IERG 4130 – 2026 Fall

香港中文大學
CUHK

BGP and the Transport Layer

Yajin ZHOU

https://yajin.org

Credits: the slides are from CS161 of UC Berkeley

Last Time: Low-Level Network Attacks



- **Classes of attackers:**
  - Off-path: Can't see, modify, or drop packets
  - On-path: Can see packets, but can't modify or drop packets
  - MITM: Can see, modify, and drop packets
- **ARP: A protocol to translate local IP addresses to MAC addresses**
  - Ask everyone on the network, "Who has the IP 1.2.3.4?"
  - Attack: The attacker can respond instead of the true device with 1.2.3.4, and packets will get routed to the attacker!
  - Defense: Switches
  - Defense: Rely on higher layers
- **DHCP: A protocol for a new client to receive a network configuration**
  - Ask everyone on the network, "What is the network configuration to use?"
  - Attack: The attacker can respond with a malicious configuration
  - Defense: Rely on higher layers

Types of Network Attackers

- **Threat model: There are 3 types of attackers we'll consider**

	Can modify or delete packets	Can read packets	Can inject spoofed packets
Man-in-the-middle attacker	✓	✓	✓
On-path attacker		✓	✓
Off-path attacker			✓

Layers 3 and 4

- Border Gateway Protocol (BGP): Routing -- determining what path packets should take
- **Transmission Control Protocol (TCP): Reliably sending packets**
- **User Datagram Protocol (UDP): Non-reliably sending packets**



Border Gateway Protocol (BGP)

Review: Internet Protocol (IP)



- **Internet Protocol** (**IP**): The universal layer-3 protocol that all devices use to transmit data over the Internet
- **IP address: An address that identifies a device on the Internet**
  - IPv4 is 32 bits (e.g. **35.163.72.93**)
  - IPv6 is 128 bits (e.g. **2607:f140:8801:0000:0000:0000:0001:0023**)
    - Shorthand: omit sets of zeros: **2607:f140:8801::1:23**
  - Globally unique from any single perspective
    - For now, you can think of them as just being globally unique
  - IP addresses help nodes make decisions on where to forward the packet

Subnets

- Recall: An IP packet might make many hops across different routers before it can reach its destination
- **IP routes by subnets, which are groups of addresses with a common prefix**
  - A subnet is written as a prefix followed by the length of the prefix (in bits)
    - Example: **128.32.0.0/16** is an IPv4 subnet with all addresses that begin with the prefix of **128.32.***
    - Since an IPv4 is a 32-bit address and there are 16 bits in the prefix, this subnet represents $2^{32 - 16} = 2^{16}$ addresses

Routing Packets

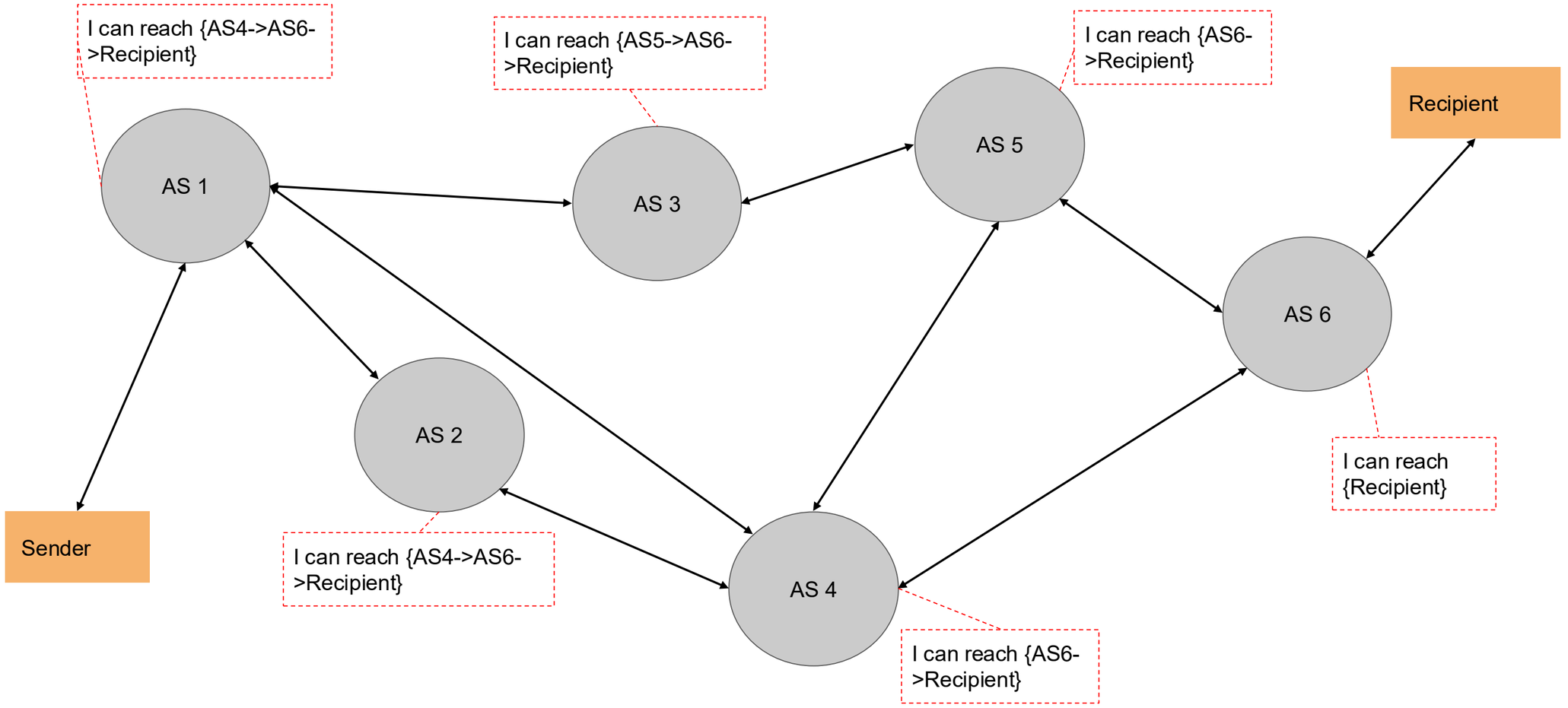
- **To send a packet to a computer within the local network:**
  - Verify that the destination IP is in the same subnet
  - Use ARP (or contact a switch) to get the destination MAC address
  - Send the packet directly to the destination using the destination MAC address
- **To send a packet to a computer that is not within the local network:**
  - Send the packet to the **gateway router**
    - **Quick question: how do you know the gateway IP and MAC?**
  - Past the gateway router, the packet goes to the Internet
  - It's the gateway router's job to deliver the packet closer to the destination

Autonomous Systems



- Once your system sends the packet to the gateway, the packet has to be routed through the Internet
- **The Internet is a network of networks, comprised of many autonomous systems (AS)**
  - Each AS handles its own internal routing
  - Each AS is uniquely identified by its **autonomous system number** (**ASN**)
  - Each AS is comprised of one or more LANs
  - Each AS can forward packets to other connected ASes
- The protocol for communicating between different Autonomous Systems is **Border Gateway Protocol** (**BGP**)
  - Each router announces what networks it can deliver to and the path onward from the router
  - The most precise route with the shortest path and no loops is the preferred route

BGP


I can reach {AS4->AS6->Recipient}
I can reach {AS5->AS6->Recipient}
I can reach {AS6->Recipient}
Recipient
AS 1
AS 3
AS 5
AS 6
AS 2
AS 4
Sender
I can reach {AS4->AS6->Recipient}
I can reach {AS6->Recipient}
I can reach {Recipient}

BGP

- **We have around 83K ASes**

address.

8.8.8.8 | Next example

Examples: AS7018 - AS16509 - AS714 - AS749 - AS7377 - AS10995

Copy


```
{
  "ip": "8.8.8.8",
  "is_bogon": false,
  "company": "Google LLC",
  "asn": "AS15169 Google LLC",
  "city": "Mountain View",
  "region": "California",
  "country": "United States",
  "lat": 37.38605,
  "lon": -122.08385,
  "timezone": "America/Los_Angeles",
  "docs": "https://ipapi.is/free-tier.html?ref=api"
}
```

IP and BGP Attacks



- **Each AS implicitly trusts the surrounding ASes and accepts advertised routes**
- **IP spoofing**: Malicious clients can send IP packets with source IP values set to a spoofed value
  - Enables packets that look like they're coming from someone else!
  - Edge ASes should block packets with source IPs set to the wrong value, but some don't
  - We rely on defenses provided by higher layers to further prevent this ("defense in depth")

IP and BGP Attacks



- **BGP hijacking**: A malicious autonomous system can lie and claim itself to be responsible for a network which it isn't
  - Example: AS3 broadcasts that it is responsible for 128.32.0.0/16
    - Now, the malicious AS can act as a MITM for traffic to 128.32.0.0!
  - No real defenses on this level, so we rely on defenses from higher levels

Real Example: The 2008 Pakistan Telecom YouTube BGP Hijack

- In February 2008, an attempt by Pakistan Telecom to block YouTube domestically accidentally took YouTube offline across the entire globe for roughly two hours.

