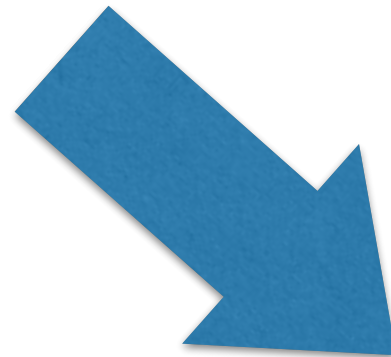


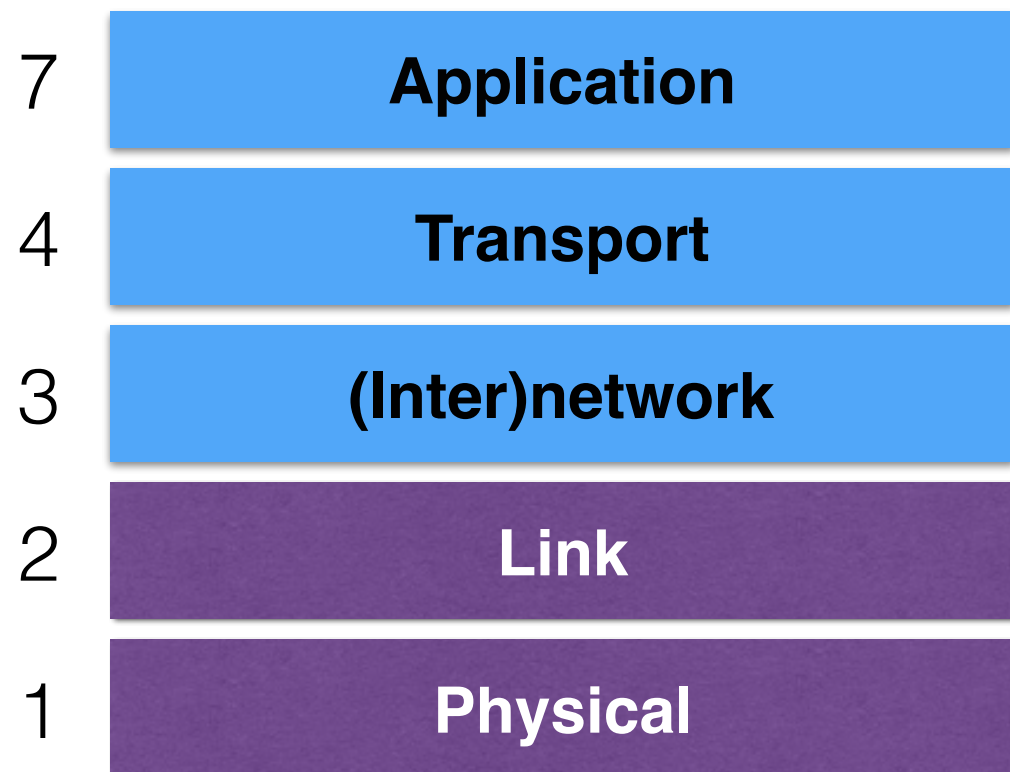
This time

Digging into
**Networking
Protocols**



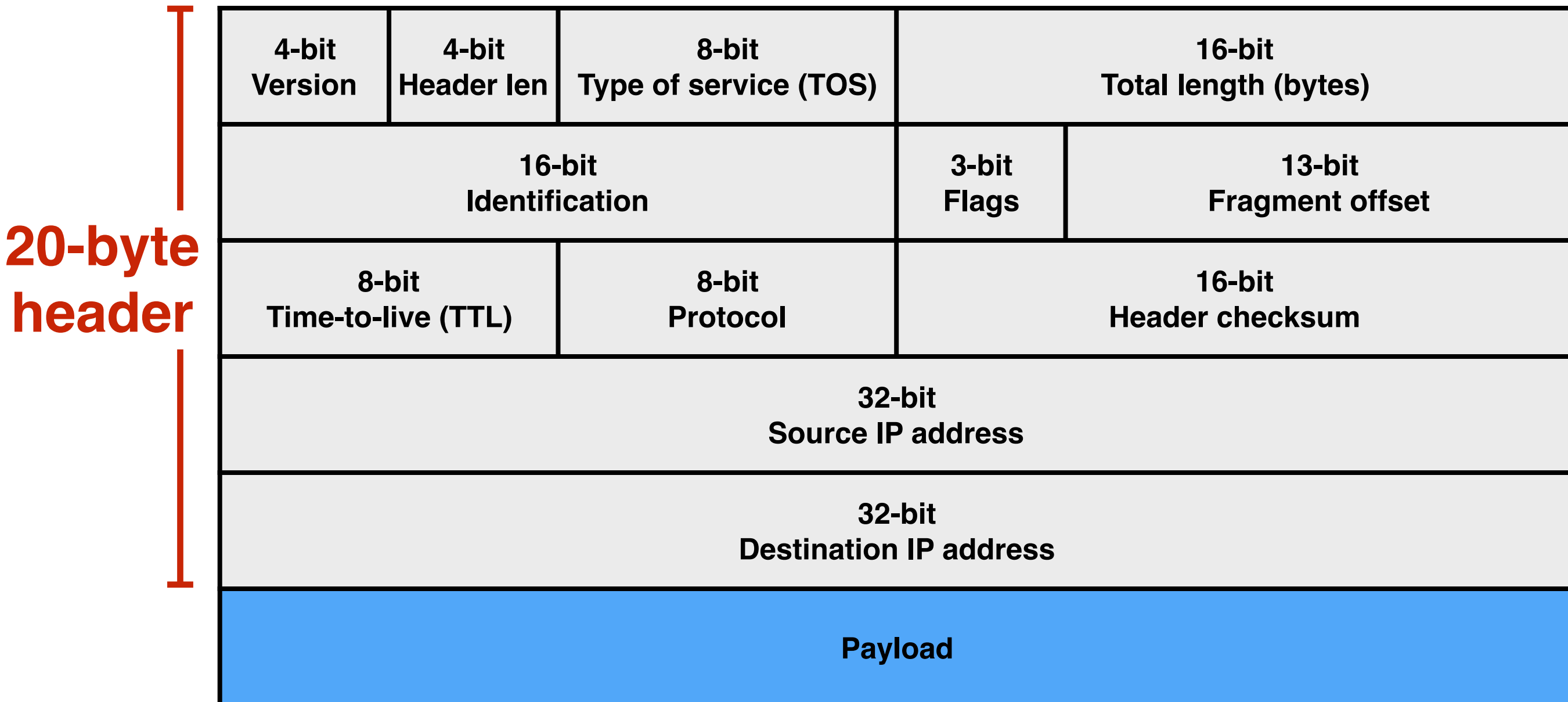
With a particular focus on
**TCP/IP details,
attacks,
and defenses**

Layer 3: (Inter)network layer



- Bridges multiple “subnets” to provide *end-to-end* **internet** connectivity between **nodes**
- Provides **global** addressing (IP addresses)
- Only provides **best-effort** delivery of data (i.e., no retransmissions, etc.)
- Works across different link technologies

IP packet “header”



IP Packet Header Fields (1)

- **Version number** (4 bits)
 - Indicates the version of the IP protocol
 - Necessary for knowing what fields follow
 - “4” (for IPv4) or “6” (for IPv6)
- **Header length** (4 bits)
 - How many 32-bit words (rows) in the header
 - Typically 5
 - Can provide IP options, too
- **Type-of-service** (8 bits)
 - Allow packets to be treated differently based on different needs
 - Low delay for audio, high bandwidth for bulk transfer, etc.

IP Packet Header Fields (2)

- Two IP addresses
 - Source (32 bits)
 - Destination (32 bits)
- Destination address
 - *Unique* identifier/locator for the receiving host
 - Allows each node (end-host and router) to make forwarding decisions
- Source address
 - Unique identifier/locator for the sending host
 - Recipient can decide whether to accept the packet
 - Allows destination to *reply* to the source

IP: “Best effort” packet delivery

- Routers inspect destination address, determine “next hop” in the forwarding table
- Best effort = “I’ll give it a try”
 - Packets may be lost
 - Packets may be corrupted
 - Packets may be delivered out of order

Fixing these is the job of the transport layer!

Attacks on IP

4-bit Version	4-bit Header len	8-bit Type of service (TOS)	16-bit Total length (bytes)	
16-bit Identification			3-bit Flags	13-bit Fragment offset
8-bit Time-to-live (TTL)		8-bit Protocol	16-bit Header checksum	
32-bit Source IP address				
32-bit Destination IP address				
Payload				

Attacks on IP

4-bit Version	4-bit Header len	8-bit Type of service (TOS)	16-bit Total length (bytes)	
16-bit Identification			3-bit Flags	13-bit Fragment offset
8-bit Time-to-live (TTL)		8-bit Protocol	16-bit Header checksum	
32-bit Source IP address				
32-bit Destination IP address				
Payload				

Source-spoof

There is nothing in IP that enforces that your source IP address is really “yours”

Attacks on IP

4-bit Version	4-bit Header len	8-bit Type of service (TOS)	16-bit Total length (bytes)	
16-bit Identification			3-bit Flags	13-bit Fragment offset
8-bit Time-to-live (TTL)		8-bit Protocol	16-bit Header checksum	
32-bit Source IP address				
32-bit Destination IP address				
Payload				

Source-spoof

There is nothing in IP that enforces that your source IP address is really “yours”

Eavesdrop / Tamper

IP provides no protection of the *payload* or *header*

Source-spoofing

- Why source-spoof?
 - Consider spam: send many emails from one computer
 - Easy defense: block many emails from a given (source) IP address
 - Easy countermeasure: spoof the source IP address
 - Counter-countermeasure?
- How do you know if a packet you receive has a spoofed source?

Salient network features

- Recall: The Internet operates via *destination-based routing*
- attacker: pkt (spoofed source) -> destination
destination: pkt -> spoofed source
- In other words, the response goes to the spoofed source, *not* the attacker

Defending against source-spoofing

- How do you know if a packet you receive has a spoofed source?
 - Send a challenge packet to the (possibly spoofed) source (e.g., a difficult to guess, random nonce)
 - If the recipient can answer the challenge, then likely that the source was not spoofed
- So do you have to do this with every packet??
 - Every packet should have something that's difficult to guess
 - Recall the query ID in the DNS queries! Easy to predict => Kaminsky attack

Source spoofing

- Why source-spoof?
 - Consider DoS attacks: generate as much traffic as possible to congest the victim's network
 - Easy defense: block all traffic from a given source near the edge of your network
 - Easy countermeasure: spoof the source address
- Challenges won't help here; the damage has been done by the time the packets reach the core of our network
- Ideally, detect such spoofing *near the source*

Egress filtering

- The point (router/switch) at which traffic *enters* your network is the *ingress point*
- The point (router/switch) at which traffic *leaves* your network is the *egress point*
- You don't know who owns all IP addresses in the world, but you *do* know who in *your own network* gets what IP addresses
 - If you see a packet with a source IP address that doesn't belong to your network trying to cross your egress point, then *drop it*

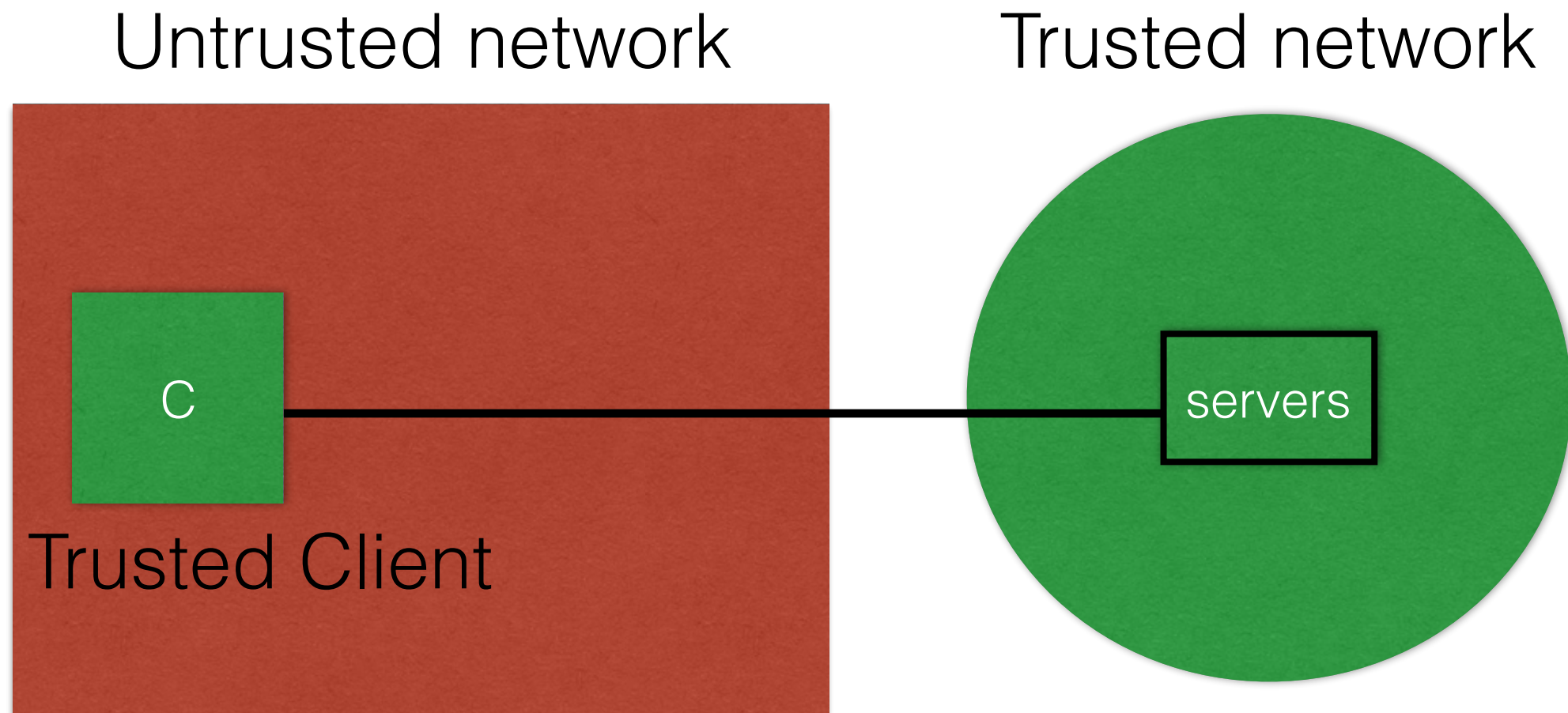
Egress filtering is not widely deployed

Eavesdropping / Tampering

4-bit Version	4-bit Header len	8-bit Type of service (TOS)	16-bit Total length (bytes)	
16-bit Identification			3-bit Flags	13-bit Fragment offset
8-bit Time-to-live (TTL)		8-bit Protocol	16-bit Header checksum	
32-bit Source IP address				
32-bit Destination IP address				
Payload				

- No security built into IP
- => Deploy secure IP *over IP*

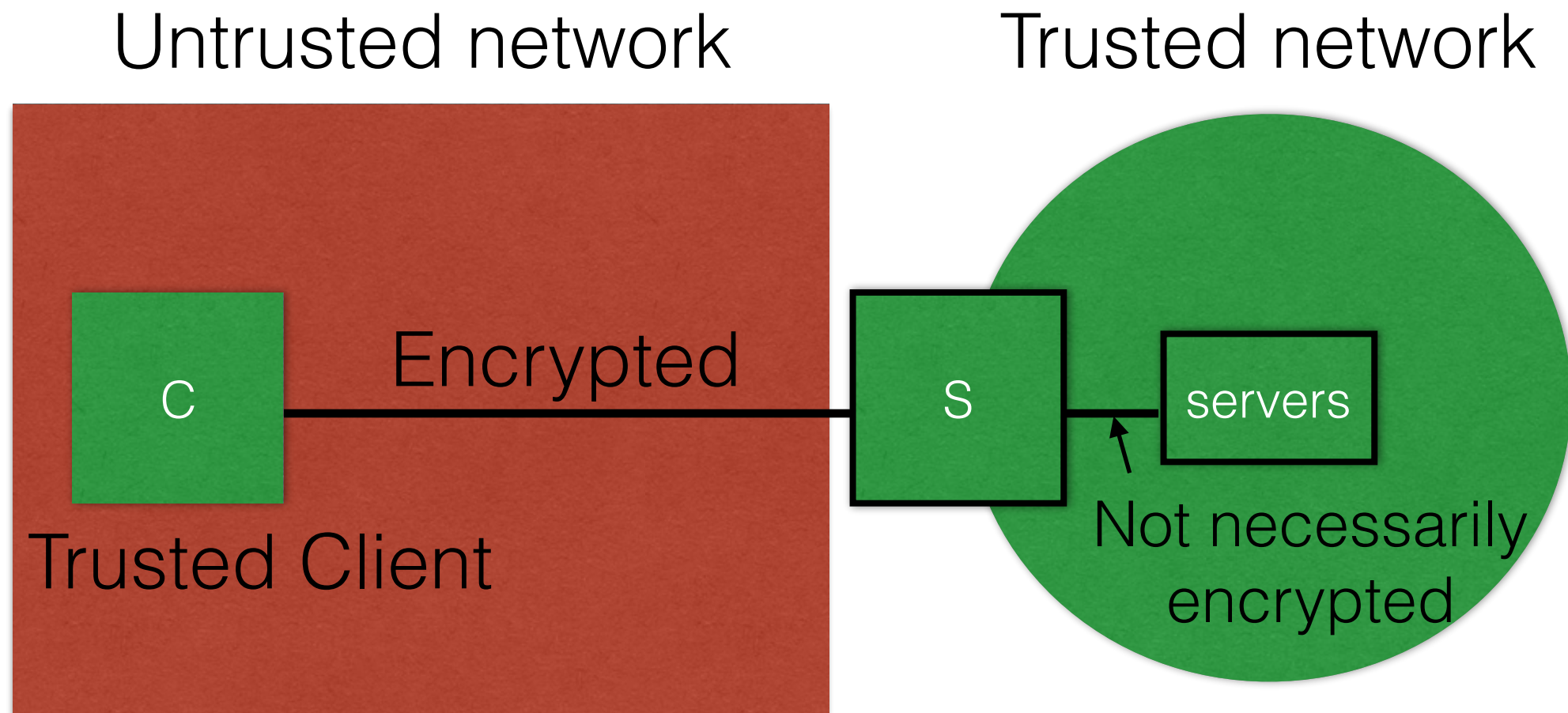
Virtual Private Networks (VPNs)



