not ack the RST
- 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
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

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

Assume RTT is constant $\Rightarrow$ $SampleRTT = RTT$
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 $2 \times EstimatedRTT$
Karn/Partridge in action

Figure 5: Performance of an RFC793 retransmit timer

from Jacobson and Karels, SIGCOMM 1988
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