- How It Happened:

  - **Domestic Censorship Order:** On February 24, 2008, the Pakistani government ordered local ISPs to block access to YouTube. To comply, **Pakistan Telecom (AS17557)** configured **a "blackhole"** route inside its internal network to drop traffic destined for YouTube's IP block (208.65.152.0/22).

  - **More Specific Prefix Configuration:** To ensure the block worked reliably, an engineer advertised a more specific subnet: 208.65.153.0/24 (a /24 prefix covers 256 IP addresses, whereas YouTube's official /22 prefix covers 1,024 IP addresses).

Real Example: The 2008 Pakistan Telecom YouTube BGP Hijack

- How It Happened:
  - **Accidental External Broadcast:** The engineer forgot to restrict this route to internal use only and advertised it via **BGP** to its upstream international transit provider, **PCCW Global (AS3491)** in Hong Kong.
  - **Longest Prefix Match Rule:** PCCW failed to filter out the unauthorized announcement and propagated it to other Tier-1 global carriers. Because BGP routers always prioritize the **most specific route (Longest Prefix Match)**, routers worldwide saw /24 as a more exact path than YouTube's /22
  - **Global Outage:** Within minutes, **nearly all global traffic meant for YouTube was rerouted into Pakistan Telecom's network**, instantly overwhelming its links and rendering YouTube inaccessible worldwide until YouTube countered by advertising even more specific /25 prefixes..



Transmission Control Protocol (TCP)

Review: IP Reliability



- **IP is unreliable and only provides a best effort delivery service, which means:**
  - Packets may be lost (“dropped”)
  - Packets may be corrupted
  - Packets may be delivered out of order
- It is up to higher level protocols to ensure that the connection is reliable, i.e., that packets are received correctly
  - Packets that are changed (random error or malicious tampering) aren't considered correctly received
  - IP packets include a checksum (unkeyed function, protects against random errors)
  - However, there is no cryptographic MAC, so there are no guarantees if an attacker maliciously modifies packets (and modifies the checksum accordingly)

Scratchpad: Let's Design It Together



- **Problem: IP packets have a limited size. How do we send long messages?**
  - To send longer messages, we have to manually **break messages into packets**
  - When sending packets: TCP will automatically split up messages
  - When receiving packets: TCP will automatically reassemble the packets
  - Now the user doesn't need to manually split up messages!
- **Problem: Packets can arrive out of order**
  - When sending packets: TCP **labels each byte of the message with increasing numbers**
  - When receiving packets: TCP can use the numbers to rearrange bytes in the correct order

Scratchpad: Let's Design It Together



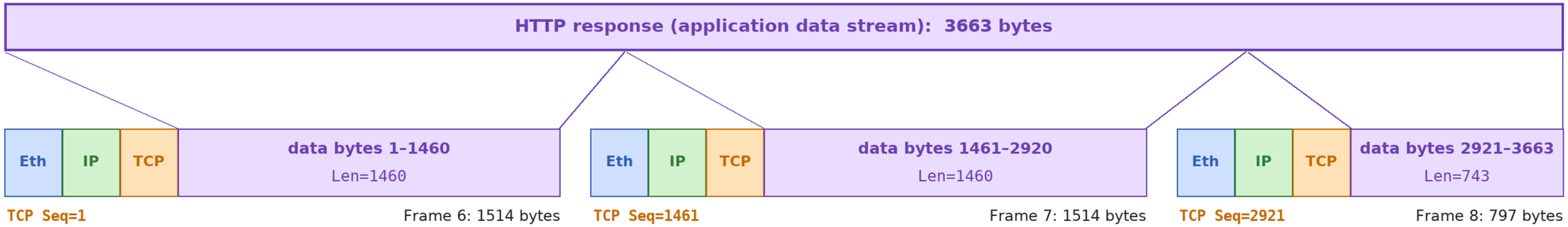
- **Problem: Packets can be dropped**
  - When receiving packets: TCP sends **an extra message acknowledging that a packet has been received**
  - When sending packets: If the acknowledgement doesn't arrive, re-send the packet

What about application writes lots of data

- TCP will do segmentation according to MSS (maximum segment size)

TCP segmentation: one 3663-byte HTTP response, three TCP segments

The server's application writes 3663 bytes. TCP cuts them into pieces of at most MSS = 1460 bytes (1500-byte Ethernet MTU − 20 B IP − 20 B TCP).



What Wireshark shows (http-get.pcap):

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	1.2.3.4	5.6.7.8	TCP	66	1234 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460
2	0.000579	5.6.7.8	1.2.3.4	TCP	66	80 → 1234 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460
3	0.001006	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [ACK] Seq=1 Ack=1 Win=502 Len=0
4	0.001438	1.2.3.4	5.6.7.8	HTTP	124	GET / HTTP/1.1
5	0.001925	5.6.7.8	1.2.3.4	TCP	54	80 → 1234 [ACK] Seq=1 Ack=71 Win=509 Len=0
6	0.002336	5.6.7.8	1.2.3.4	TCP	1514	80 → 1234 [ACK] Seq=1 Ack=71 Len=1460 [TCP segment of a reassembled PDU]
7	0.002927	5.6.7.8	1.2.3.4	TCP	1514	80 → 1234 [ACK] Seq=1461 Ack=71 Len=1460 [TCP segment of a reassembled PDU]
8	0.003405	5.6.7.8	1.2.3.4	HTTP	797	HTTP/1.1 200 OK (text/html)
9	0.004037	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [ACK] Seq=71 Ack=3664 Win=502 Len=0

Seq of each segment = Seq of the previous one + its Len (1 → 1461 → 2921). The client's Ack=3664 means "I have every byte up to 3663".

Wireshark labels the first two "TCP segment of a reassembled PDU" and shows the whole HTTP response at frame 8, after reassembly.

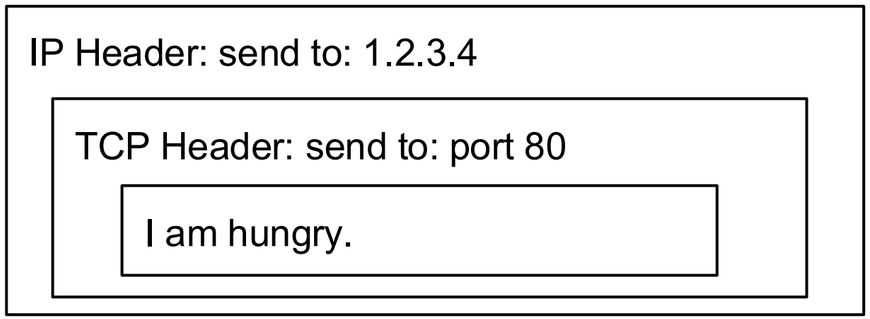
TCP



- **Abstraction: a byte stream**
  - Bytes go in one end of the stream at the source and come out at the other end at the destination
  - TCP automatically breaks streams into **segments**, which are sent as layer 3 packets
- **Provides ordering**
  - Segments contain sequence numbers, so the destination can reassemble the stream in order
- **Provides reliability**
  - The destination sends **acknowledgements** (ACKs) for each sequence number received
  - If the source doesn't receive the ACK, the source sends the packet again
- **Provides ports**
  - Multiple services can share the same IP address by using different ports

Ports

- **Ports** help us distinguish between different applications on the same computer or server
  - On private computers, port numbers can be random
  - On public servers, port numbers should be constant and well-known (so users can access the right port)
- Remember: TCP is built on top of IP, so the IP header (and therefore the IP address) is still present



Establishing Sequence Numbers



- **Each TCP connection requires two sets of sequence numbers**
  - One sequence number for messages from the client to the server
  - One sequence number for messages from the server to the client
- Before starting a TCP connection, the client and server must agree on two **initial sequence numbers** (ISNs)
  - The ISNs are **different and random for every connection** (for security reasons)