Goal: Allow the client to connect to the trusted network from within an untrusted network

Example: Connect to your company's network (for payroll, file access, etc.) while visiting a competitor's office

Virtual Private Networks (VPNs)



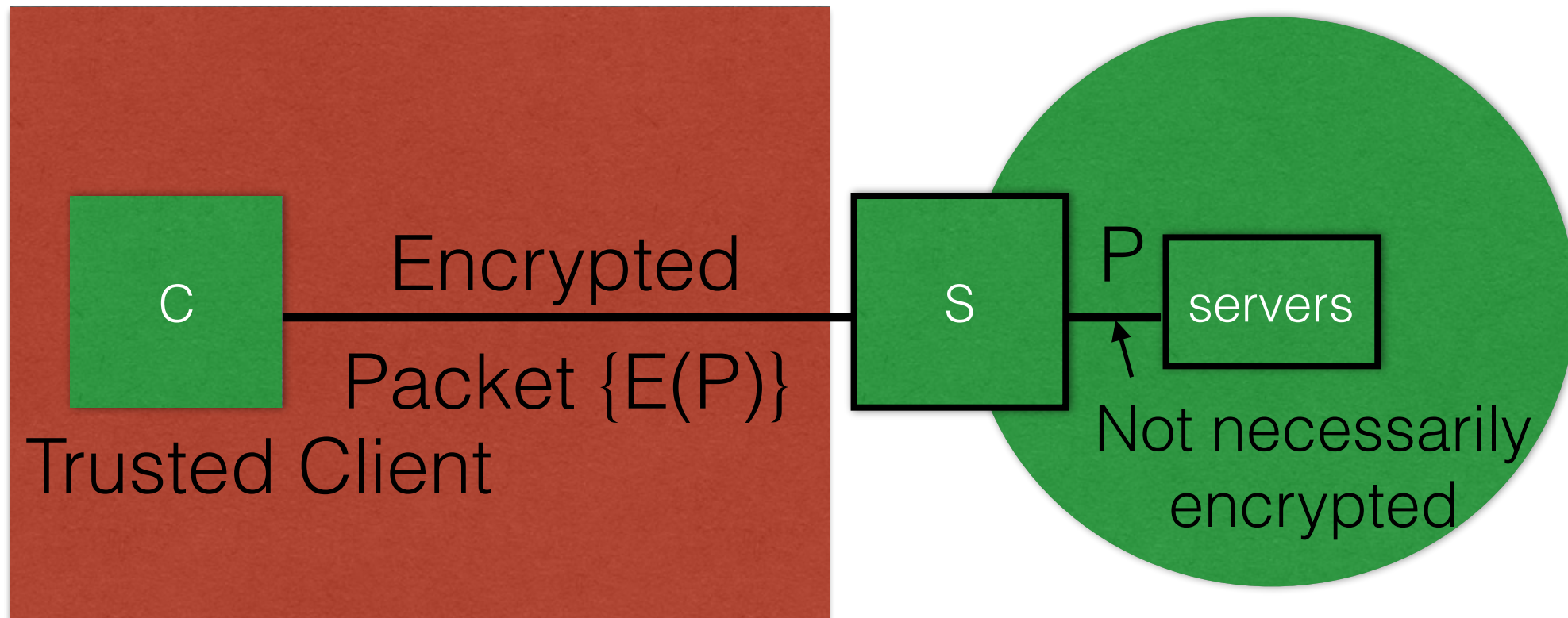
Idea: A VPN “client” and “server” together create end-to-end encryption/authentication

Predominate way of doing this: IPSec

IPSec

- Operates in a few different modes
 - Transport mode: Simply encrypt the payload but not the headers
 - Tunnel mode: Encrypt the payload *and* the headers
- But how do you encrypt the headers? How does routing work?
 - Encrypt the entire IP packet and make that the payload of another IP packet

Tunnel mode

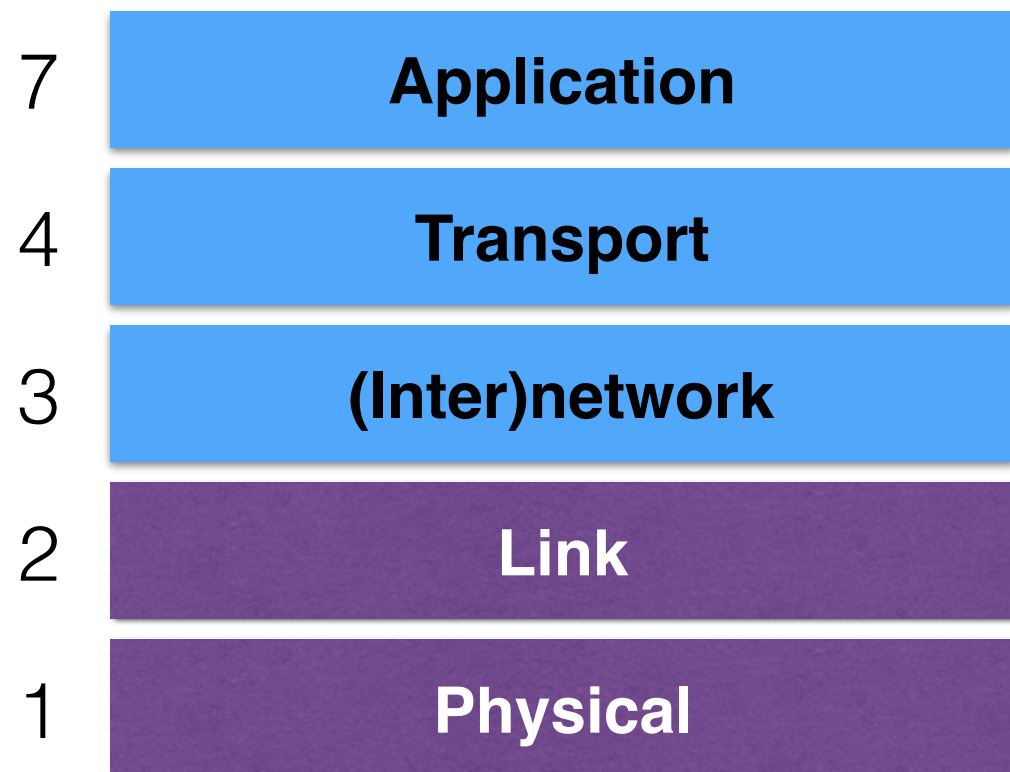


The VPN server decrypts and then sends the payload (itself a full IP packet) as if it had just received it from the network

From the client/servers' perspective:

Looks like the client is physically connected to the network!

Layer 4: Transport layer



- End-to-end communication between **processes**
- Different types of services provided:
 - UDP: unreliable *datagrams*
 - TCP: *reliable* byte stream
- “Reliable” = keeps track of what data were received properly and retransmits as necessary

TCP: reliability

- Given best-effort deliver, the goal is to ensure *reliability*
 - All packets are delivered to applications
 - ... in order
 - ... unmodified (with reasonably high probability)
- Must robustly detect and retransmit lost data

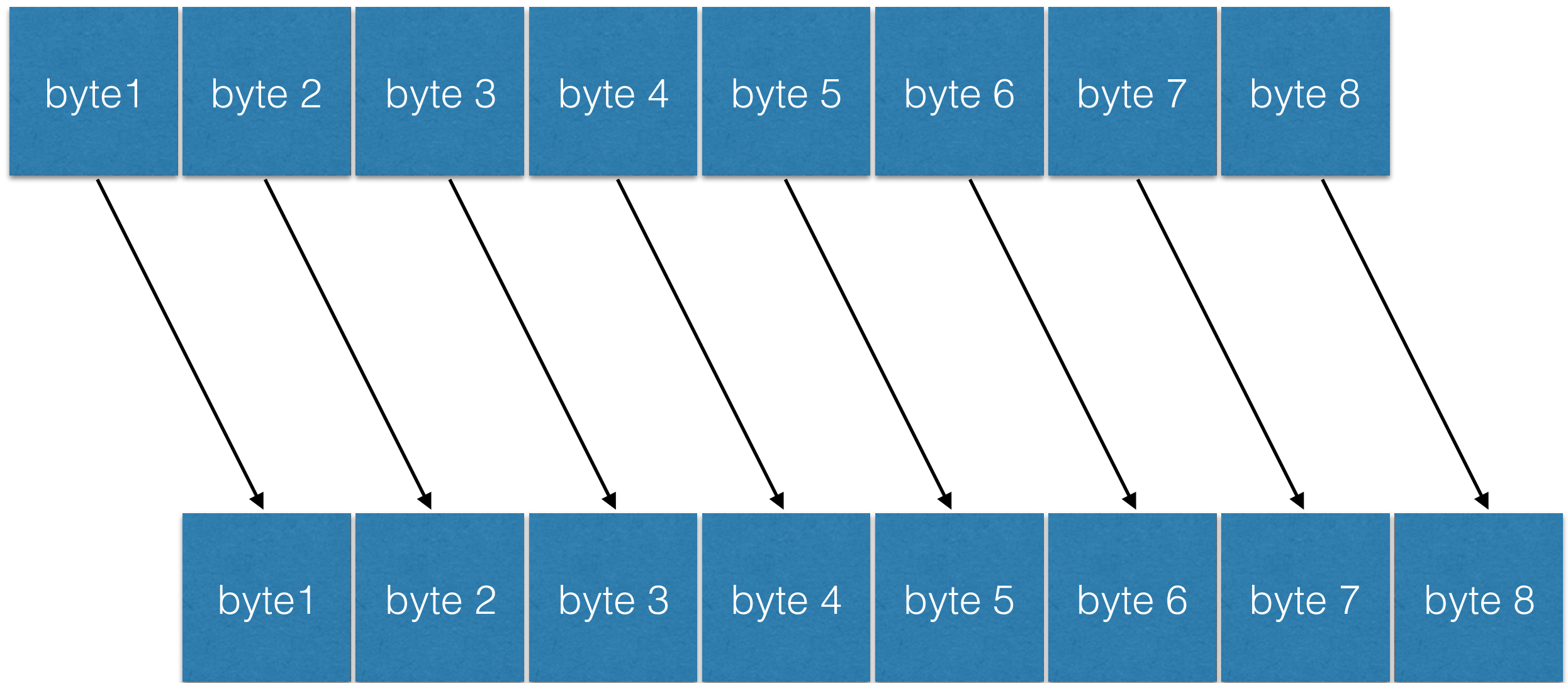
TCP's bytestream service

- Process A on host 1:
 - Send byte 0, byte 1, byte 2, byte 3, ...
- Process B on host 2:
 - Receive byte 0, byte 1, byte 2, byte 3, ...
- The applications do **not** see:
 - packet boundaries (looks like a stream of bytes)
 - lost or corrupted packets (they're all correct)
 - retransmissions (they all only appear once)