H	e	l	l	o		s	e	r	v	e	r
50	51	52	53	54	55	56	57	58	59	60	61

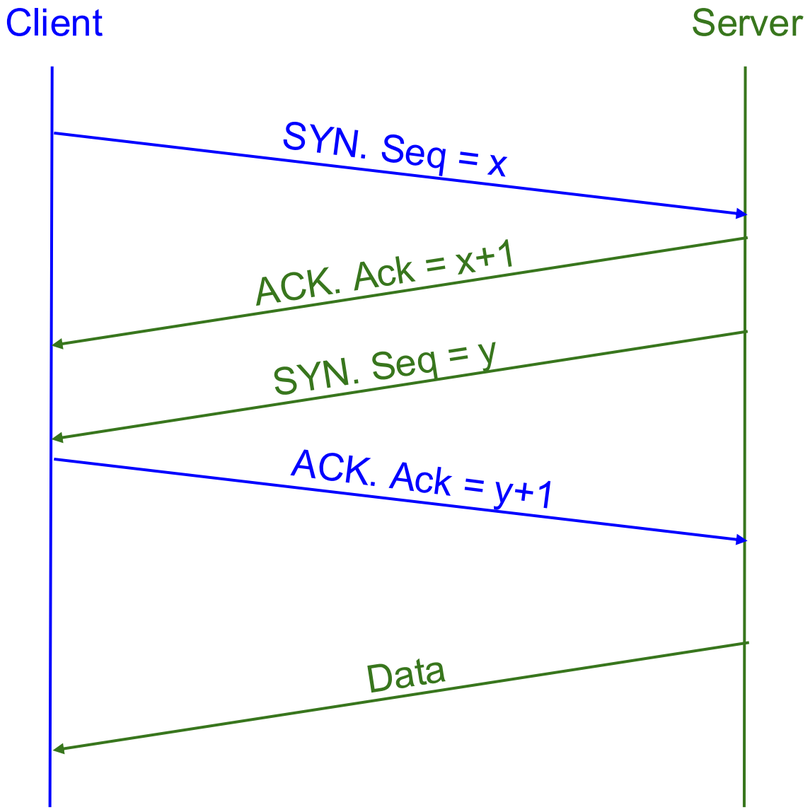
Messages from the client are numbered starting at 50.

H	e	l	l	o		c	l	i	e	n	t
25	26	27	28	29	30	31	32	33	34	35	36

Messages from the server are numbered starting at 25.

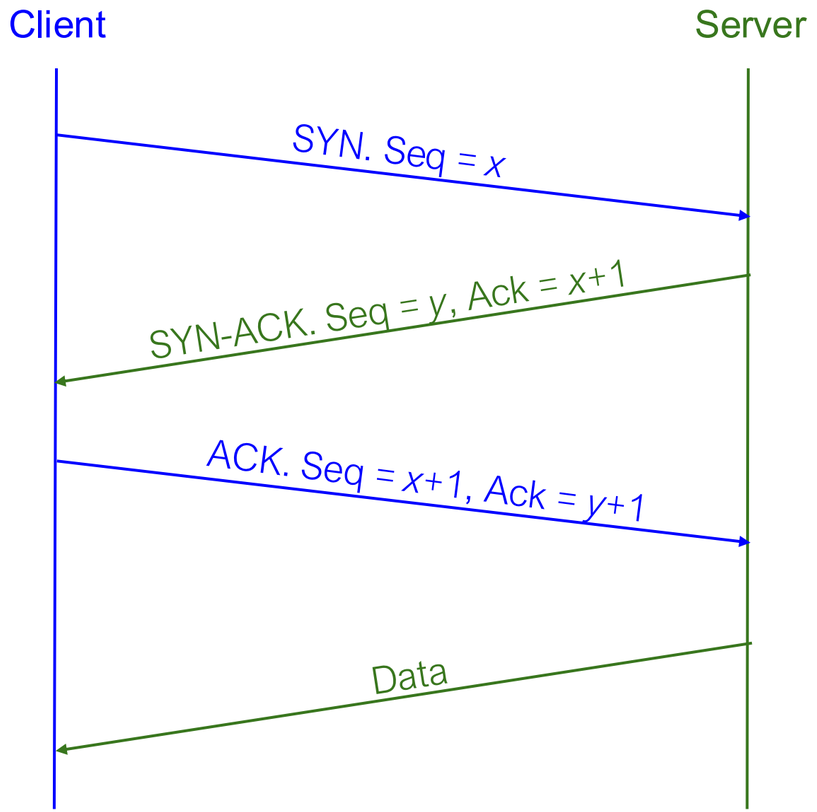
TCP: Conceptual Idea of 3-Way Handshake

1. Client chooses an initial sequence number *x* for bytes it will send and sends a SYN (synchronize) packet to the server
2. **Server acknowledges receipt by returning *x*+1**
3. Server chooses an initial sequence number *y* for bytes it will send and sends a SYN packet
4. **Client acknowledges with an ACK packet**
5. Once both hosts have synchronized sequence numbers, the connection is “established”



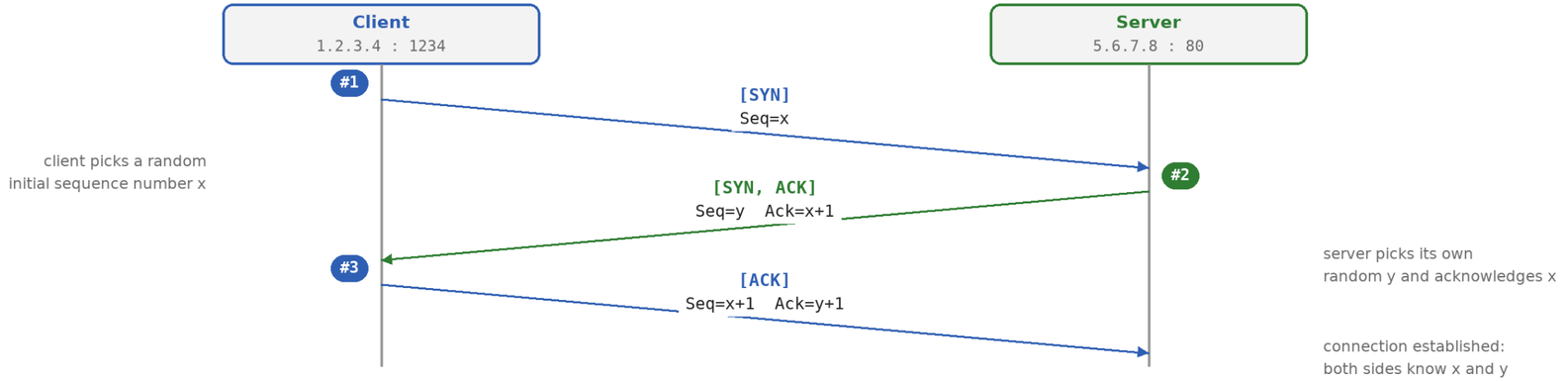
TCP: 3-Way Handshake (Actual version)

1. Client chooses an initial sequence number *x* for bytes it will send and sends a SYN (synchronize) packet to the server
2. Server chooses an initial sequence number *y* for bytes it will send and responds with a SYN-ACK packet
3. **Client acknowledges with an ACK packet**
4. Once both hosts have synchronized sequence numbers, the connection is “established”



TCP: 3-Way Handshake (Actual version)

The TCP three-way handshake (tcp-session.pcap, frames 1–3)



In this capture: x = 3052814921, y = 1904226517 (Wireshark shows them as 0: "relative sequence numbers")

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	1.2.3.4	5.6.7.8	TCP	66	1234 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM WS=128
2	0.012303	5.6.7.8	1.2.3.4	TCP	66	80 → 1234 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM WS=128
3	0.012361	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [ACK] Seq=1 Ack=1 Win=502 Len=0

The SYN of frame 1, as Wireshark decodes it:

```
▾ Transmission Control Protocol, Src Port: 1234, Dst Port: 80, Seq: 0, Len: 0
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 3052814921
    Acknowledgment Number: 0
    Acknowledgment number (raw): 0    (no ACK flag yet, so this field is ignored)
    1000 .... = Header Length: 32 bytes (8)
  ▾ Flags: 0x002 (SYN)
      .... .... ..1. = Syn: Set
    Window: 64240
  ▾ Options: (12 bytes), Maximum segment size, SACK permitted, No-Operation (NOP), Window scale
      Maximum segment size: 1460 bytes
      SACK permitted
      Window scale: 7 (multiply by 128)
```

- SYN and FIN each use up one sequence number, although they carry no data: that is why the ACK is x+1, not x.
- The initial sequence numbers are random so that an off-path attacker cannot guess them (see the spoofing figure).
- The SYN also tells the server how large a segment the client accepts (MSS); MSS and window scale appear only in SYN segments.