TCP bytestream service

Abstraction: Each *byte* reliably delivered in order

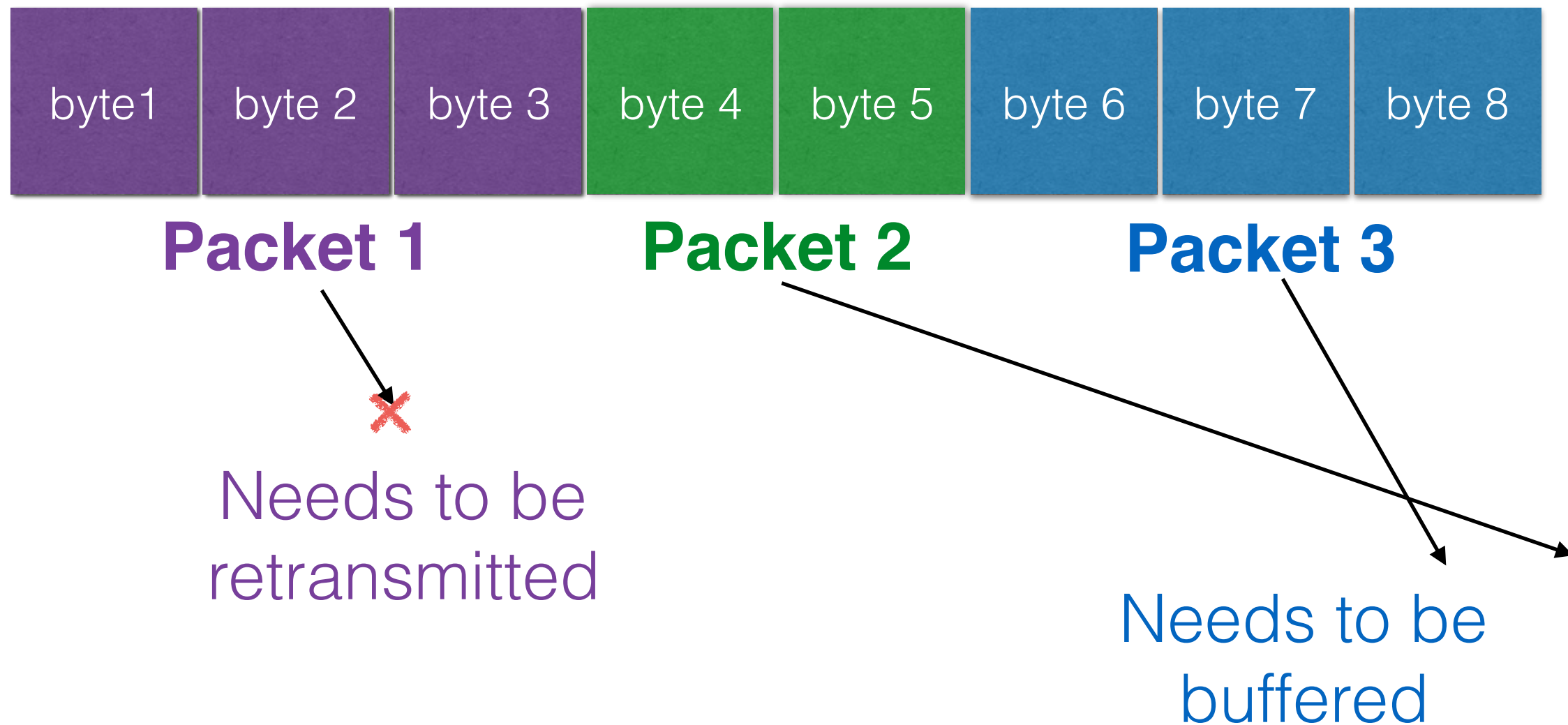
Process A on host H1



Process B on host H2

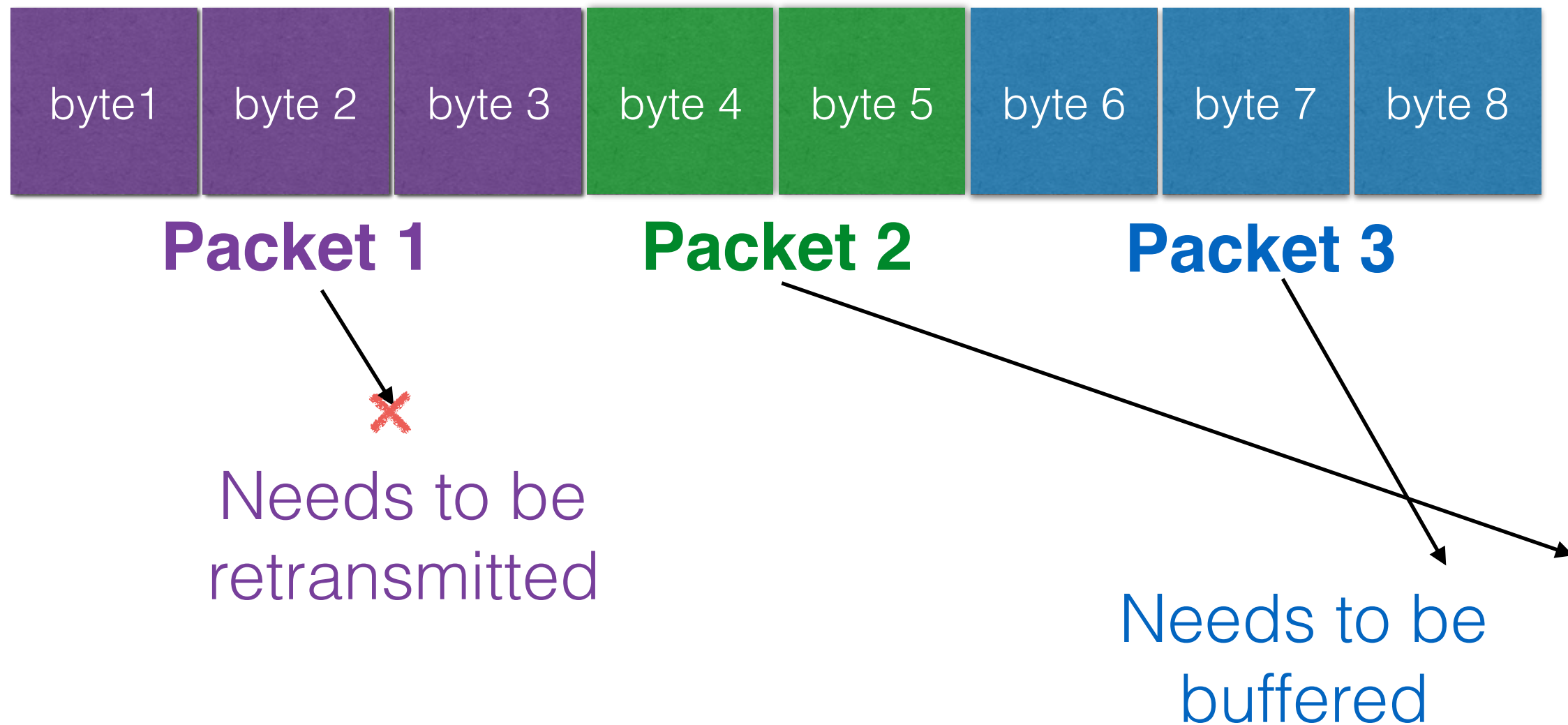
TCP bytestream service

**Reality: *Packets* sometimes retransmitted,
sometimes arrive out of order**



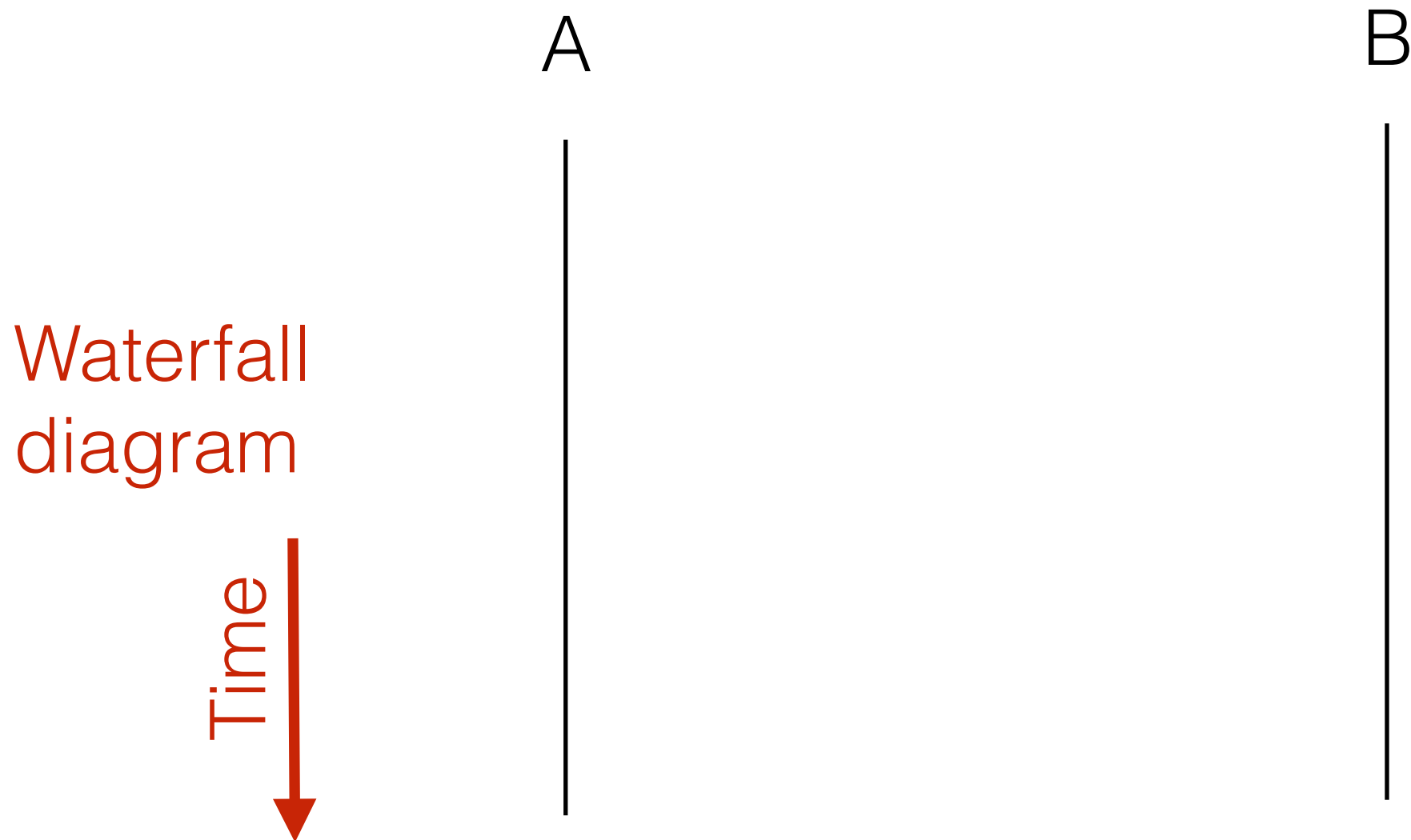
TCP bytestream service

**Reality: *Packets* sometimes retransmitted,
sometimes arrive out of order**

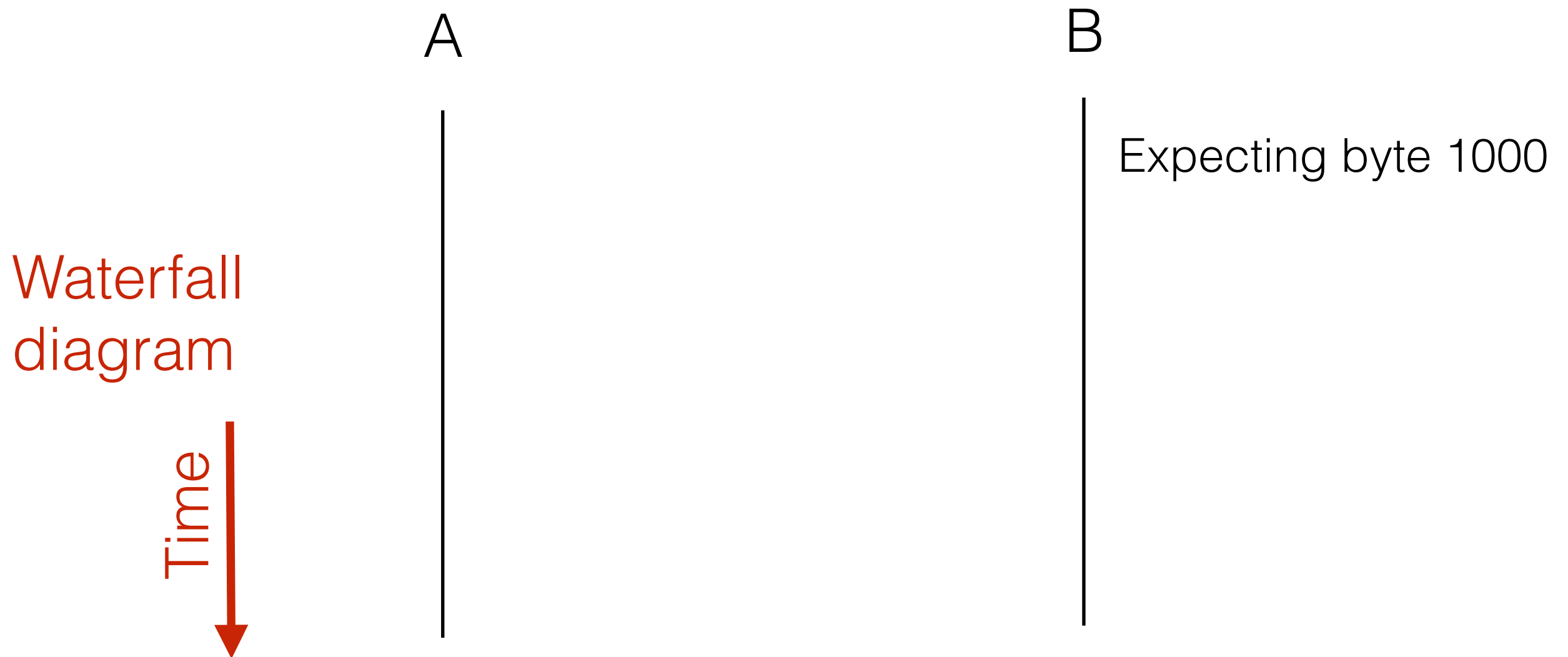


**TCP's first job: achieve the abstraction while
hiding the reality from the application**

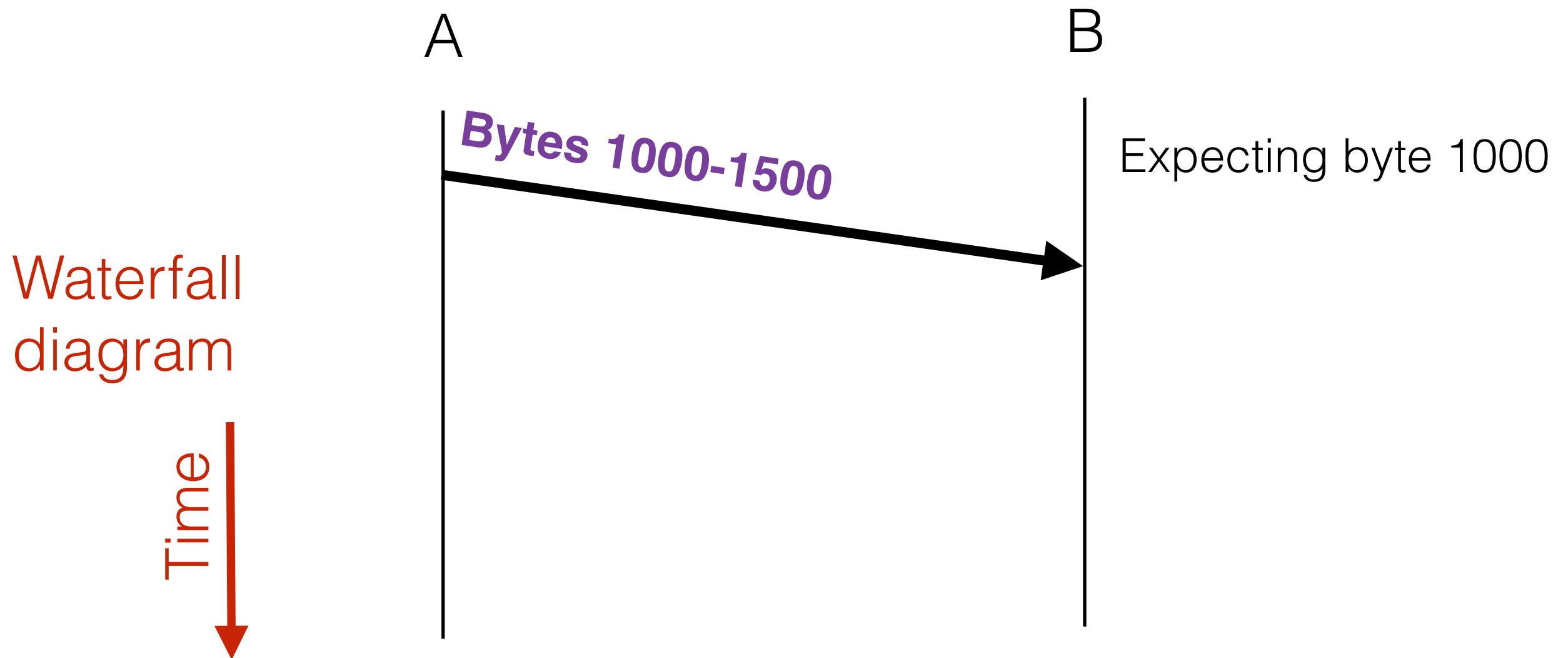
How does TCP achieve reliability?



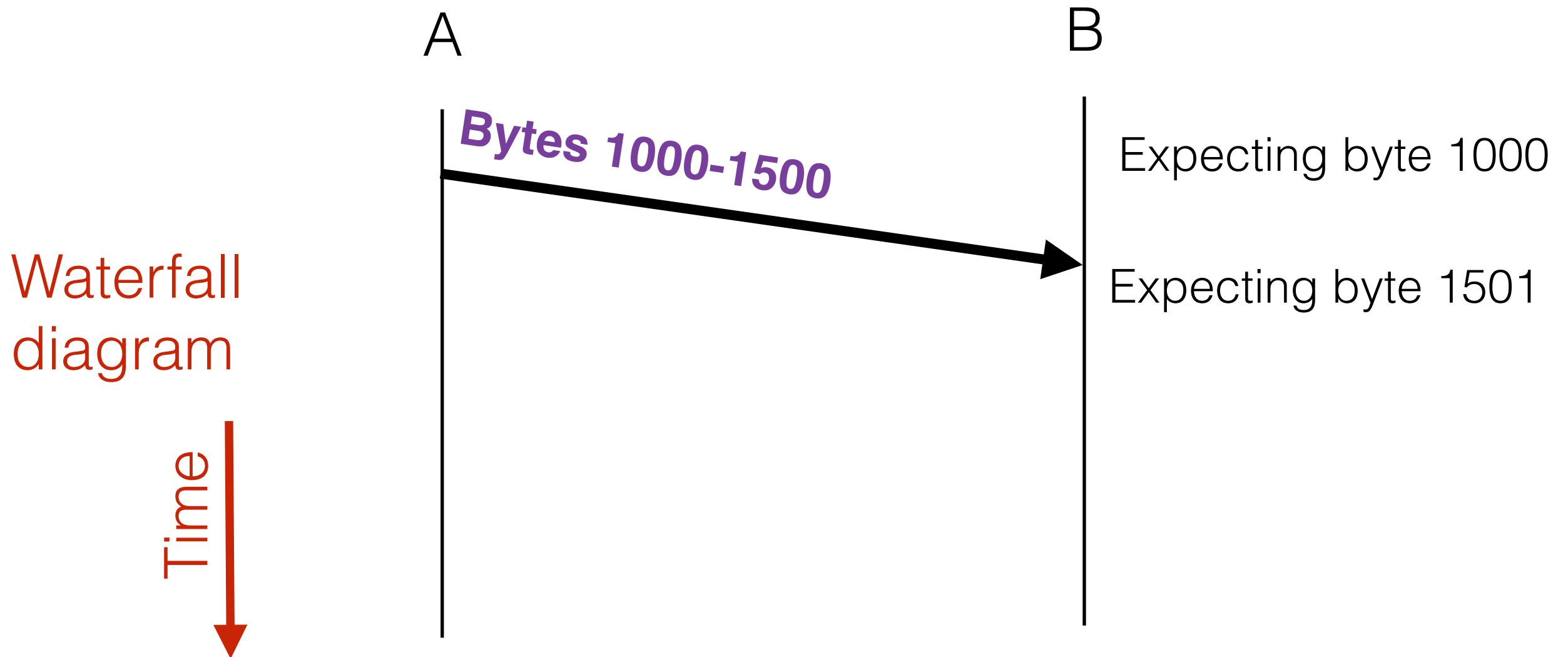
How does TCP achieve reliability?



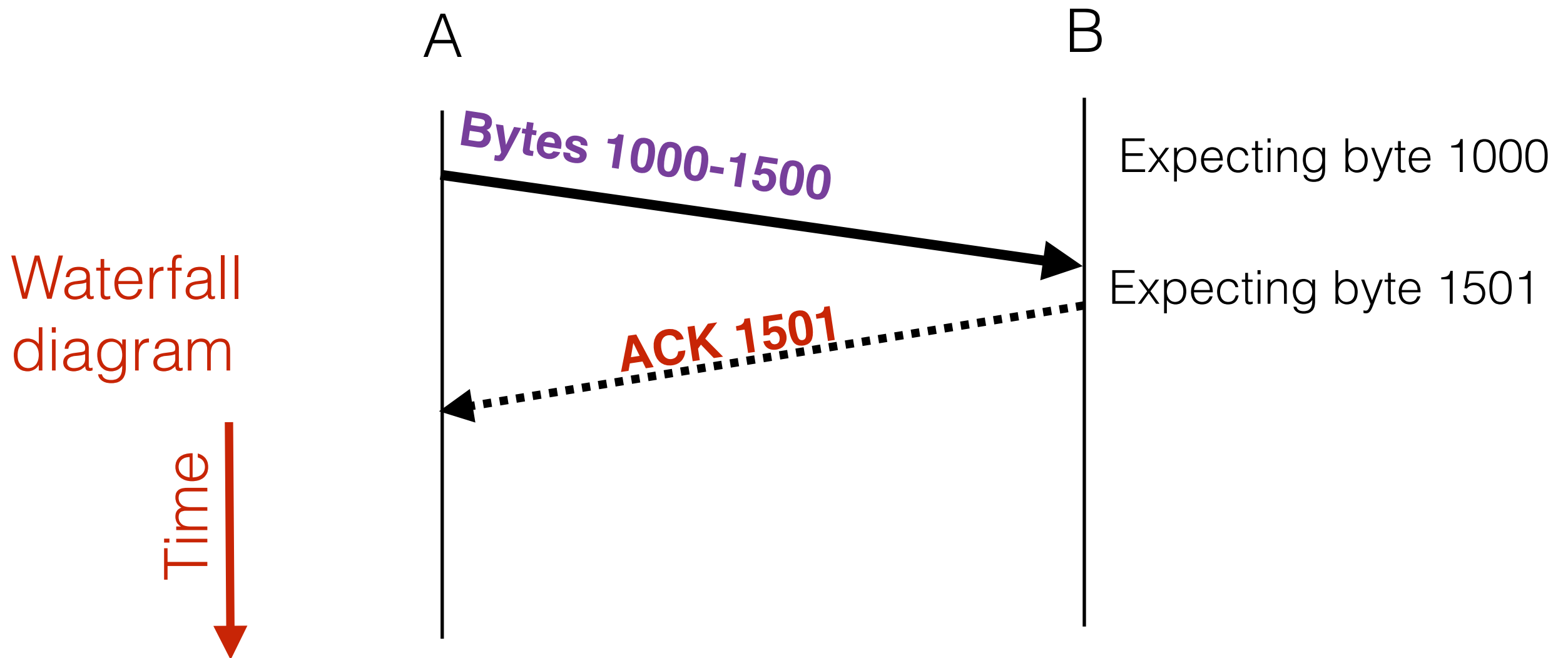
How does TCP achieve reliability?



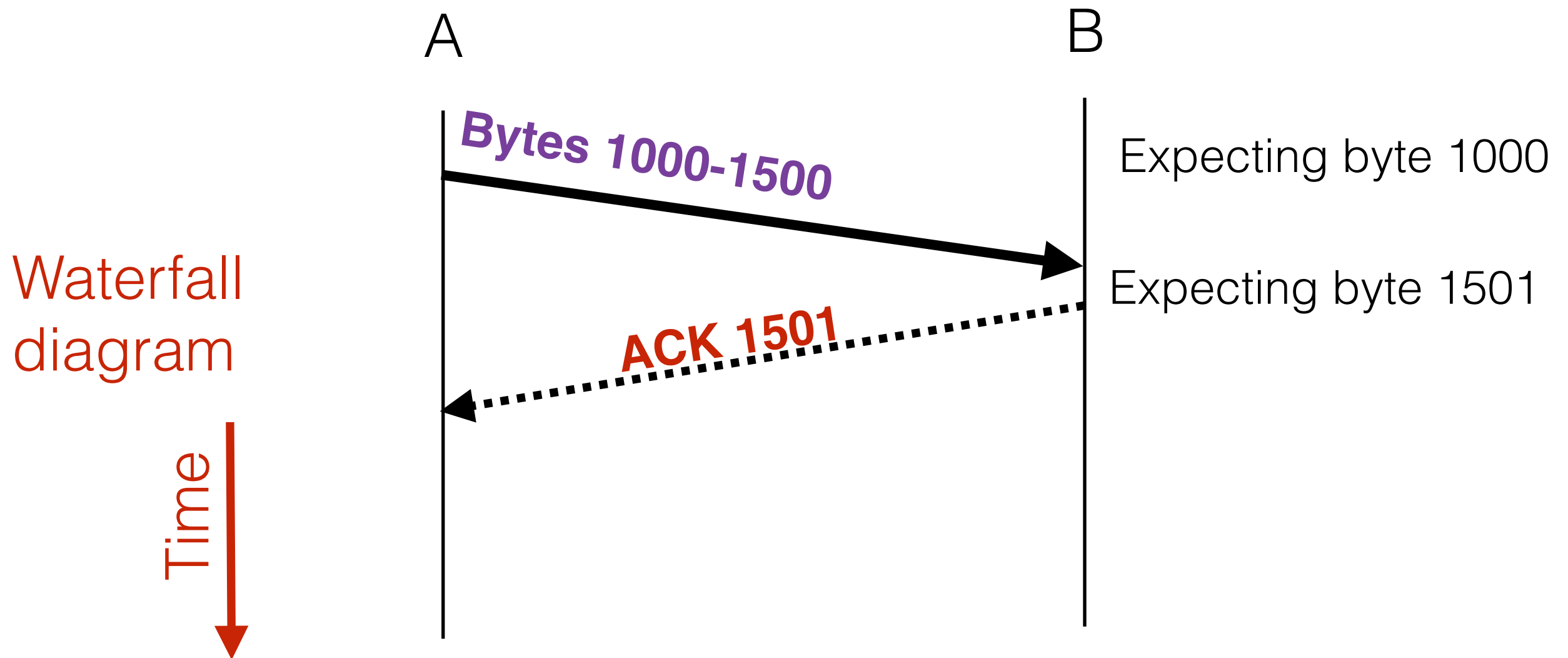
How does TCP achieve reliability?



How does TCP achieve reliability?

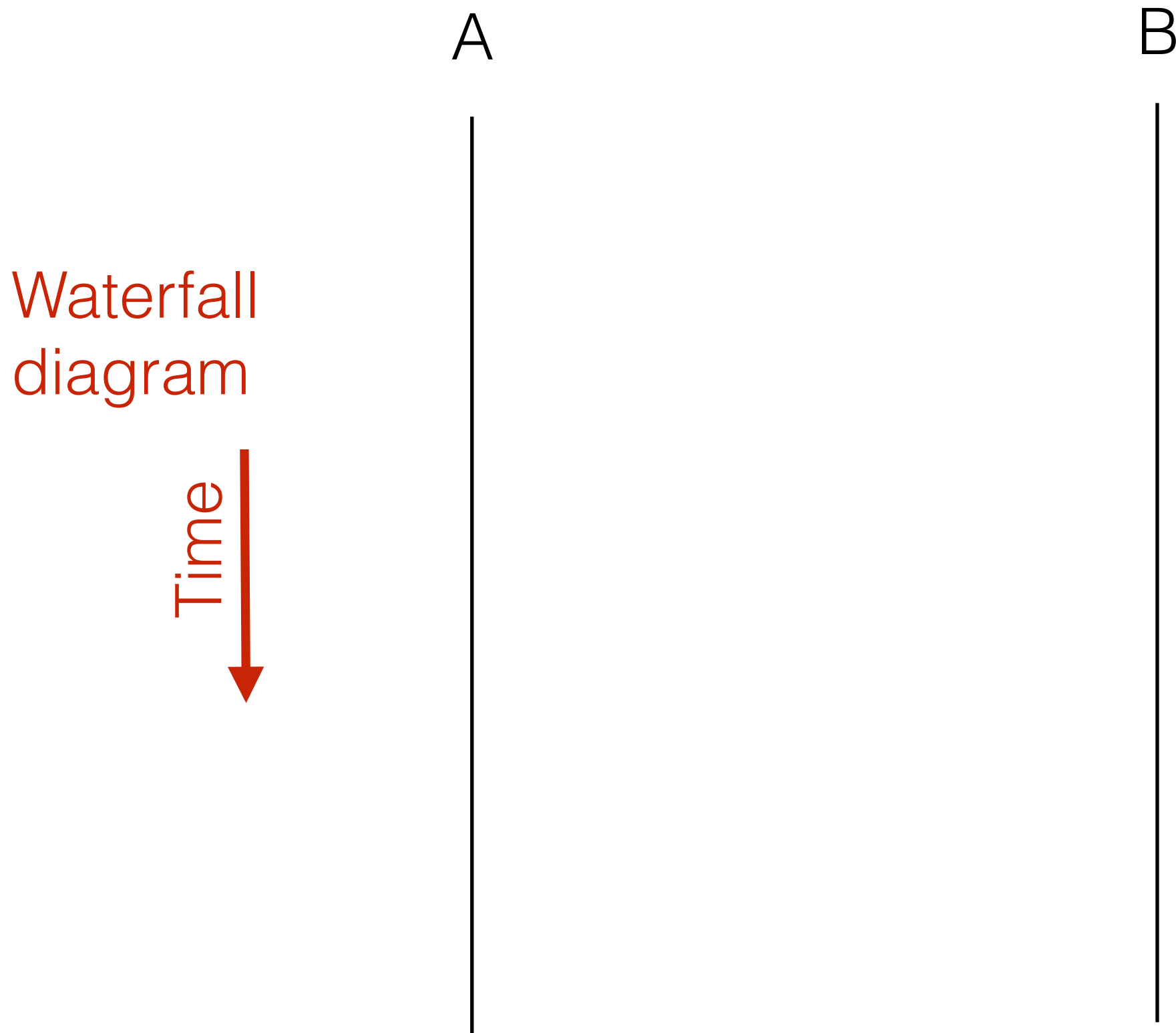


How does TCP achieve reliability?

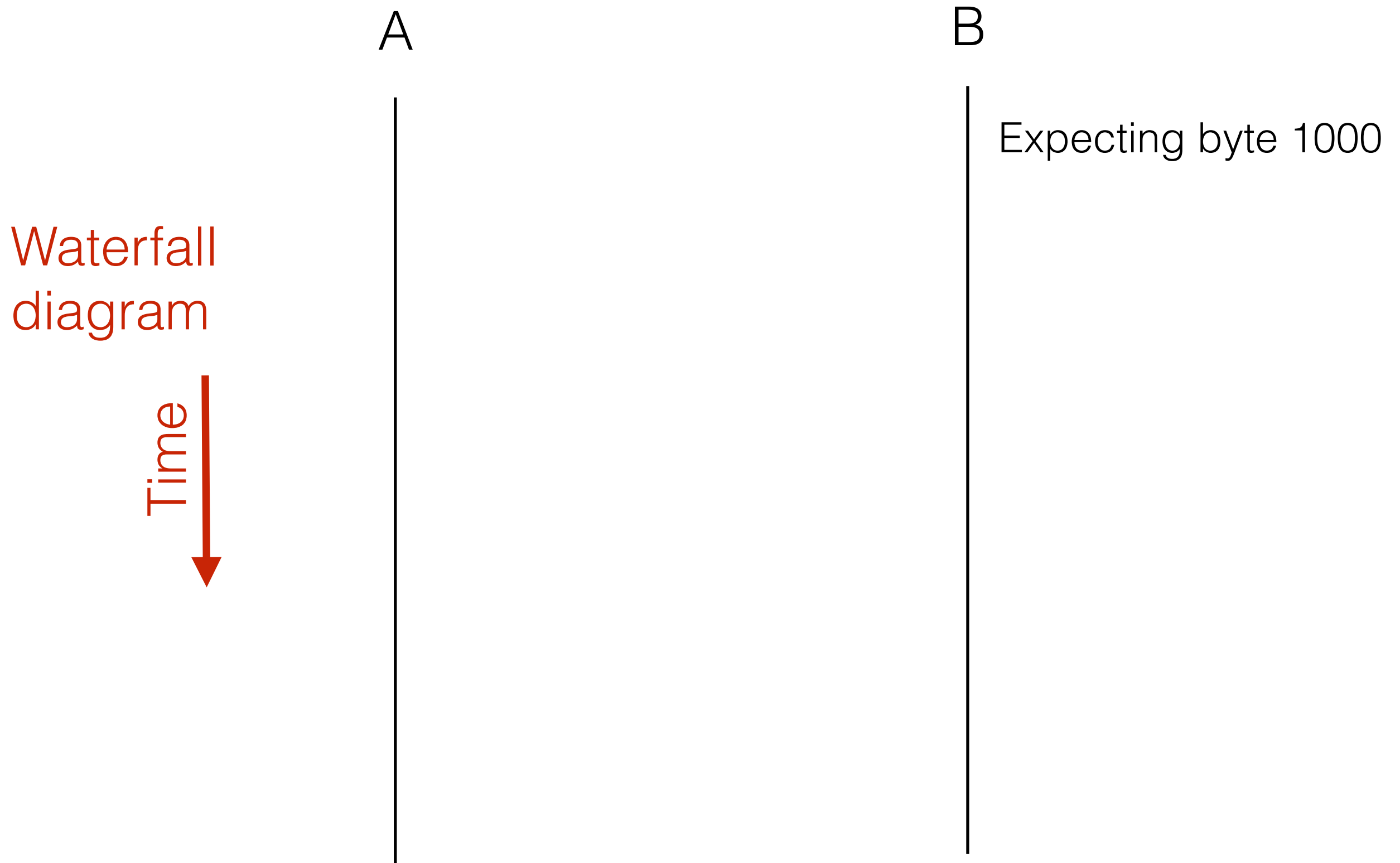


Reliability through acknowledgments
to determine whether something was received.

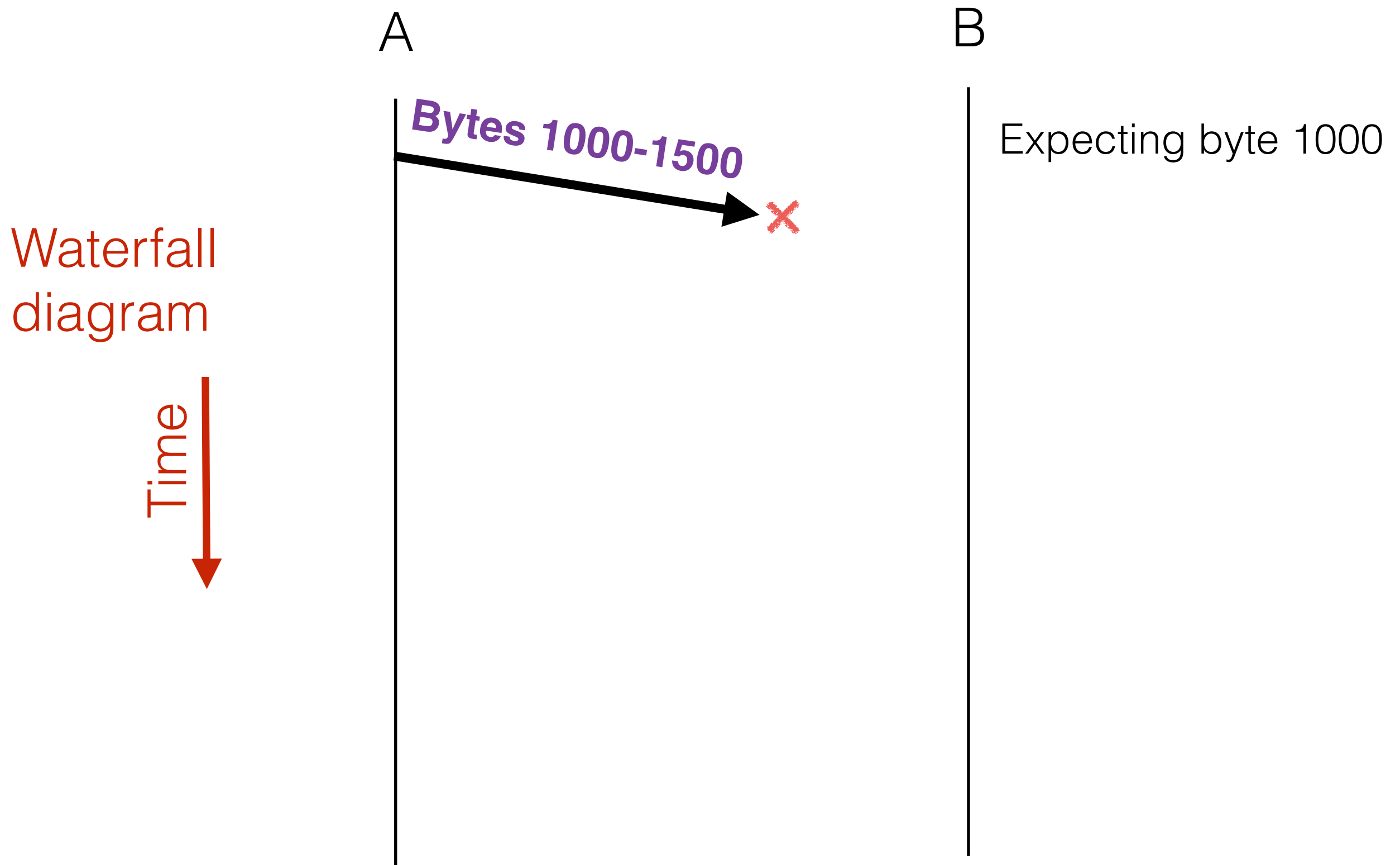
How does TCP achieve reliability?



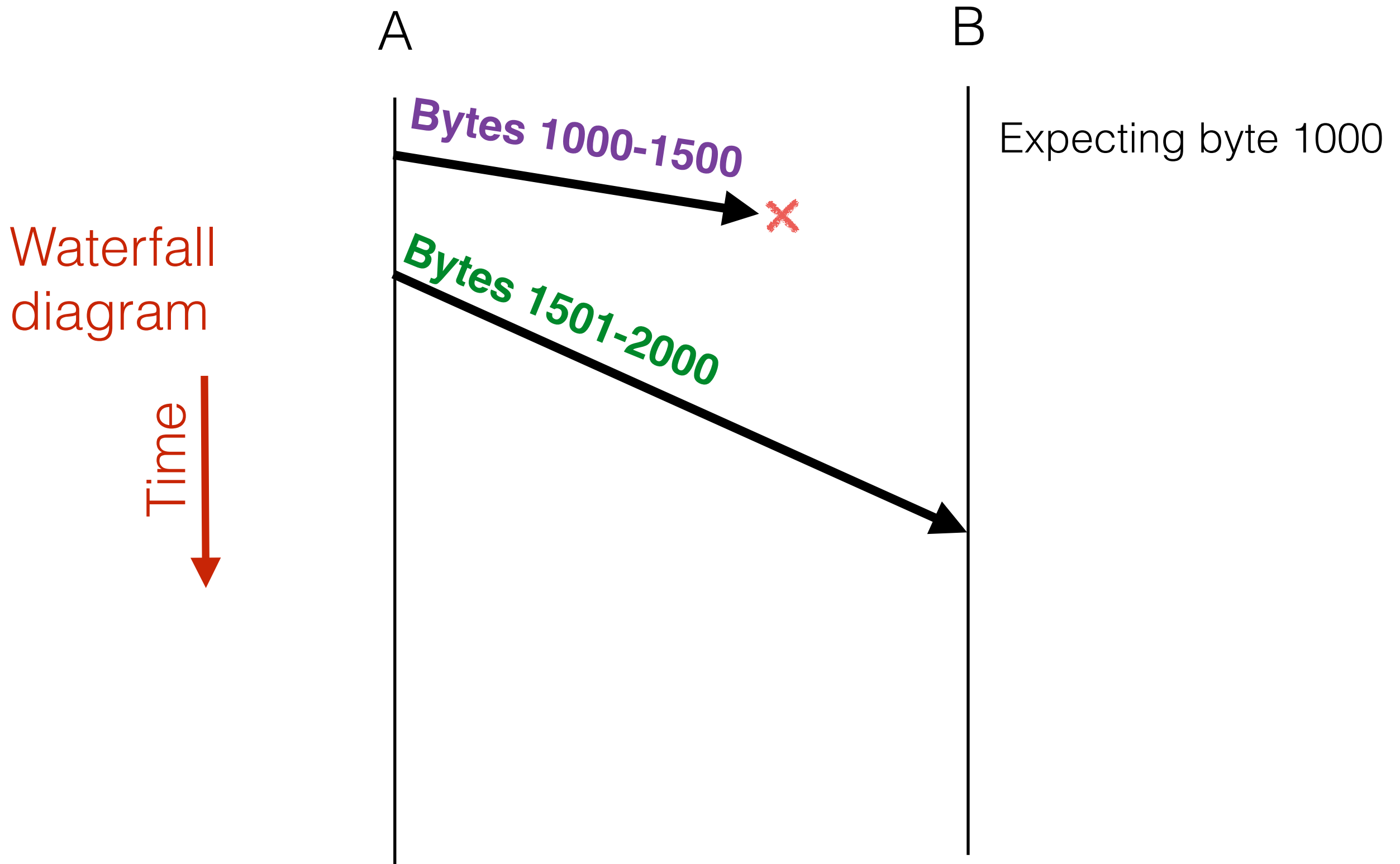
How does TCP achieve reliability?