TCP: Sending and Receiving Data



- The TCP software on each side tracks which TCP segments have been received for each connection
  - A connection is identified by these 5 values (sometimes called a 5-tuple)
    - Source IP
    - Destination IP
    - Source Port
    - Destination Port
    - Protocol (TCP)

TCP: Sending and Receiving Data



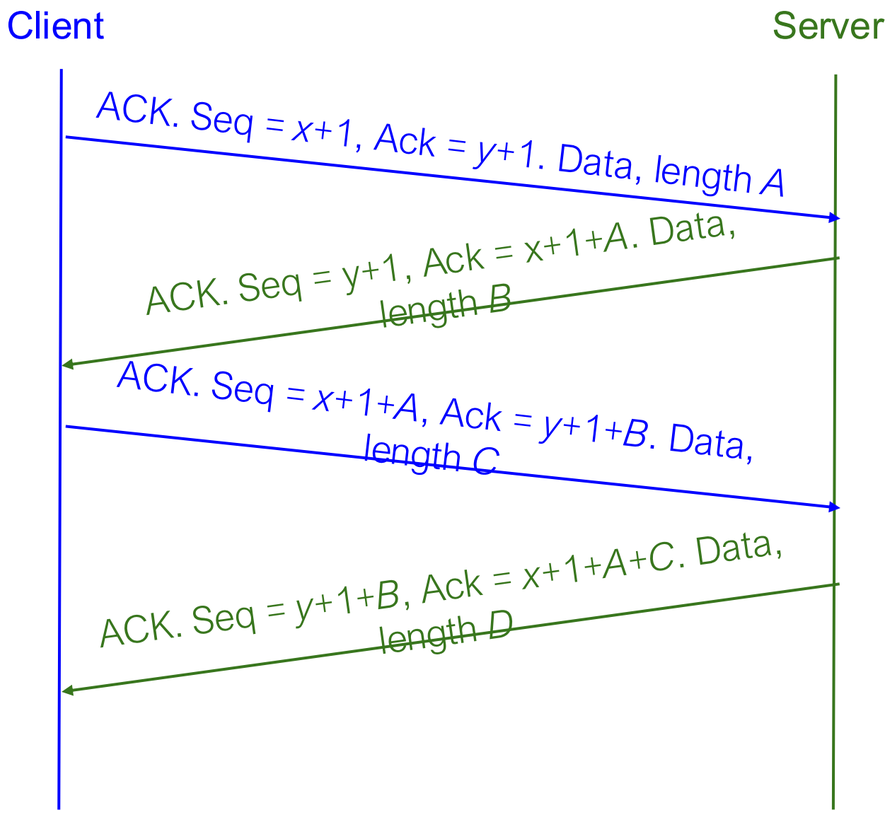
- Data from the bytestream can be presented to the application when all data before has been received and presented
  - Recall: TCP presents data to the application as a bytestream, so the order must be preserved from one end to the other, even if packets are received out of order

TCP: Sending and Receiving Data



- **Byte *i* of the bytestream is represented by sequence number *x* + *i***
  - The first byte is byte *i* = 1, since sequence number *x* was used for the SYN packet and *y* for the SYN-ACK packet
- **A packet's sequence number is the sequence number of the first byte of its data**
  - This number is from the sender's set of sequence numbers
- A packet's ACK number, if the ACK flag is set, is the sequence number of the byte immediately after the last received byte
  - This number is from the set of sequence numbers from the party who sent the data
  - This would be (sequence number) + (length of data) for the last received packet

TCP: Sending and Receiving Data

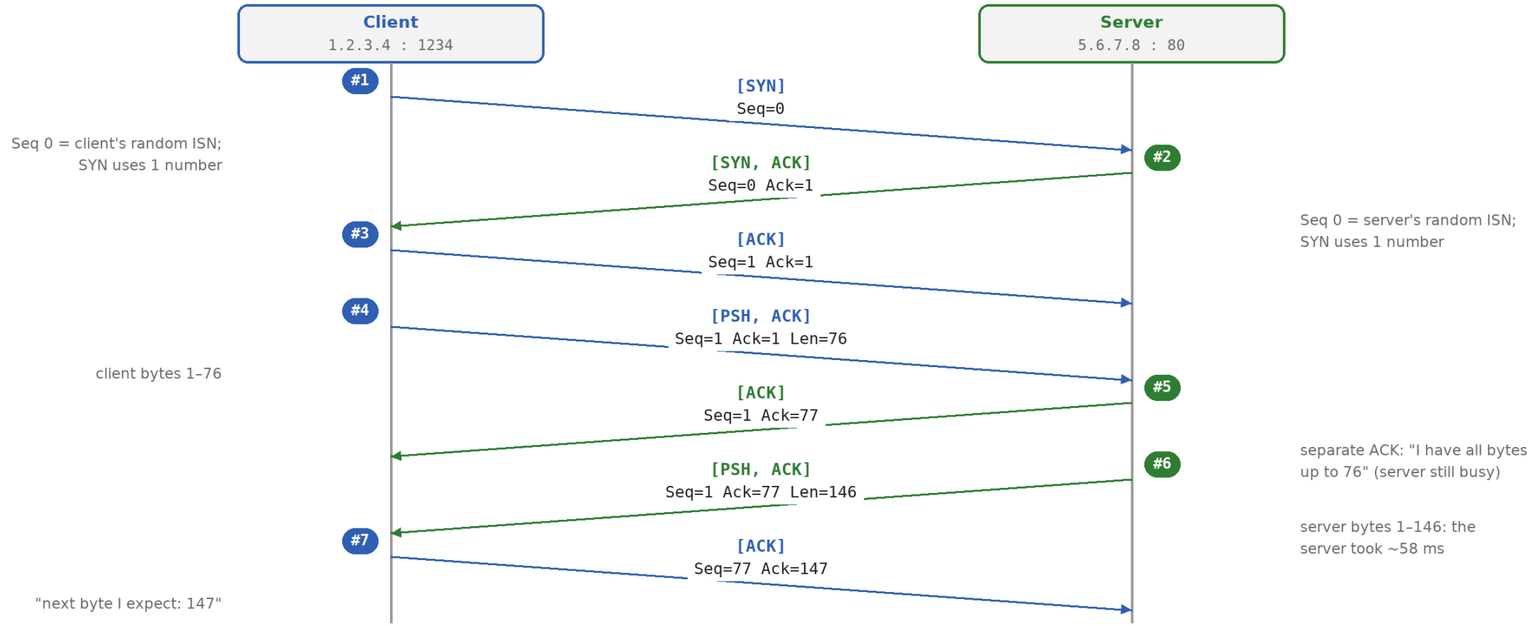


TCP:

Sequence and acknowledgement numbers count bytes

Numbers are relative, as Wireshark shows them: each side's random initial sequence number (ISN) is shown as 0.

Real ISNs in this capture: client 3052814921, server 1904226517. Raw number = ISN + relative number (mod 2^32).



	Meaning
Seq	number of the first data byte in this segment (relative: the first data byte after the SYN is 1)
Len	number of data bytes in this segment
Ack	the next byte the sender expects from the other side = all bytes before it have arrived
Rule	next Seq from the same side = this Seq + Len (+1 for a SYN or a FIN)

Worked through frames 4–7:

- Frame 4: the client sends the request, Seq=1, Len=76 → its bytes 1 … 76.
- Frame 5: the server acknowledges with Ack=1+76 = 77.
- Frame 6: the server sends its response, Seq=1, Len=146; it still says Ack=77 (nothing new from the client).
- Frames 5 and 6 are separate: TCP delays an ACK for up to ~40 ms (Linux), hoping to send it with data. This server took ~58 ms to answer, so the ACK went alone (frame 5, about 52 ms after the request). A faster server would put the ACK in the data segment.
- Frame 7: the client's Seq is now 77 (= 1 + 76) and it acknowledges Ack=147 (= 1 + 146).
- Each direction is numbered separately, starting from that side's own random initial sequence number.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	1.2.3.4	5.6.7.8	TCP	66	1234 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM WS=128
2	0.012303	5.6.7.8	1.2.3.4	TCP	66	80 → 1234 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM WS=128
3	0.012361	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [ACK] Seq=1 Ack=1 Win=502 Len=0
4	0.012448	1.2.3.4	5.6.7.8	HTTP	130	GET /grades HTTP/1.1
5	0.064902	5.6.7.8	1.2.3.4	TCP	54	80 → 1234 [ACK] Seq=1 Ack=77 Win=509 Len=0
6	0.083120	5.6.7.8	1.2.3.4	HTTP	200	HTTP/1.1 200 OK (text/html)
7	0.083171	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [ACK] Seq=77 Ack=147 Win=502 Len=0

TCP: Retransmission



- **If a packet is dropped (lost in transit):**
  - The recipient will not send an ACK, so the sender will not receive the ACK
  - The sender repeatedly tries to send the packet again until it receives the ACK
- **If a packet is received, but the ACK is dropped:**
  - The sender tries to send the packet again since it didn't receive the ACK
  - The recipient ignores the duplicate data and sends the ACK again
- **When packets are dropped in TCP, TCP assumes that there is congestion and sends the data at a slower rate**