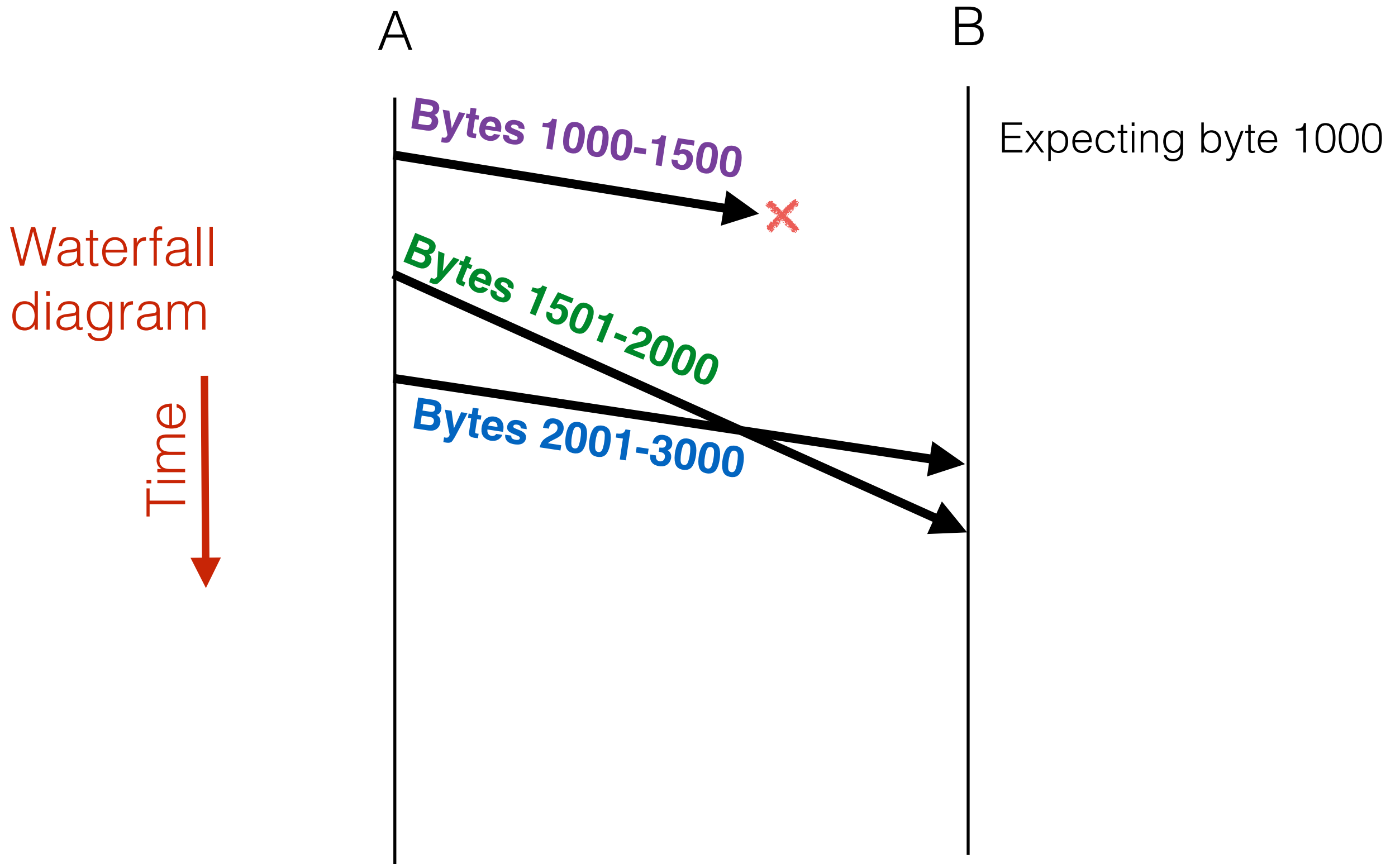
How does TCP achieve reliability?



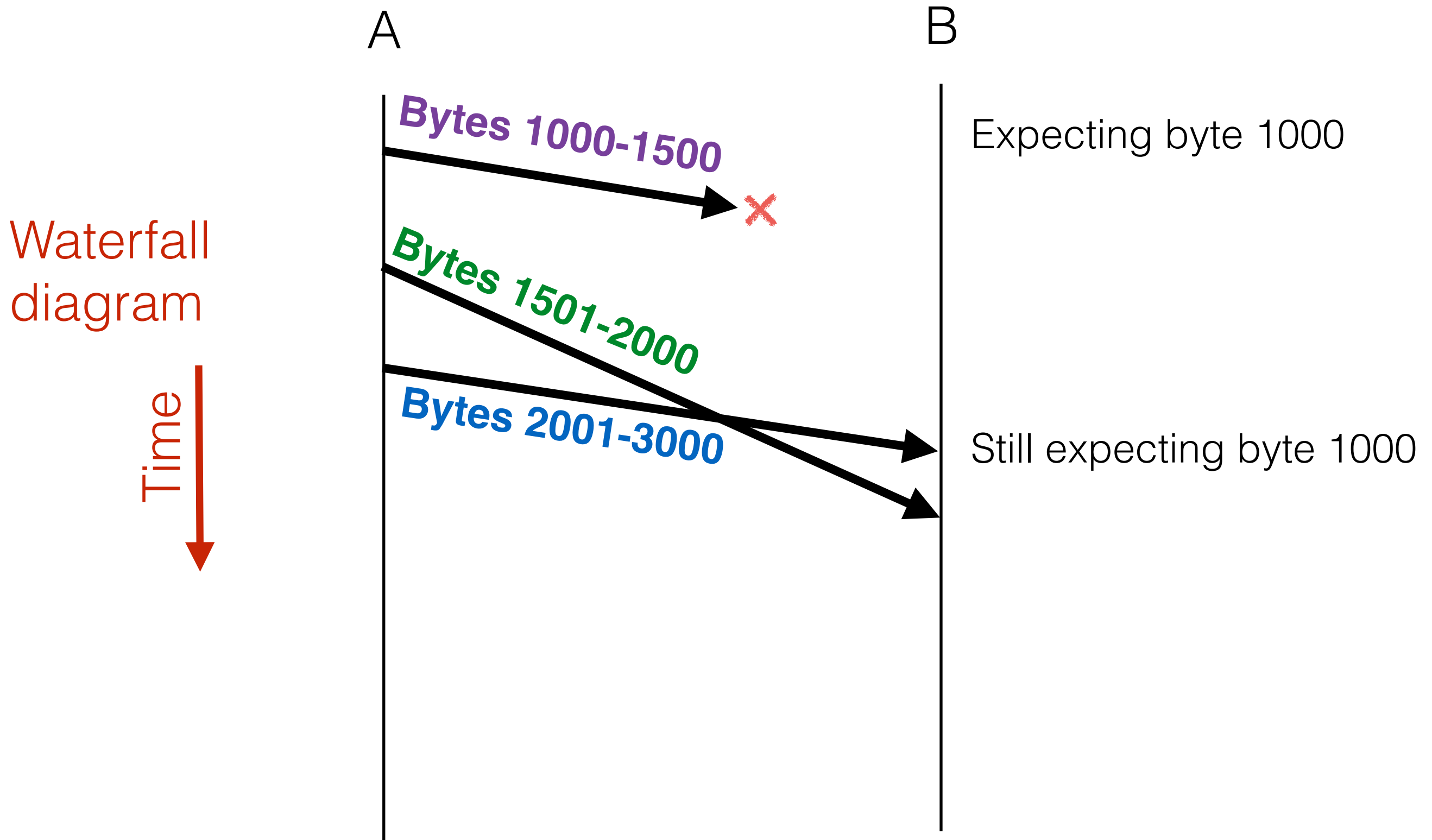
How does TCP achieve reliability?



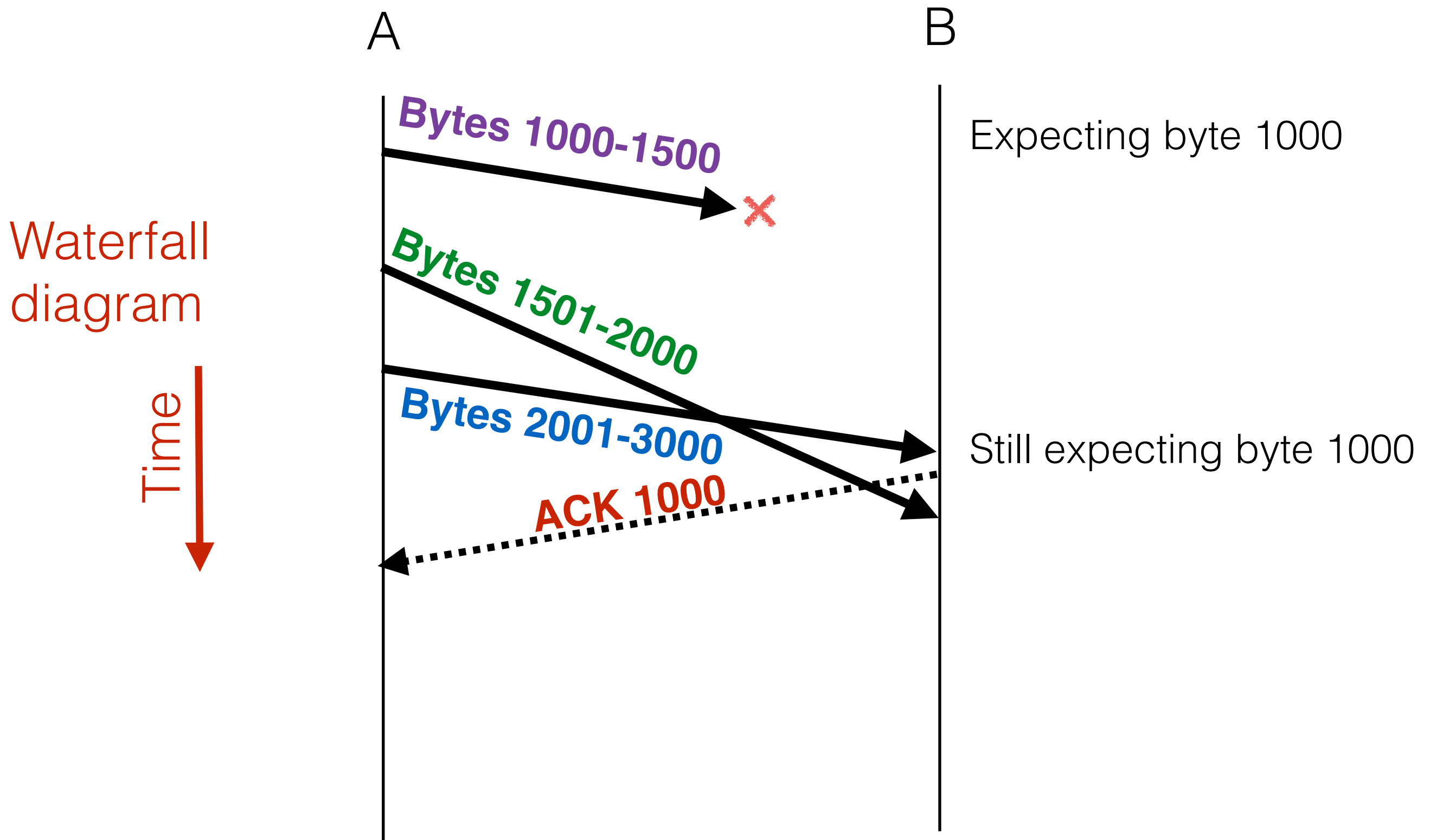
How does TCP achieve reliability?



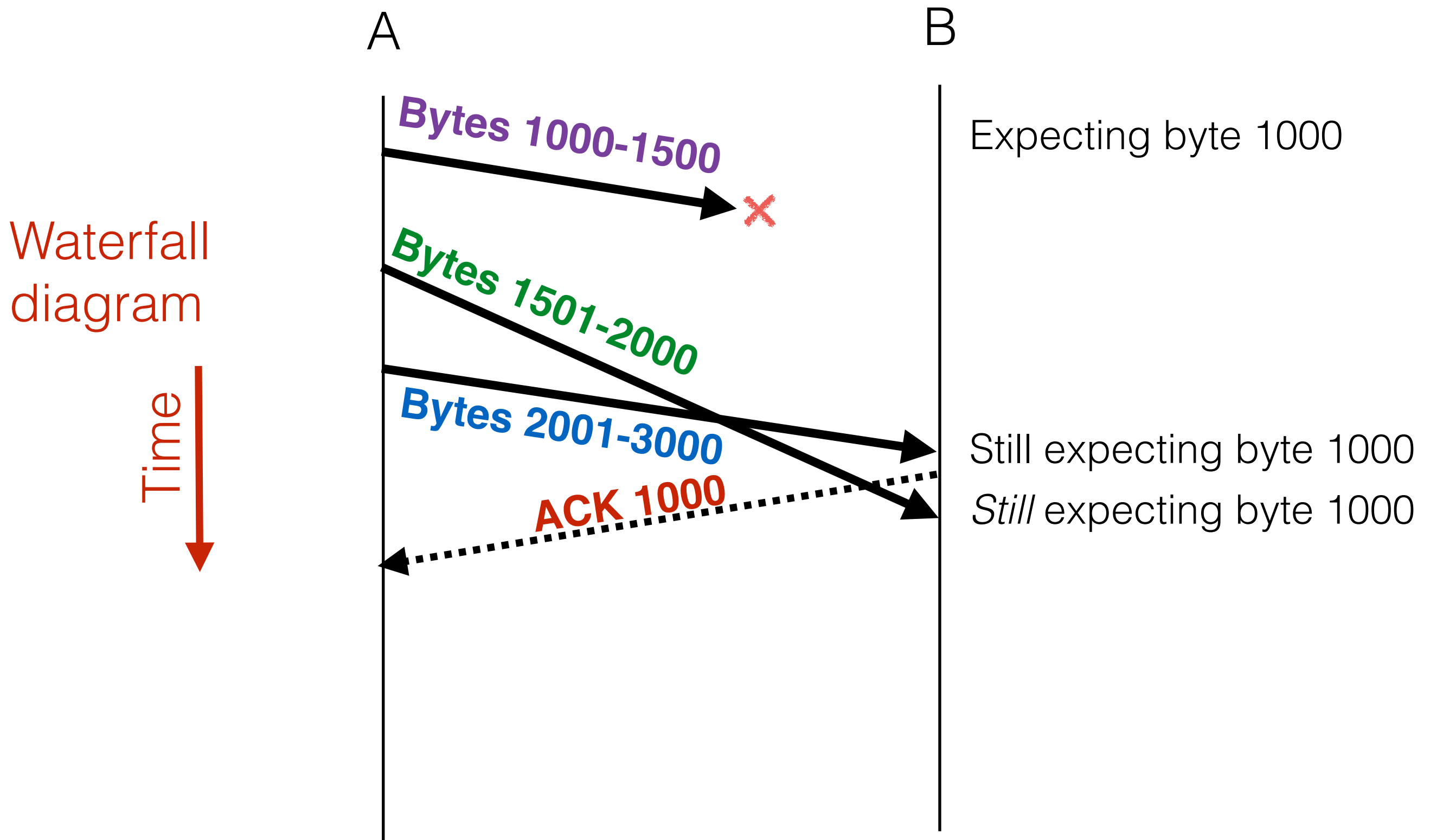
How does TCP achieve reliability?



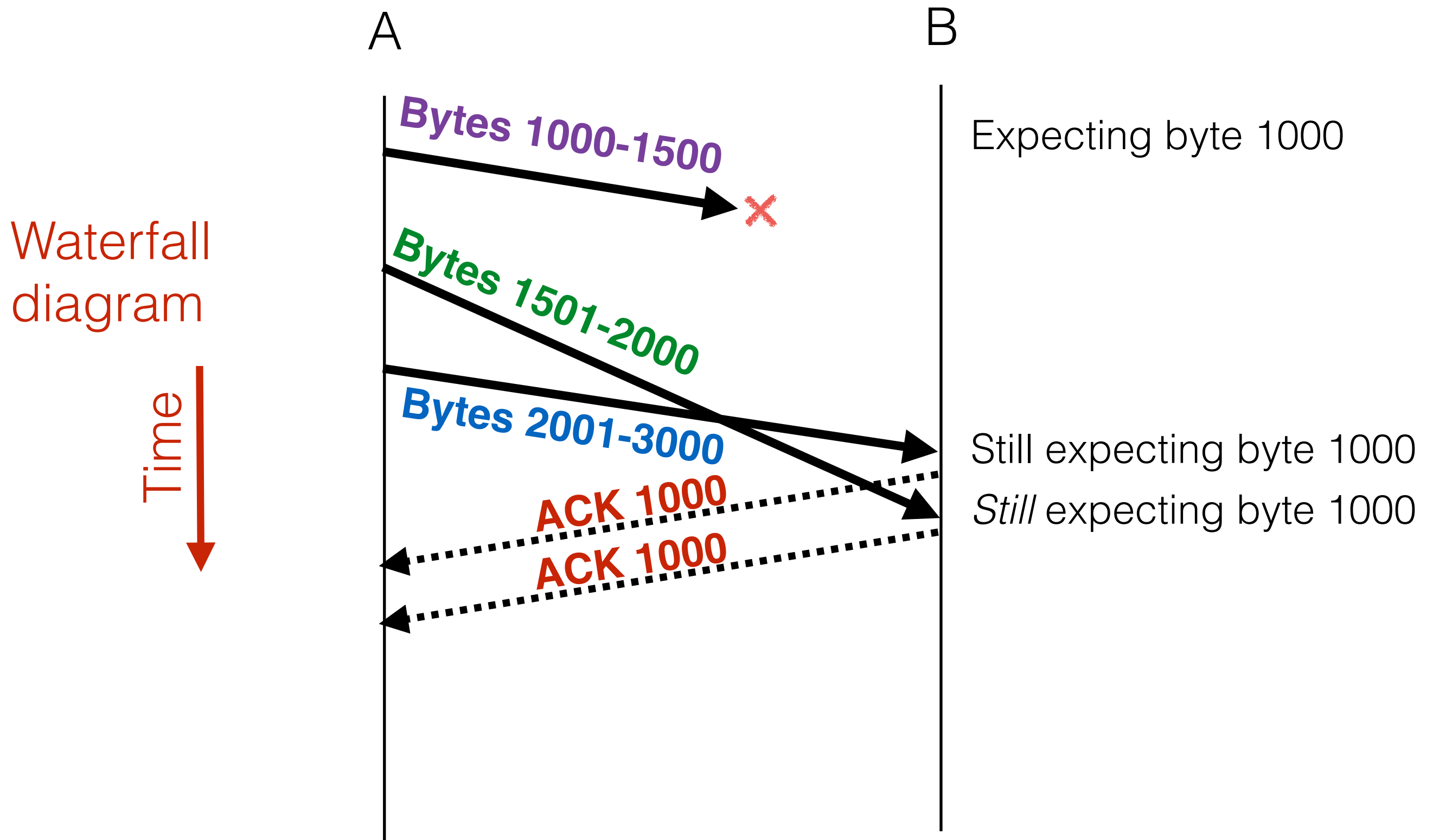
How does TCP achieve reliability?