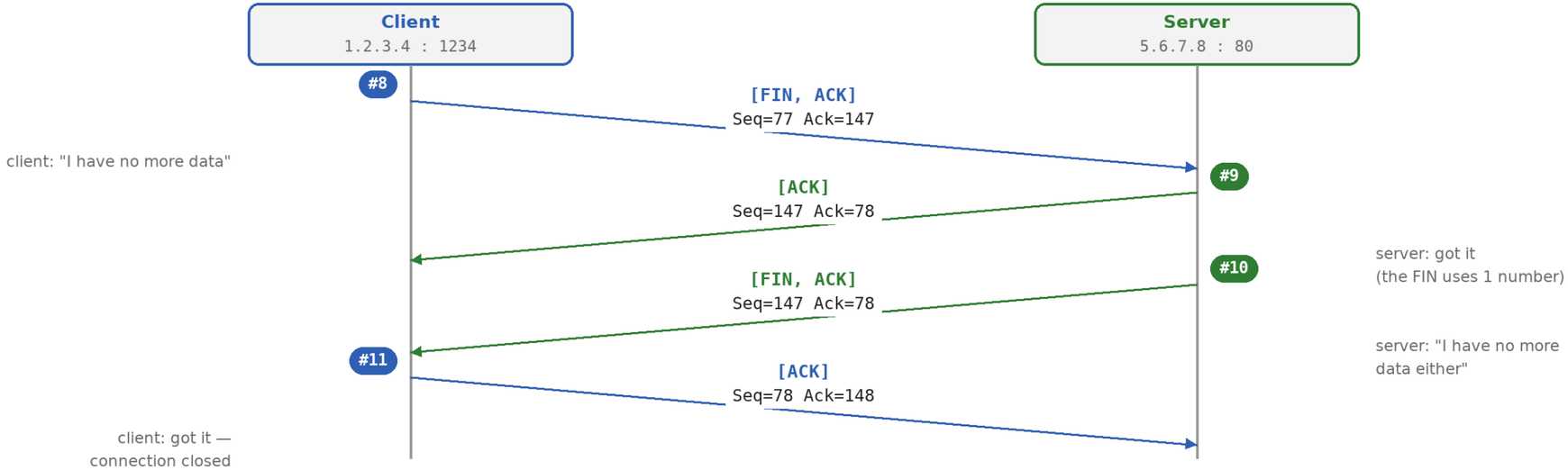
TCP: Ending/Aborting a Connection



- To **end** a connection, one side sends **a packet with the FIN (finish) flag set**, which should then be acknowledged
  - This means “I will no longer be sending any more packets, but I will continue to receive packets”
  - Once the other side is no longer sending packets, it sends a packet with the FIN flag set
- **To abort a connection, one side sends a packet with the RST (reset) flag set**
  - This means “I will no longer be sending nor receiving packets on this connection”
  - RST packets are not acknowledged since they usually mean that something went wrong

TCP: Ending/Aborting a Connection

Closing a connection gracefully with FIN (tcp-session.pcap, frames 8–11)



- Each direction is closed separately: a FIN means "no more data from me", the other side may still send.
- A FIN uses up one sequence number, like a SYN: the client's FIN has Seq=77, so the server answers Ack=78.
- Often the server sends its ACK and its FIN in one segment [FIN, ACK]; then the close takes three segments, not four.
- Frame 8 comes 1.4 s after frame 7: closing is triggered by the application, not by TCP.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	1.2.3.4	5.6.7.8	TCP	66	1234 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM WS=128
2	0.012303	5.6.7.8	1.2.3.4	TCP	66	80 → 1234 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM WS=128
3	0.012361	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [ACK] Seq=1 Ack=1 Win=502 Len=0
4	0.012448	1.2.3.4	5.6.7.8	HTTP	130	GET /grades HTTP/1.1
5	0.064902	5.6.7.8	1.2.3.4	TCP	54	80 → 1234 [ACK] Seq=1 Ack=77 Win=509 Len=0
6	0.083120	5.6.7.8	1.2.3.4	HTTP	200	HTTP/1.1 200 OK (text/html)
7	0.083171	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [ACK] Seq=77 Ack=147 Win=502 Len=0
8	1.459110	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [FIN, ACK] Seq=77 Ack=147 Win=502 Len=0
9	1.471350	5.6.7.8	1.2.3.4	TCP	54	80 → 1234 [ACK] Seq=147 Ack=78 Win=509 Len=0
10	1.471370	5.6.7.8	1.2.3.4	TCP	54	80 → 1234 [FIN, ACK] Seq=147 Ack=78 Win=509 Len=0
11	1.471410	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [ACK] Seq=78 Ack=148 Win=502 Len=0

TCP: Ending/Aborting a Connection

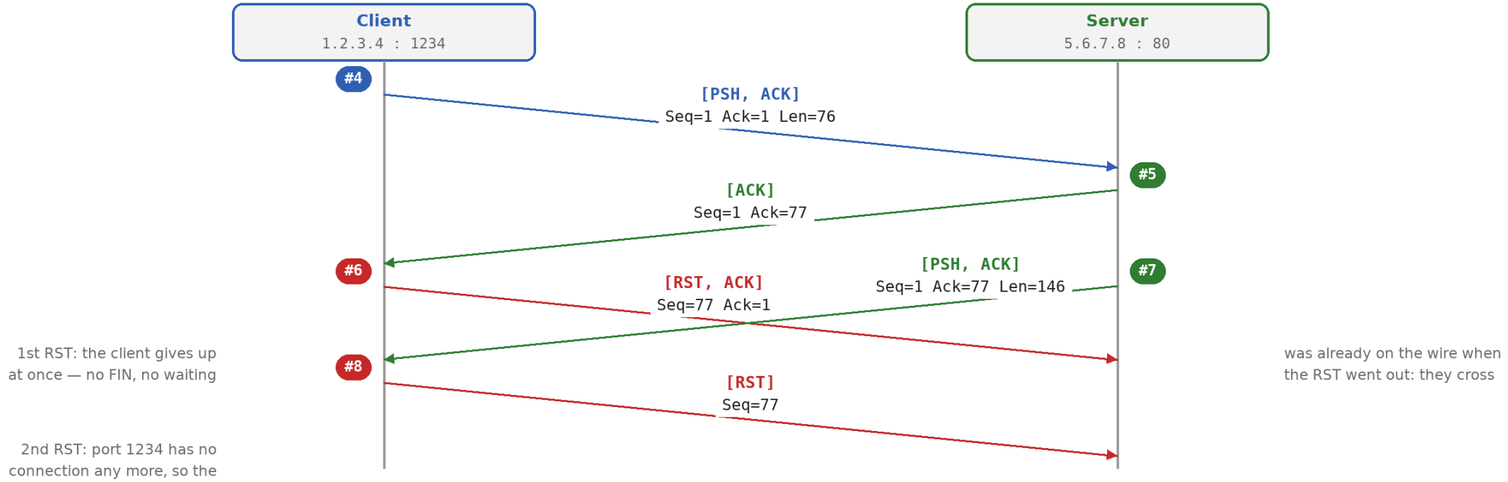
Aborting with RST: two common cases

Case 1 — nothing listens on the port (tcp-rst-closed-port.pcap)

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	1.2.3.4	5.6.7.8	TCP	66	1234 → 81 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM WS=128
2	0.012011	5.6.7.8	1.2.3.4	TCP	54	81 → 1234 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

- The client asks for port 81; the server's operating system has no program there and answers at once with RST.
- This is how a port scanner tells "closed" (RST comes back) from "filtered" (nothing comes back).

Case 2 — the application aborts in the middle (tcp-rst-abort.pcap; handshake = frames 1–3, not drawn)



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	1.2.3.4	5.6.7.8	TCP	66	1234 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM WS=128
2	0.012303	5.6.7.8	1.2.3.4	TCP	66	80 → 1234 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM WS=128
3	0.012361	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [ACK] Seq=1 Ack=1 Win=502 Len=0
4	0.012448	1.2.3.4	5.6.7.8	HTTP	130	GET /grades HTTP/1.1
5	0.024861	5.6.7.8	1.2.3.4	TCP	54	80 → 1234 [ACK] Seq=1 Ack=77 Win=509 Len=0
6	0.025010	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [RST, ACK] Seq=77 Ack=1 Win=0 Len=0
7	0.025902	5.6.7.8	1.2.3.4	HTTP	200	HTTP/1.1 200 OK (text/html)
8	0.025931	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [RST] Seq=77 Win=0 Len=0

- FIN: "I am done sending" — data in flight is still delivered. RST: "this connection is over now" — data in flight is thrown away.
- Both RSTs come from the client. Frame 6 is the abort itself. Frame 8 is not a second abort: it is the automatic answer to frame 7, the server's response that crossed frame 6 on the wire and arrived when the connection was gone.
- Modern stacks (RFC 5961) accept a RST only if its Seq is exactly the next byte they expect (here 77); older stacks accepted any RST inside the receive window. This check is all that protects against RST injection.

TCP Flags



- **ACK**
  - Indicator that the user is acknowledging the receipt of something (in the ack number)
  - Pretty much always set except the very first packet
- **SYN**
  - Indicator of the beginning of the connection
- **FIN**
  - One way to end the connection
  - Requires an acknowledgement
  - No longer sending packets, but will continue to receive
- **RST**
  - One way to end a connection
  - Does not require an acknowledgement
  - No longer sending or receiving packets

7 The TCP flags field: which bits are set in each kind of segment

Reserved	Reserved	Reserved	AE	CWR	ECE	URG	**ACK**	**PSH**	**RST**	**SYN**	**FIN**
bit 11	bit 10	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

12 bits after the header length. The five in orange are the ones this course uses; Wireshark prints the value in hex.

Flags	Value	Meaning	Sent by
SYN	**0x002**	first segment of the handshake	client → server
SYN, ACK	**0x012**	second segment of the handshake	server → client
ACK	**0x010**	acknowledges data; set on every segment after the SYN	both
PSH, ACK	**0x018**	usually on data segments: "deliver to the application now"	both
FIN, ACK	**0x011**	"I have no more data to send" (graceful close)	either side
RST	**0x004**	abort the connection immediately	either side
RST, ACK	**0x014**	abort, e.g. a SYN to a port where nothing listens	usually server

How Wireshark shows the flags of a FIN, ACK segment (frame 8 of tcp-session.pcap):

```
▾ Flags: 0x011 (FIN, ACK)
    000. .... .... = Reserved: Not set
    ...0 .... .... = Accurate ECN: Not set
    .... 0... .... = Congestion Window Reduced: Not set
    .... .0.. .... = ECN-Echo: Not set
    .... ..0. .... = Urgent: Not set
    .... ...1 .... = Acknowledgment: Set
    .... .... 0... = Push: Not set
    .... .... .0.. = Reset: Not set
    .... .... ..0. = Syn: Not set
    .... .... ...1 = Fin: Set
    [TCP Flags: ·······A···F]
```

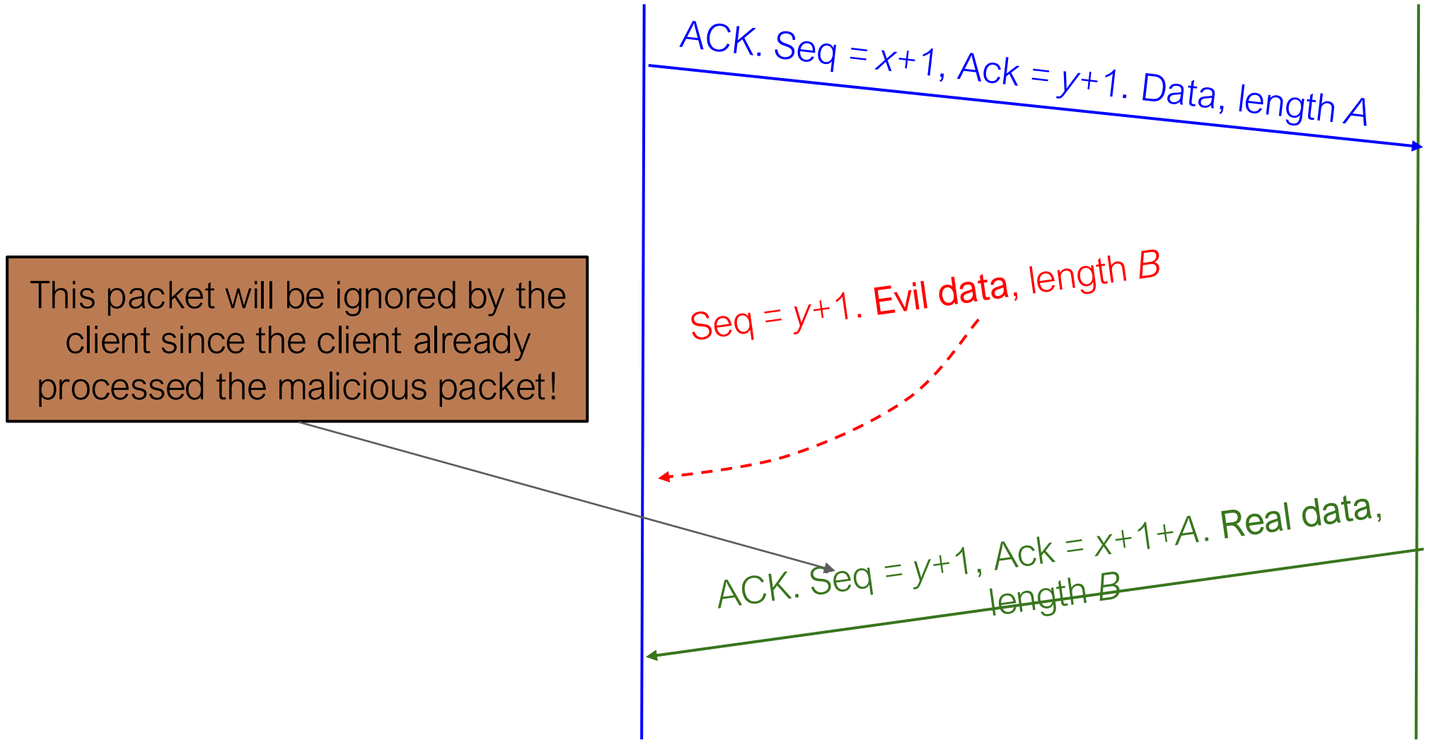
- In Wireshark's packet list, SYN and FIN segments are grey and RST segments dark red; the figures that follow use the same colours.

TCP Attacks



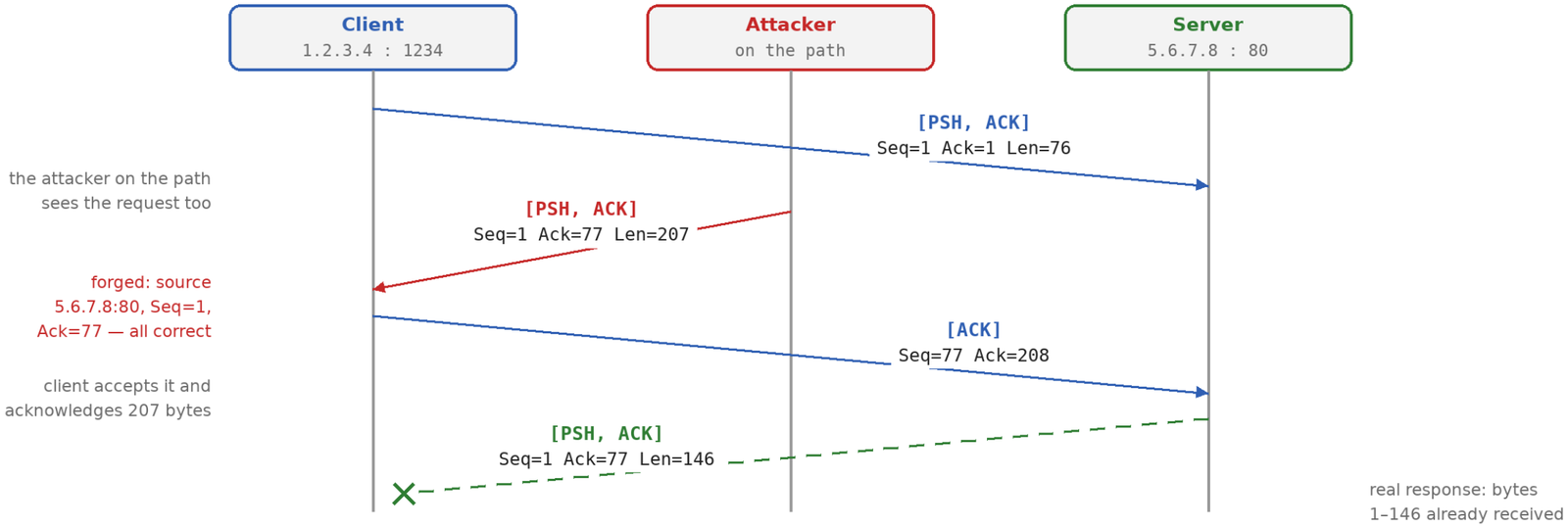
- **TCP hijacking**: Tampering with an existing session to modify or inject data into a connection
  - **Data injection**: **Spoofing packets to inject malicious data into a connection**
    - **Need to know: The sender's sequence number**
    - Easy for MITM and on-path attackers, but off-path attackers must guess 32-bit sequence number (called **blind injection/hijacking**, considered difficult)
    - For on-path attackers, this becomes a race condition since they must beat the sender
  - **RST injection**: Spoofing a RST packet to forcibly terminate a connection
    - Same requirements as packet injection, so easy for on-path and MITM attackers, but hard for off-path attackers
    - Often used in censorship scenarios to block access to sites