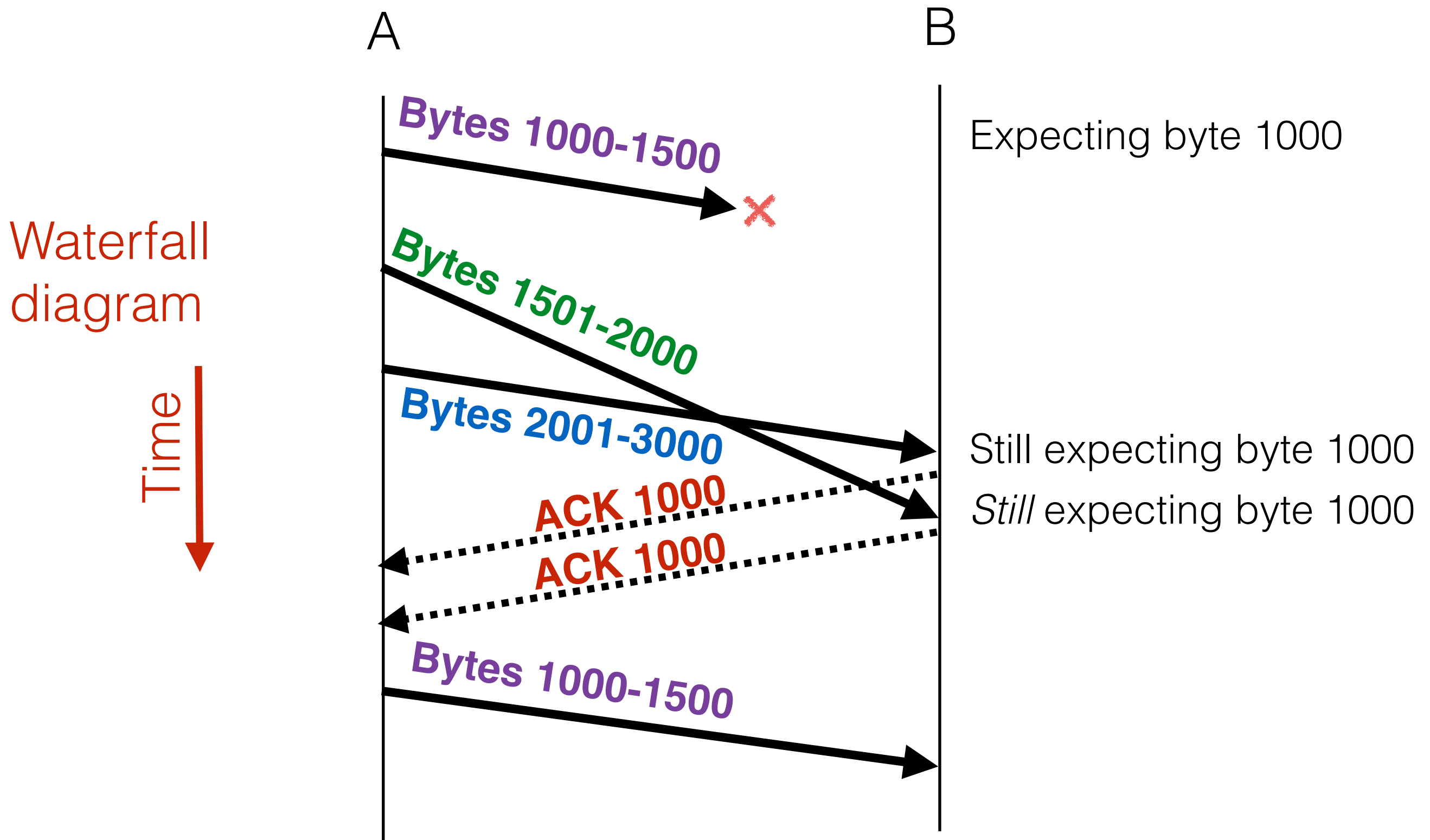
How does TCP achieve reliability?



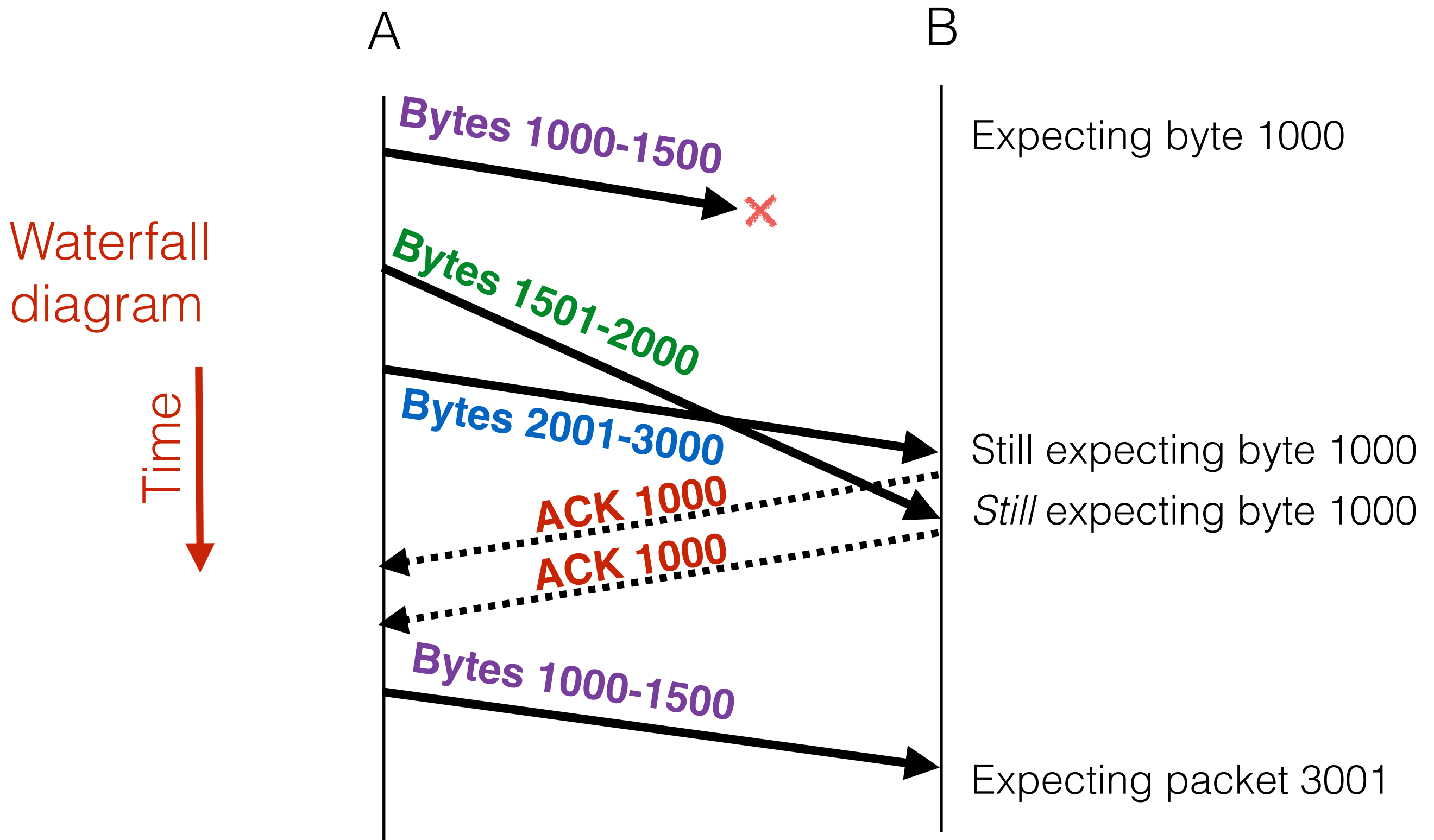
How does TCP achieve reliability?



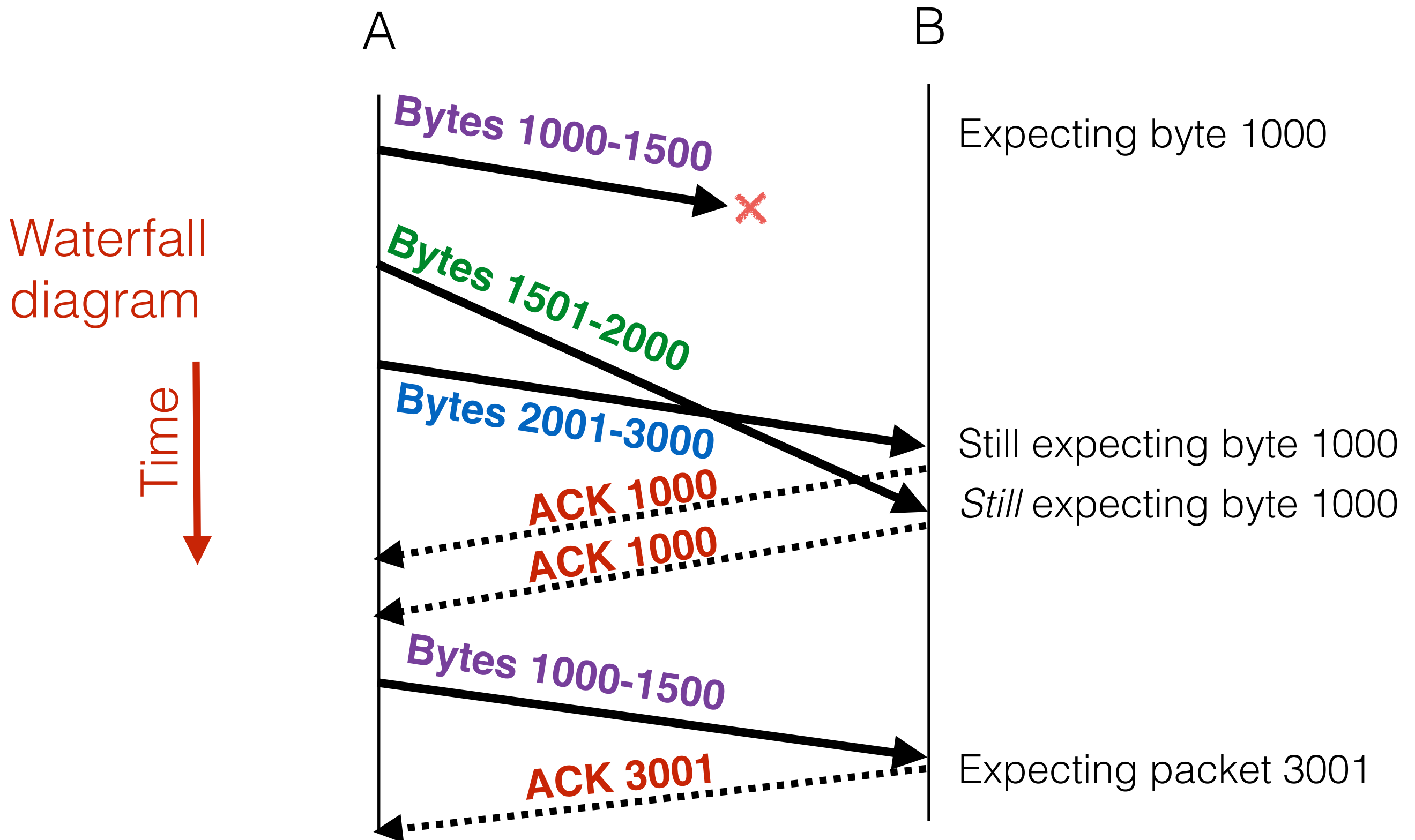
How does TCP achieve reliability?



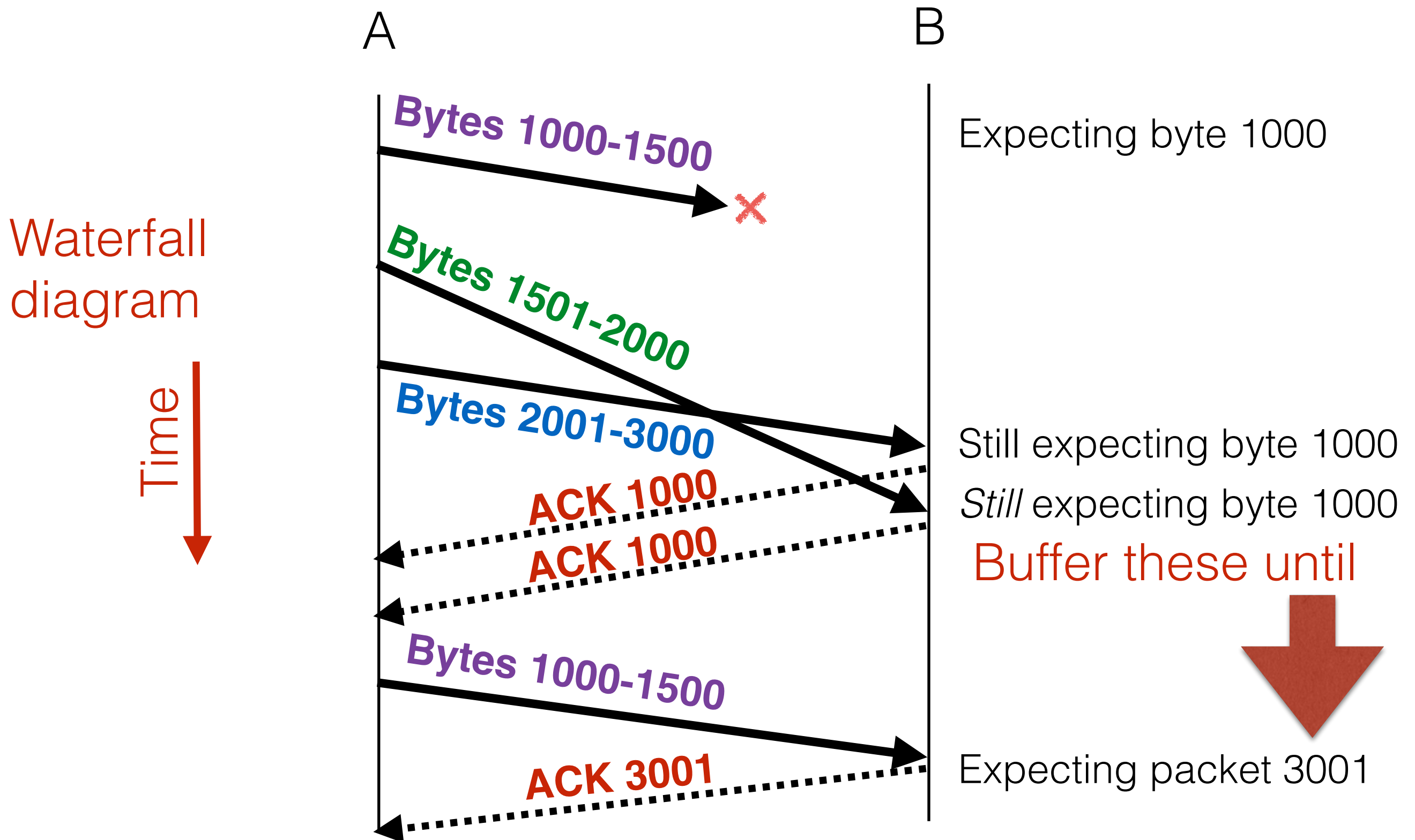
How does TCP achieve reliability?



How does TCP achieve reliability?



How does TCP achieve reliability?

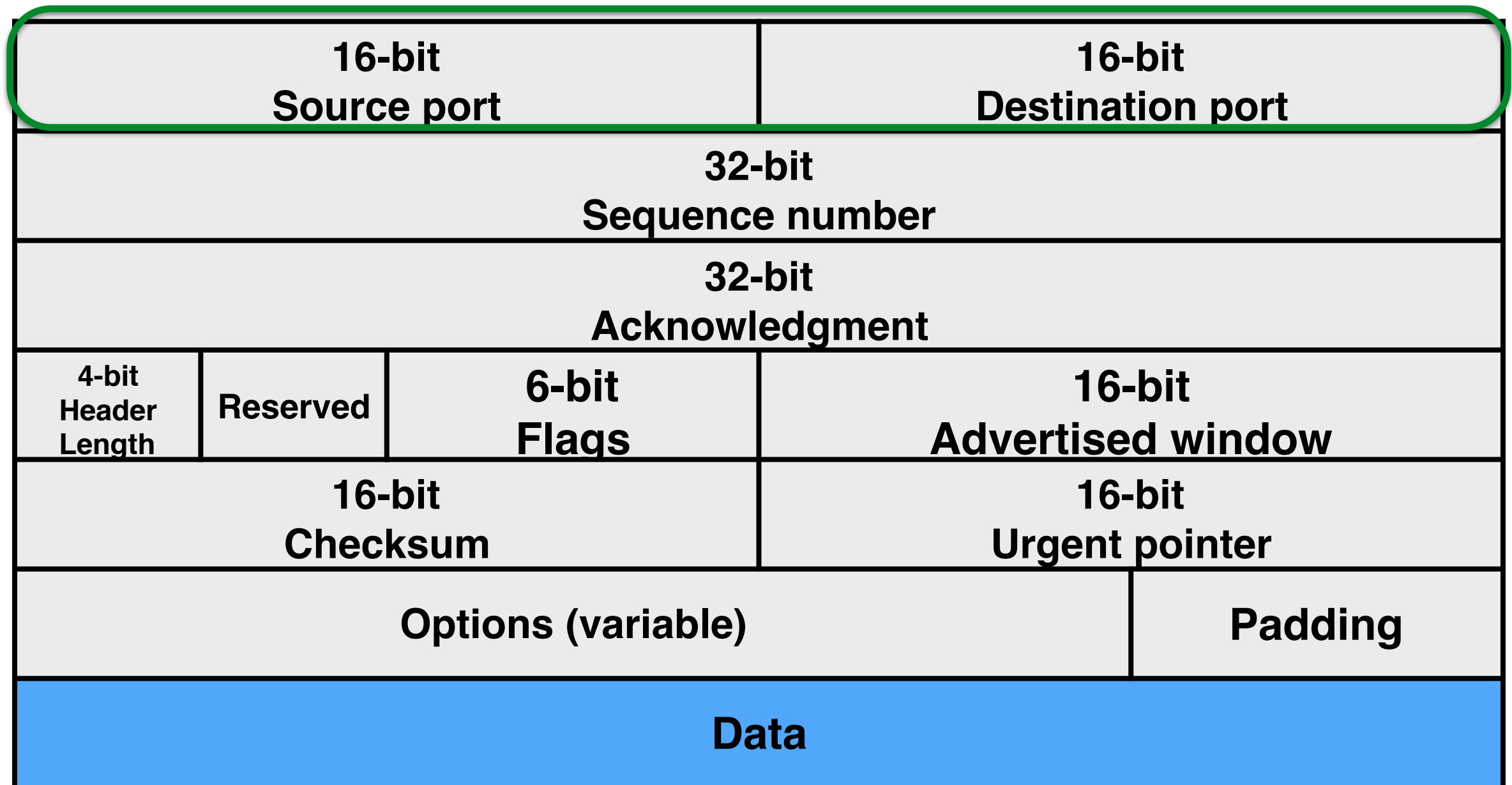


TCP congestion control

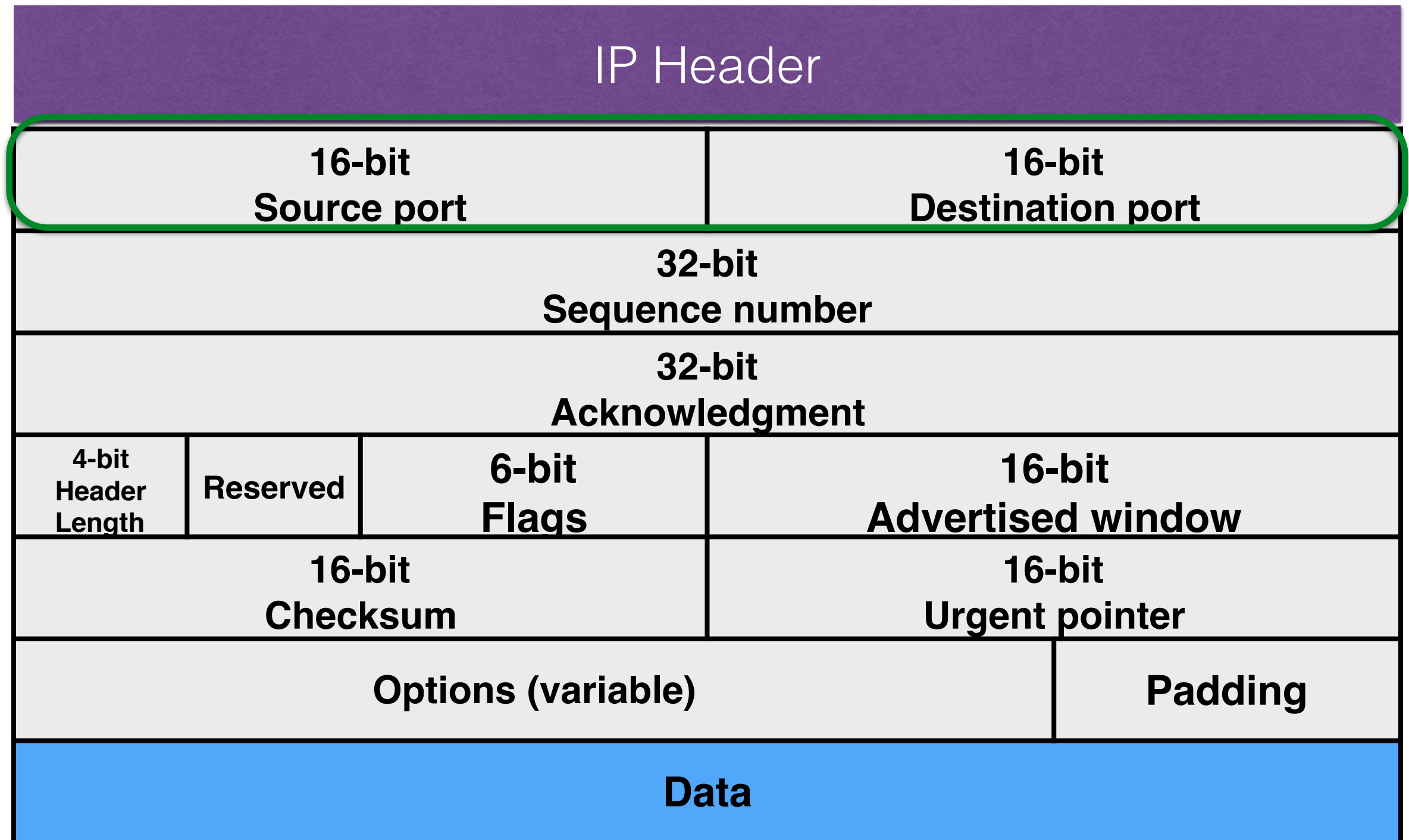
TCP's second job: don't break the network!

- Try to use as much of the network as is safe (does not adversely affect others' performance) and efficient (makes use of network capacity)
- Dynamically adapt how quickly you send based on the network path's capacity
- When an ACK doesn't come back, the network may be beyond capacity: slow down.

TCP header



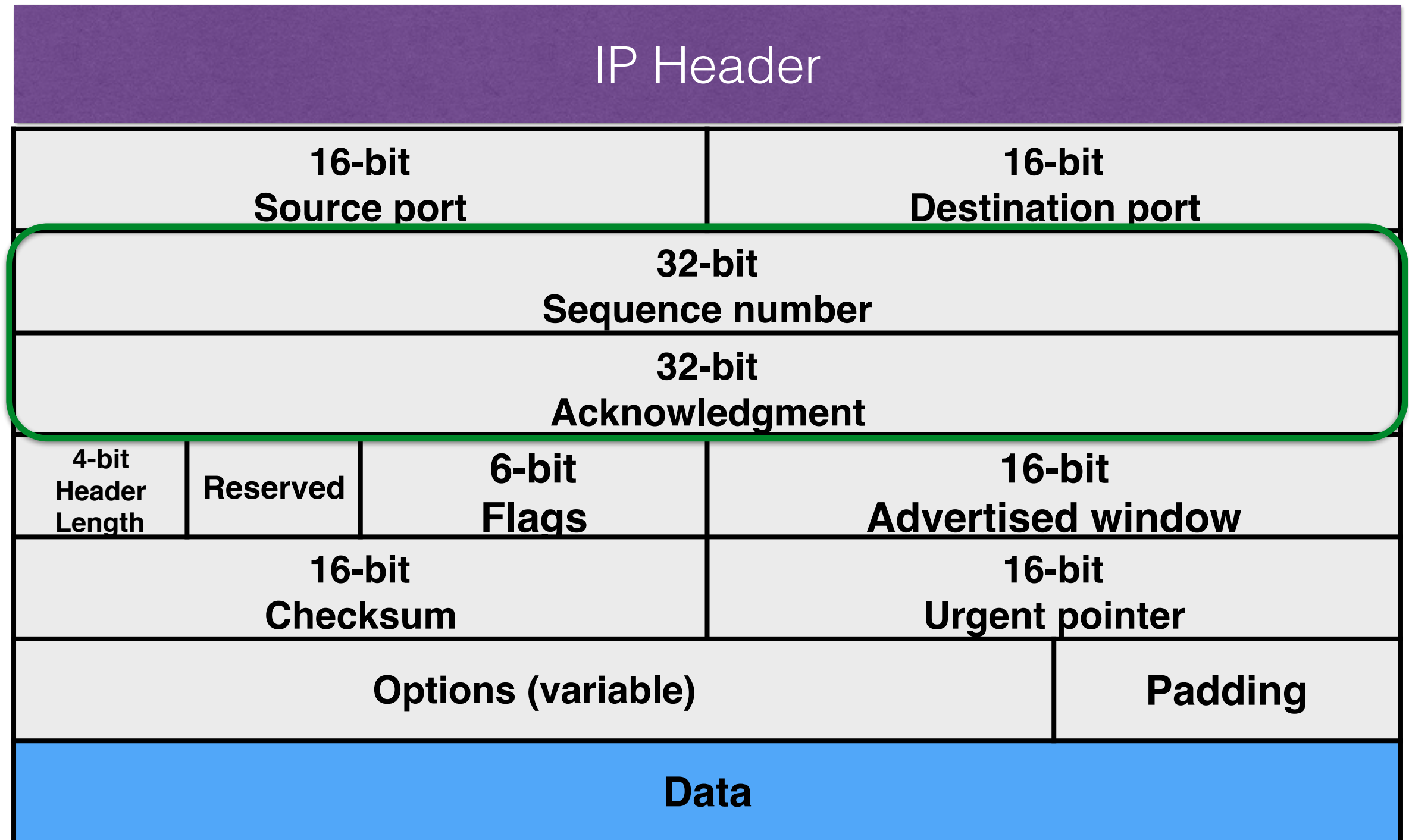
TCP header



TCP ports

- Ports are associated with **OS processes**
- Sandwiched between IP header and the application data
- {src IP/port, dst IP/port} : this 4-tuple uniquely identifies a TCP connection
- Some port numbers are well-known
 - 80 = HTTP
 - 53 = DNS

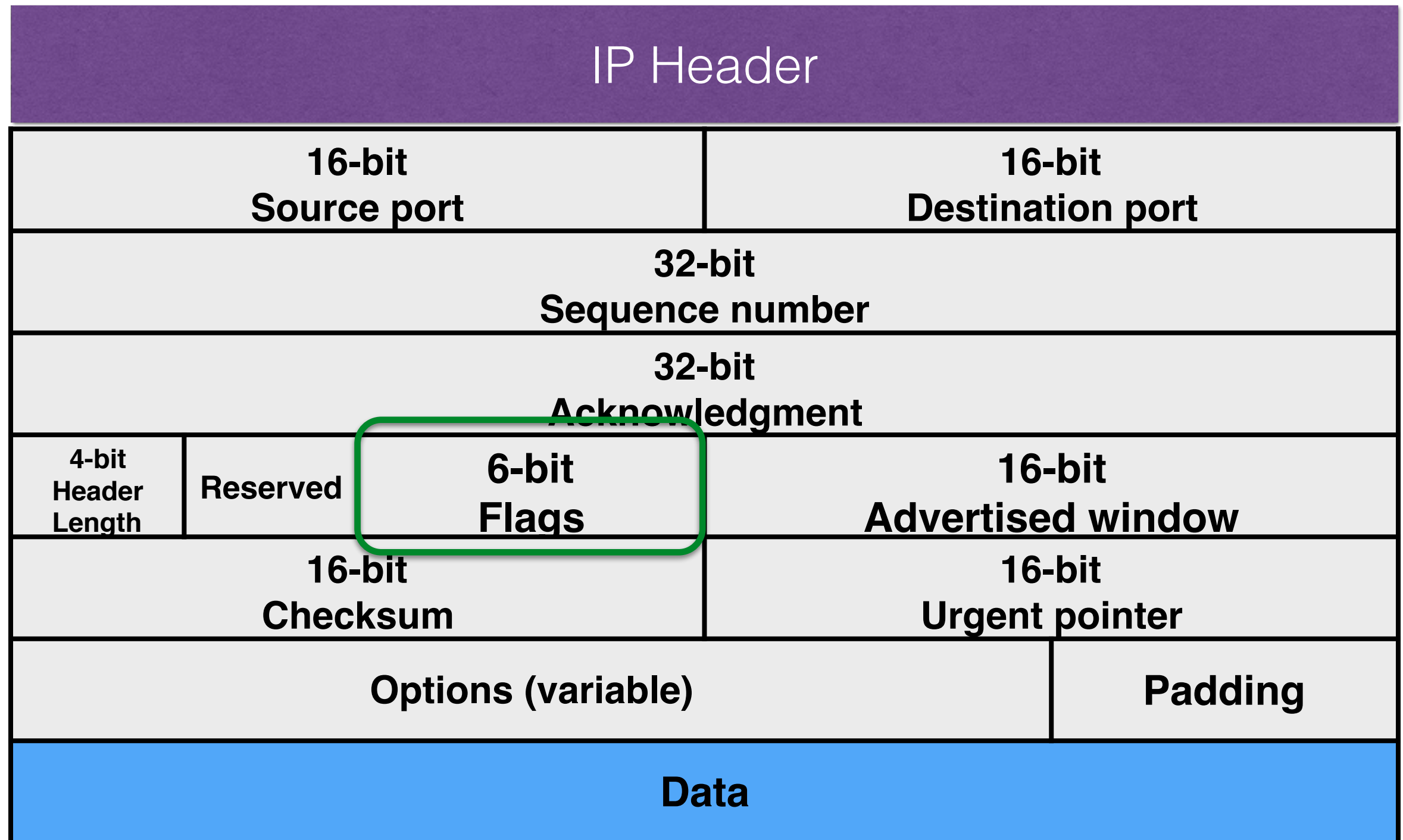
TCP header



TCP seqno

- Each byte in the byte stream has a unique “sequence number”
 - Unique for both directions
- “Sequence number” in the header = sequence number of the **first** byte in the packet’s data
- Next sequence number = previous seqno + previous packet’s data size
- “Acknowledgment” in the header = the **next** seqno you expect from the other end-host

TCP header



TCP flags

- SYN
 - Used for setting up a connection
- ACK
 - Acknowledgments, for data and “control” packets
- FIN
- RST