TCP Data Injection



On-path attack 1: injecting data into a TCP connection (tcp-injection.pcap)

The attacker sees the client's request, so she knows the ports, Seq and Ack. She sends her own response first.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	1.2.3.4	5.6.7.8	TCP	66	1234 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM WS=128
2	0.012303	5.6.7.8	1.2.3.4	TCP	66	80 → 1234 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM WS=128
3	0.012361	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [ACK] Seq=1 Ack=1 Win=502 Len=0
4	0.012448	1.2.3.4	5.6.7.8	HTTP	130	GET /grades HTTP/1.1
5	0.013102	5.6.7.8	1.2.3.4	HTTP	261	HTTP/1.1 302 Found (text/html) ← forged by the attacker
6	0.013150	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [ACK] Seq=77 Ack=208 Win=502 Len=0
7	0.025902	5.6.7.8	1.2.3.4	TCP	200	[TCP Spurious Retransmission] 80 → 1234 [PSH, ACK] Seq=1 Ack=77 Win=509 Len=146
8	0.025931	1.2.3.4	5.6.7.8	TCP	54	[TCP Dup ACK 6#1] 1234 → 80 [ACK] Seq=77 Ack=208 Win=502 Len=0

	Frame 5 (forged, arrives first)	Frame 7 (real, arrives later)
Source / destination	5.6.7.8:80 → 1.2.3.4:1234	5.6.7.8:80 → 1.2.3.4:1234
Sequence number	**1904226518 (relative 1)**	**1904226518 (relative 1)**
Acknowledgement number	relative 77	relative 77
Data	**HTTP/1.1 302 Found → evil.example (207 B)**	**HTTP/1.1 200 OK, the grades (146 B)**
IP Time to Live	**64**	**52**

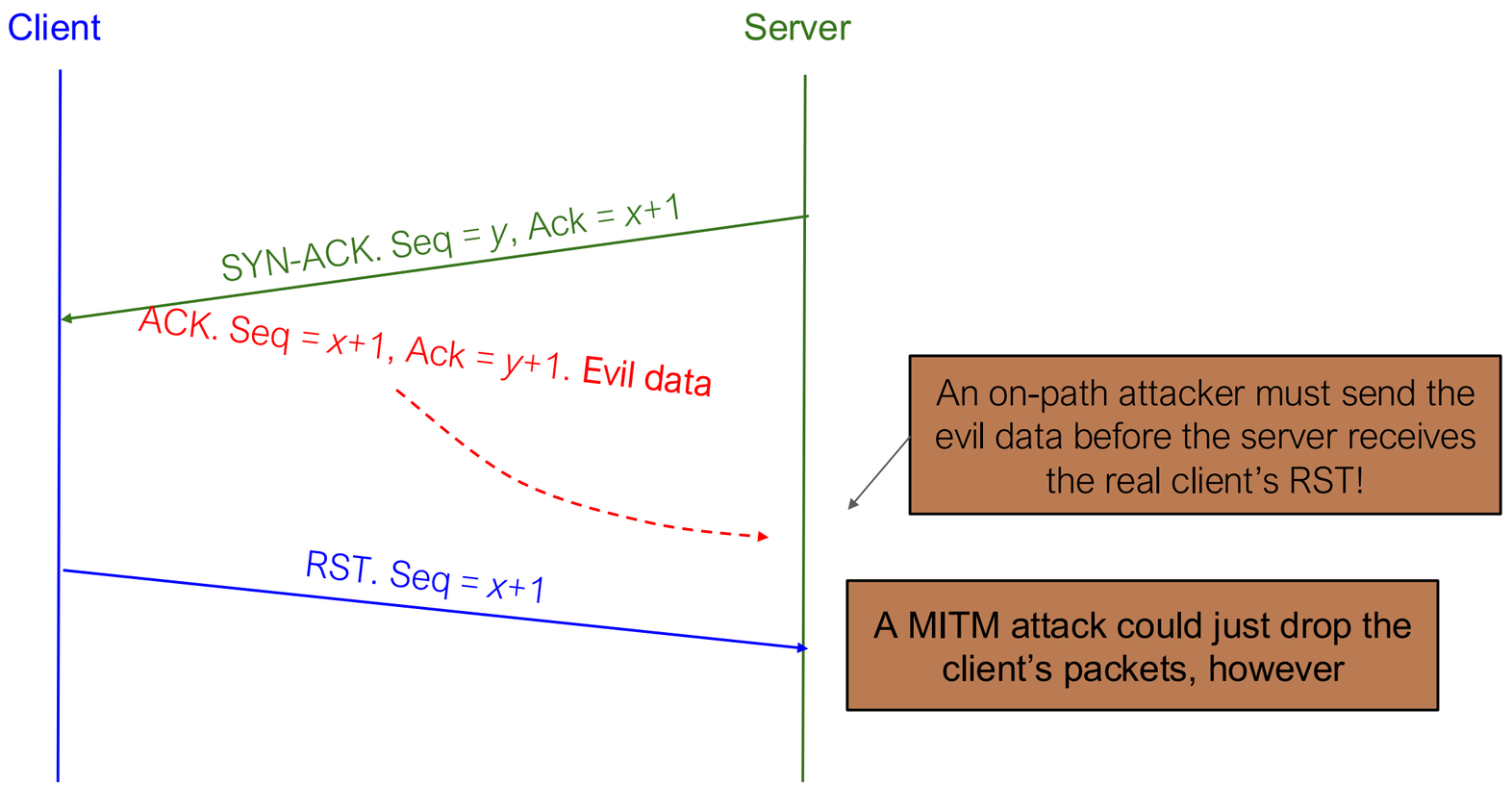
- TCP has no authentication: a segment with the right addresses, ports and sequence number is accepted, whoever sent it.
- The first one to arrive wins. The real data then looks like a retransmission of bytes already received, and is dropped.
- The forged response is longer than the real one, so no byte of the real response is new to the client.
- Only hints remain: a different TTL (64 vs 52), and the server receives Ack=208 for bytes it never sent.
- The fix is cryptography above TCP: with TLS the forged bytes fail the MAC check.

TCP Attacks



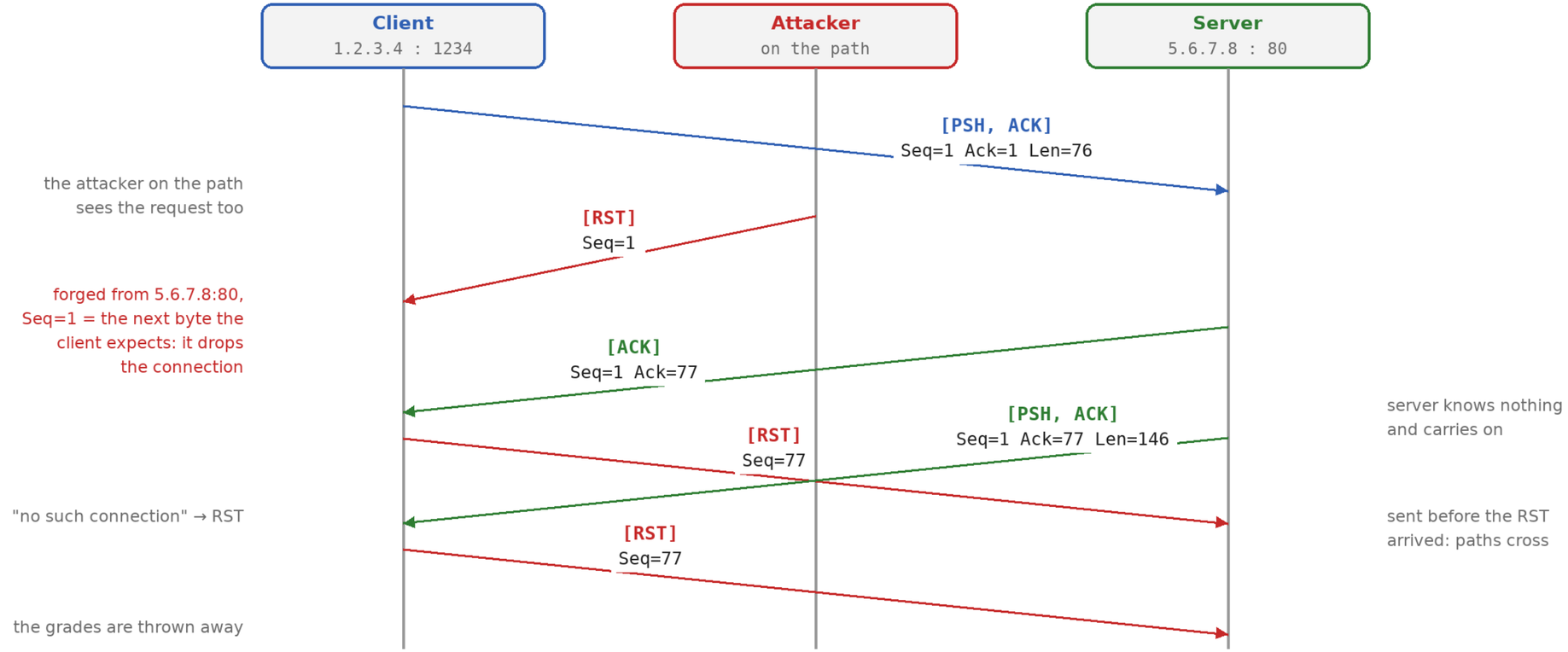
- **TCP spoofing**: Spoofing a TCP connection to appear to come from another source IP address
  - Recall: IP packets can often be spoofed if the **sender AS doesn't block spoofed packets**
  - **Quick question: how AS blocks spoofed packets?**
  - The attacker will need to know the sequence number in the server's response SYN-ACK packet
  - Easy for MITM and on-path attackers, but off-path attackers must guess 32-bit sequence number (called **blind spoofing**, also considered difficult)
  - For on-path attackers, this involves a race condition, since the real client will send a RST upon receiving the server's SYN-ACK!

TCP Attacks


Client
Server
SYN-ACK. Seq = y, Ack = x+1
ACK. Seq = x+1, Ack = y+1. Evil data
RST. Seq = x+1
An on-path attacker must send the evil data before the server receives the real client's RST!
A MITM attack could just drop the client's packets, however

On-path attack 2: RST injection — tearing down someone else's connection

Same position as before, but the attacker sends one RST. (tcp-rst-injection.pcap)



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	1.2.3.4	5.6.7.8	TCP	66	1234 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM WS=128
2	0.012303	5.6.7.8	1.2.3.4	TCP	66	80 → 1234 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM WS=128
3	0.012361	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [ACK] Seq=1 Ack=1 Win=502 Len=0
4	0.012448	1.2.3.4	5.6.7.8	HTTP	130	GET /grades HTTP/1.1
5	0.013020	5.6.7.8	1.2.3.4	TCP	54	80 → 1234 [RST] Seq=1 Win=0 Len=0 ← forged by the attacker
6	0.024861	5.6.7.8	1.2.3.4	TCP	54	80 → 1234 [ACK] Seq=1 Ack=77 Win=509 Len=0
7	0.024880	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [RST] Seq=77 Win=0 Len=0
8	0.025902	5.6.7.8	1.2.3.4	HTTP	200	HTTP/1.1 200 OK (text/html)
9	0.025920	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [RST] Seq=77 Win=0 Len=0

- The client checks only that the RST's sequence number fits; it was right, so the client drops the connection at once.
- Afterwards the client answers every segment of the old connection with RST: from the server's view the client hung up.
- This is cheaper than data injection — one small packet, no need to win a race — and has been used to censor connections.
- TLS does not stop it: RST is a TCP flag, below the encryption. TLS can only notice that the connection ended early.

Off-path attack: spoofing a segment without seeing the connection

The attacker can send packets with source 5.6.7.8:80, but never sees the client's packets. Black rows: her forged segments.

Forged Seq (raw)	Client expects	Difference	Result
2717302955	1904226518	+813076437	**dropped**
1045993270	1904226518	-858233248	**dropped**
3960114812	1904226518	+2055888294	**dropped**
1904226518	1904226518	0	**accepted (hypothetical: the one right guess)**

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	1.2.3.4	5.6.7.8	TCP	66	1234 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM WS=128
2	0.012303	5.6.7.8	1.2.3.4	TCP	66	80 → 1234 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM WS=128
3	0.012361	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [ACK] Seq=1 Ack=1 Win=502 Len=0
4	0.012448	1.2.3.4	5.6.7.8	HTTP	130	GET /grades HTTP/1.1
5	0.024861	5.6.7.8	1.2.3.4	TCP	54	80 → 1234 [ACK] Seq=1 Ack=77 Win=509 Len=0
6	0.030000	5.6.7.8	1.2.3.4	TCP	261	[TCP Previous segment not captured] 80 → 1234 [PSH, ACK] Seq=813076438 Ack=77 Win=509 Len=207
7	0.030020	1.2.3.4	5.6.7.8	TCP	54	1234 → 80 [ACK] Seq=77 Ack=1 Win=502 Len=0 → to the real server
8	0.031000	5.6.7.8	1.2.3.4	TCP	261	[TCP Retransmission] 80 → 1234 [PSH, ACK] Seq=3436734049 Ack=77 Win=509 Len=207
9	0.031020	1.2.3.4	5.6.7.8	TCP	54	[TCP Dup ACK 7#1] 1234 → 80 [ACK] Seq=77 Ack=1 Win=502 Len=0 → to the real server
10	0.032000	5.6.7.8	1.2.3.4	TCP	261	[TCP Previous segment not captured] 80 → 1234 [PSH, ACK] Seq=2055888295 Ack=77 Win=509 Len=207
11	0.032020	1.2.3.4	5.6.7.8	TCP	54	[TCP Dup ACK 7#2] 1234 → 80 [ACK] Seq=77 Ack=1 Win=502 Len=0 → to the real server

- Each wrong guess is dropped, and the client answers with a duplicate ACK — but that ACK goes to the real server, so the off-path attacker never learns how close she was.
- Sequence numbers are 32 bits and the initial one is random: a blind guess lands exactly on the next expected byte with probability 1 / 2^32. (Real stacks also accept data anywhere in the receive window, here 64 KB, which makes a guess about 64K times more likely to land — still tiny for a single packet.)
- An on-path attacker reads the right number off the wire; an off-path attacker can only guess. That is the whole difference.
- The ports must be right too. The server port (80) is public; the client's port is another thing to guess.
- UDP has no sequence numbers: an off-path attacker guesses only the ports and whatever the application adds (for DNS, a 16-bit query ID — see the DNS lecture).

TCP Attacks



- Quick question: how AS blocks spoofed packets?
  - AS can check whether the source IP complies with the network structure
  - For instance, the AS knows that the subnet is 203.0.113.0/24
  - If received a packet inside the AS stating the source IP is 8.8.8.8 -> drop

TCP Attacks



- **TCP provides no confidentiality or integrity**
  - Instead, we rely on higher layers (like TLS, more on this next time) to prevent those kind of attacks
- **Defense against off-path attackers rely on choosing random sequence numbers**
  - Bad randomness can lead to trivial off-path attacks: TCP sequence numbers used to be based on the system clock!

Real Example: On-path RST injection



- Anyone on the path can kill your connection: no auth on plain TCP.
- Detection: the forged RST's TTL and other fingerprints don't match the real endpoint.
- Real cases: Comcast 2007 (FCC case), Verizon 2011

Real Example: On-path RST injection



- **Comcast vs. BitTorrent (USA, 2007–2008)**
  - Oct 2007: the Associated Press shows that Comcast interferes with BitTorrent uploads; EFF tests confirm **Comcast forged TCP RST packets impersonating each peer**, so both ends believed the other had hung up.
- **Verizon Wireless (USA, 2011)**
  - July 2011: a DSLreports user captures his dying uploads with Wireshark: the RSTs come from **Verizon-owned IPs (170.49.96.x), not from his peer**.
  - EFF, "Verizon Wireless Falsely Reports BitTorrent Uploads to Throttle" (14 Jul 2011): BitTorrent "falsely believes the peer rejected the connection" and retries endlessly.

Real Example: On-path data injection



- **ISP content injection (USA, 2017)**
  - Comcast injected its own JavaScript notices into **unencrypted HTTP** pages returned to customers (widely reported Sept 2017); the injected content rides on the same "modify or add to a plain-TCP stream" power as figure 06.
- On plain TCP the response is whatever arrives first — the attacker can win the race.
- Content injection = same trick, without breaking the connection: users notice less.
- TLS ends all of this: the attacker can reset, but not forge, a session