Setting up a connection

Three-way handshake

A

B

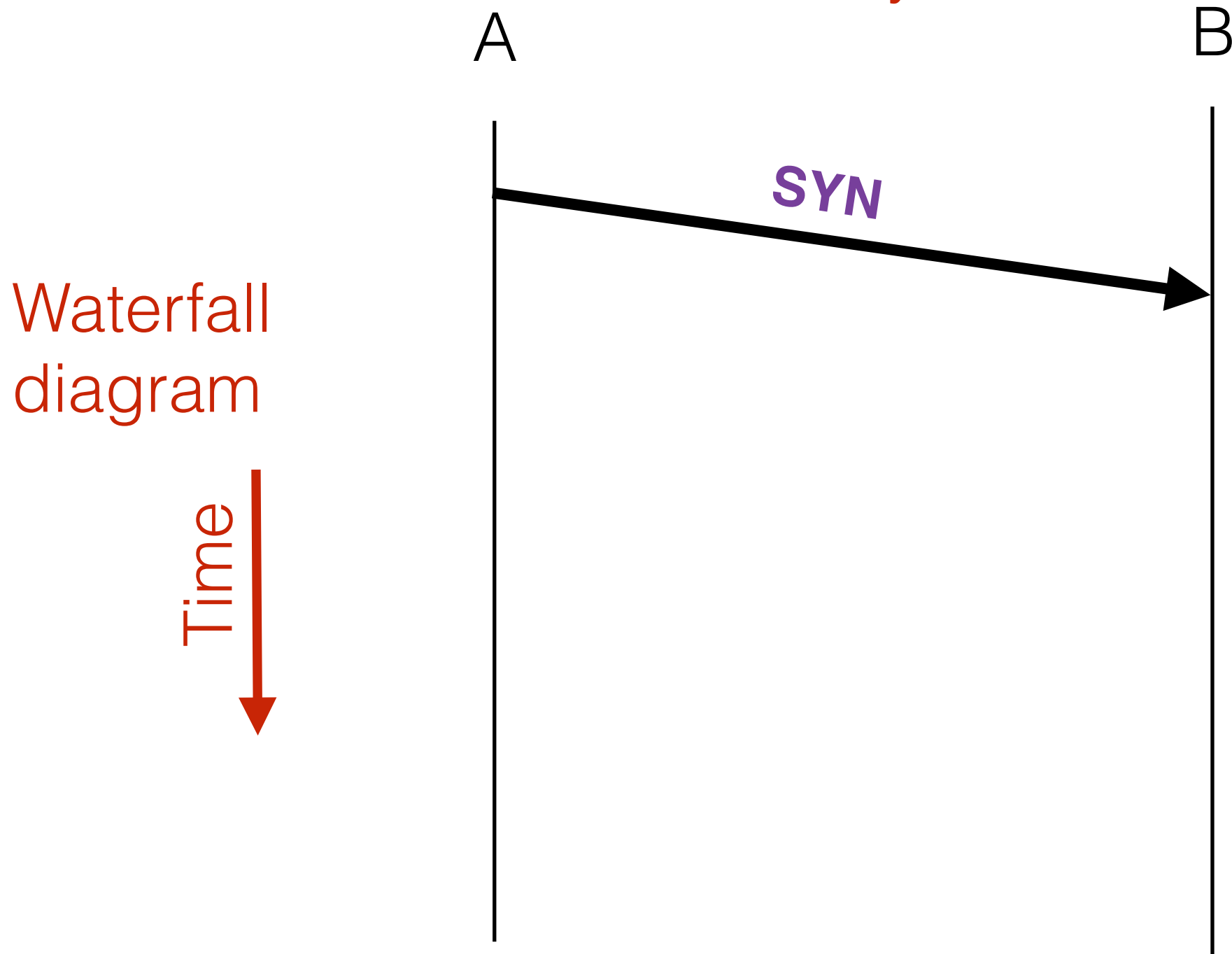
Waterfall
diagram

Time



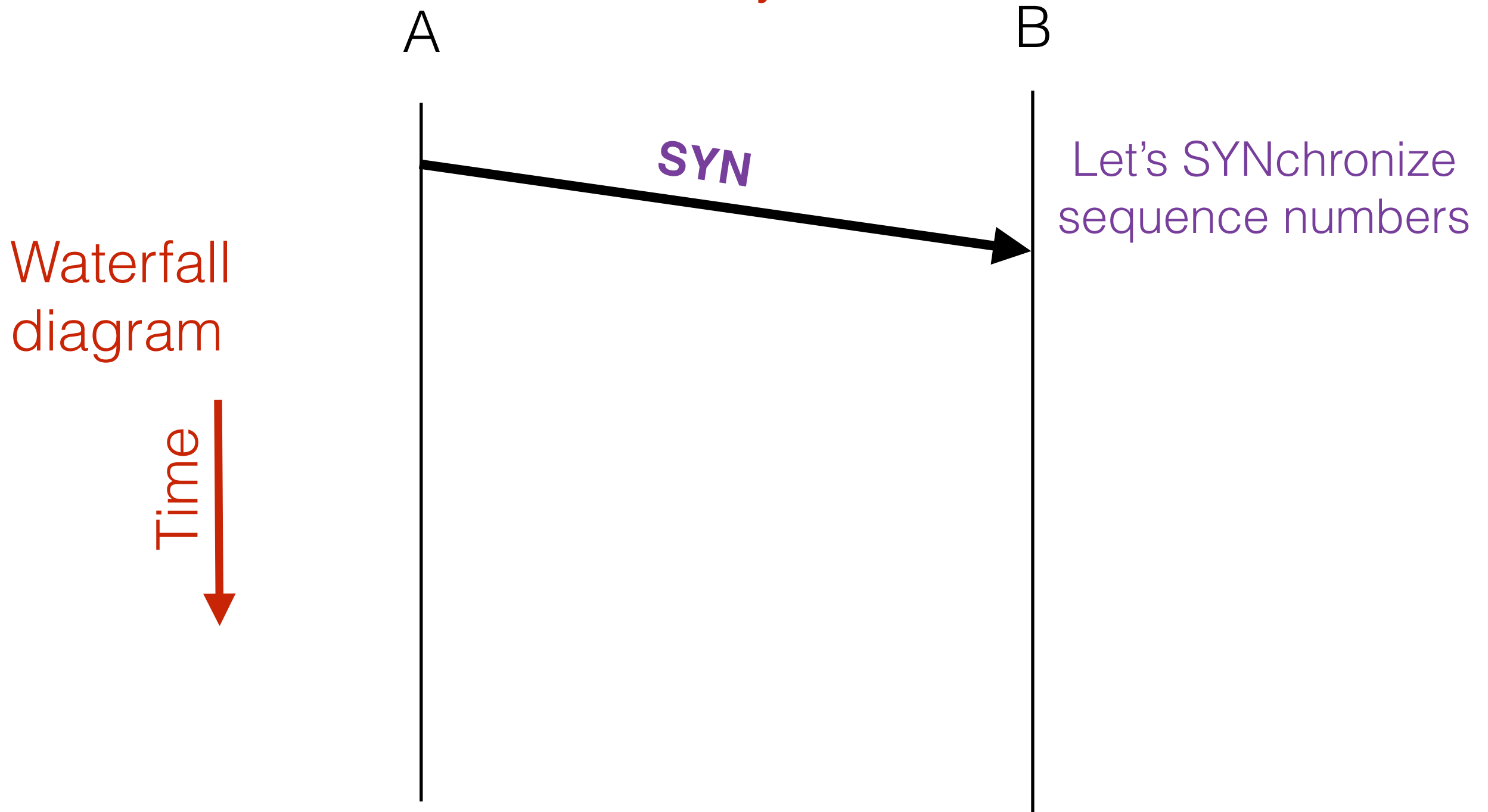
Setting up a connection

Three-way handshake



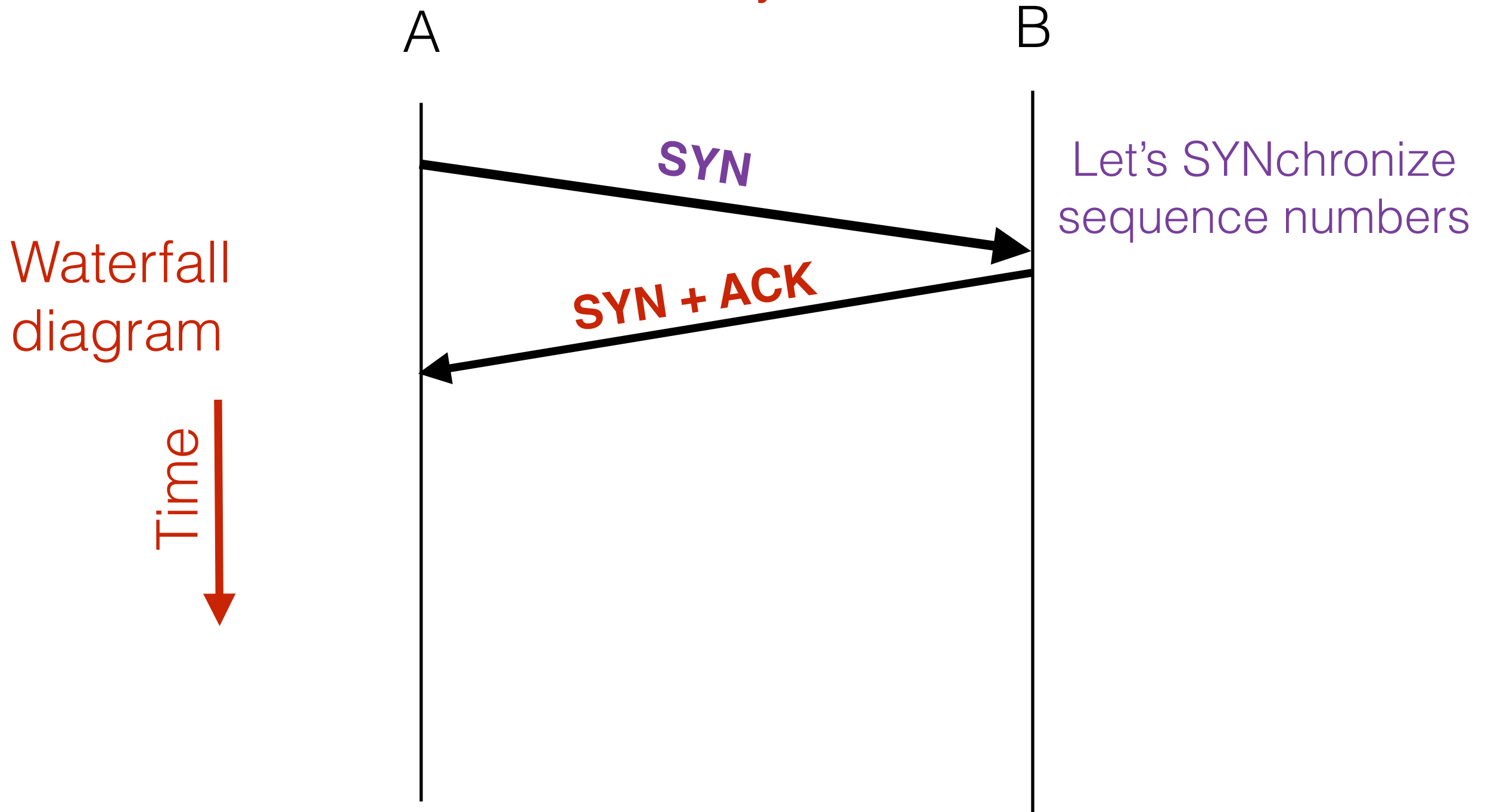
Setting up a connection

Three-way handshake



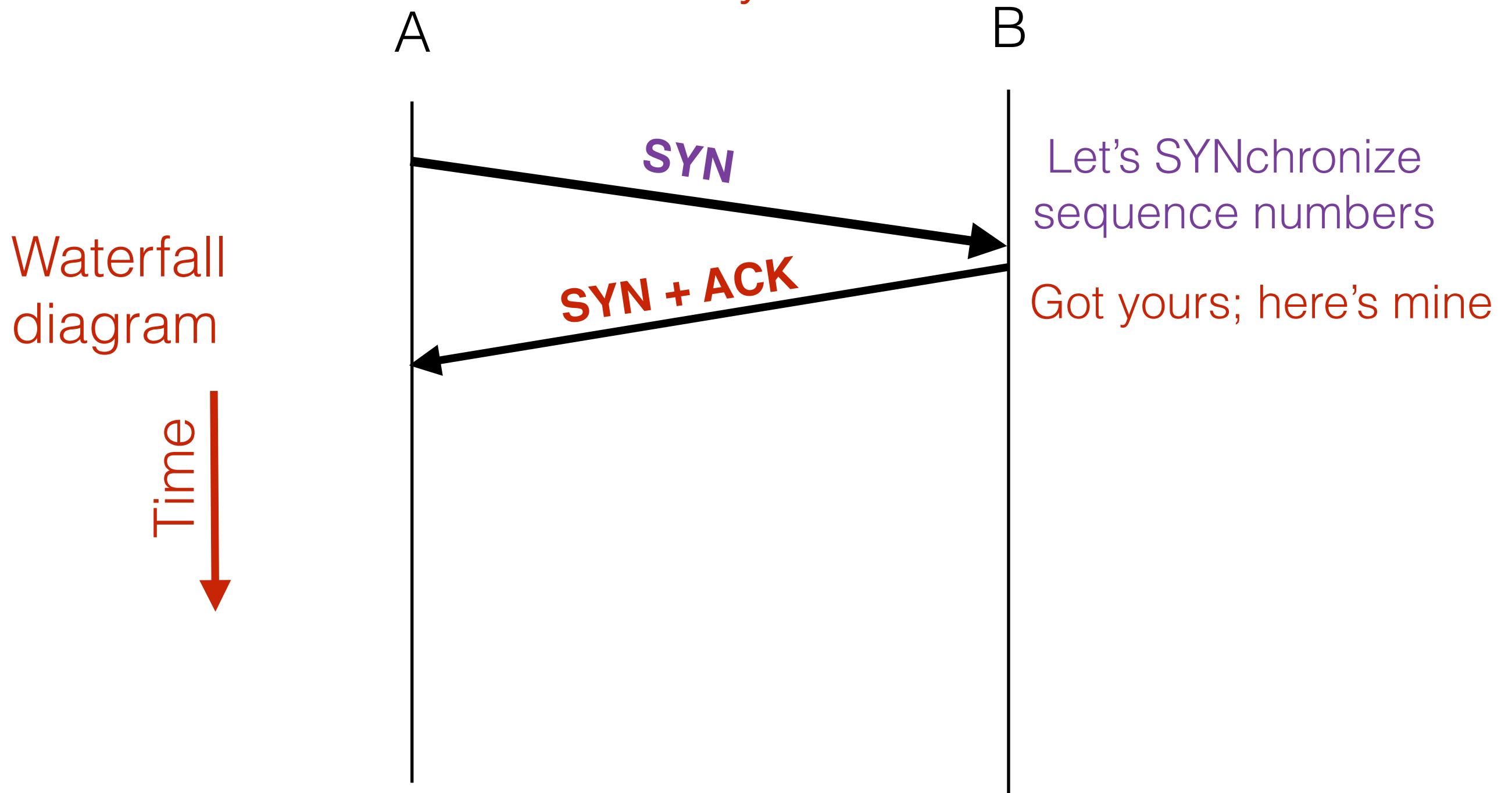
Setting up a connection

Three-way handshake



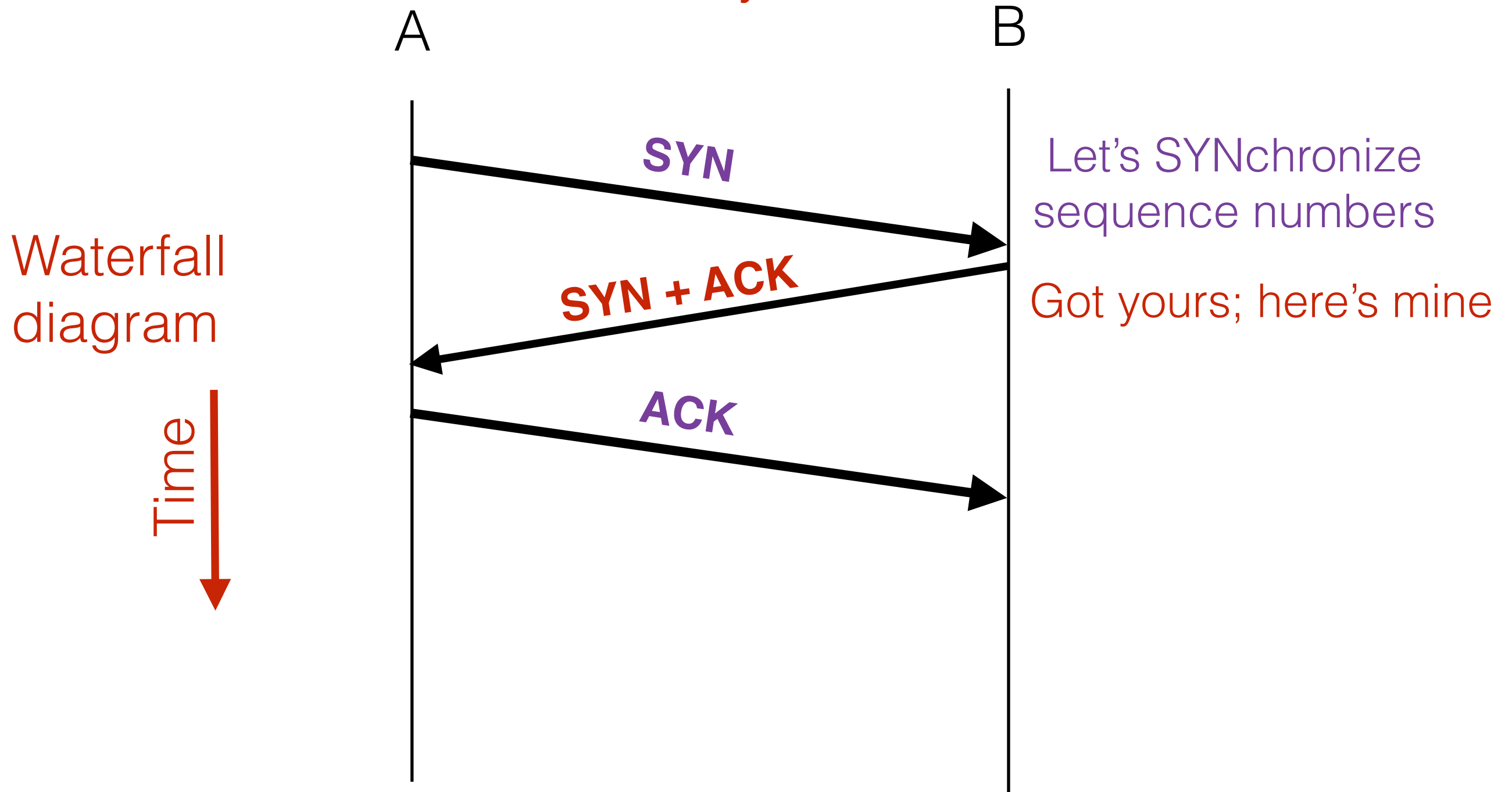
Setting up a connection

Three-way handshake



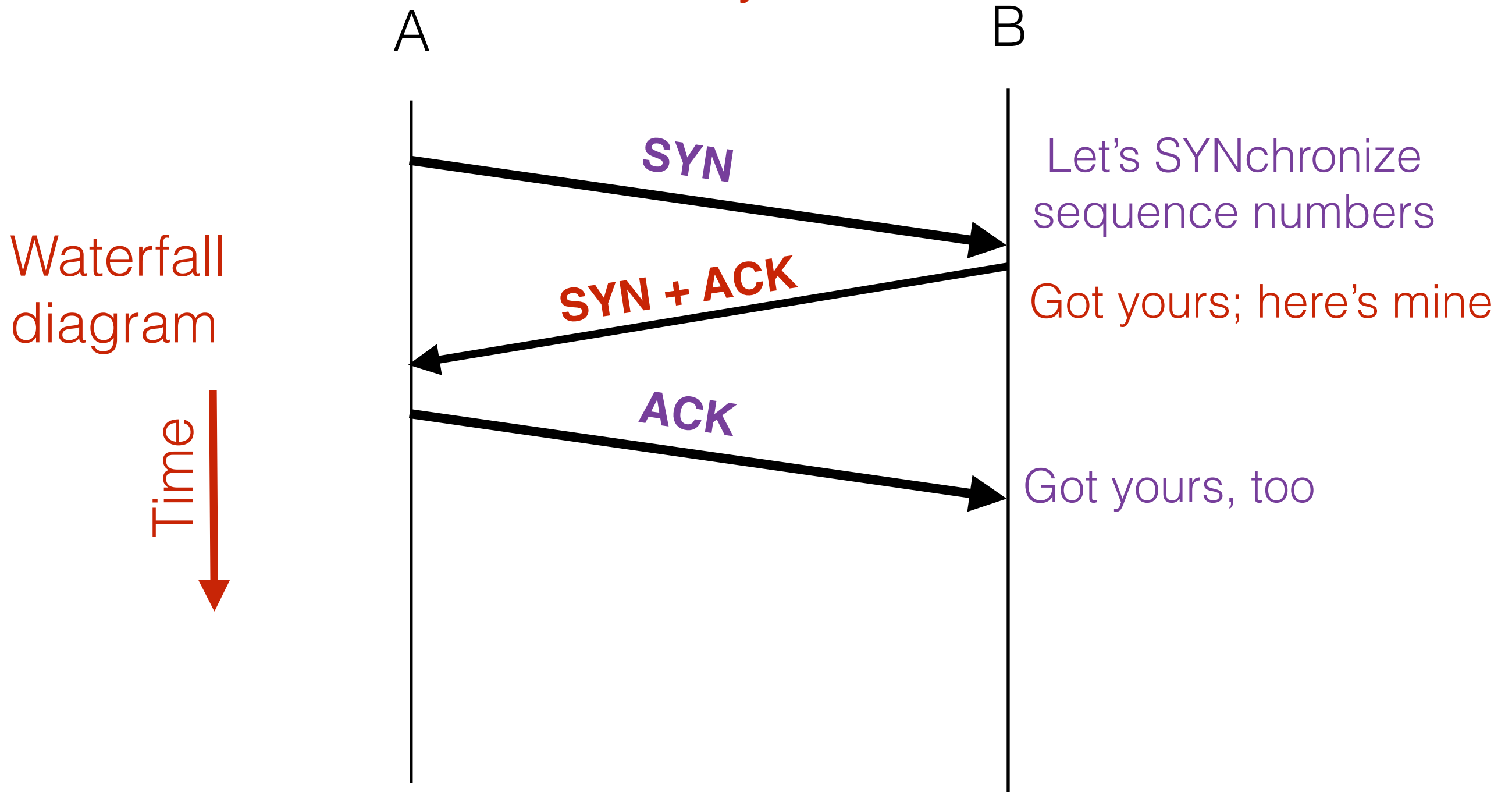
Setting up a connection

Three-way handshake



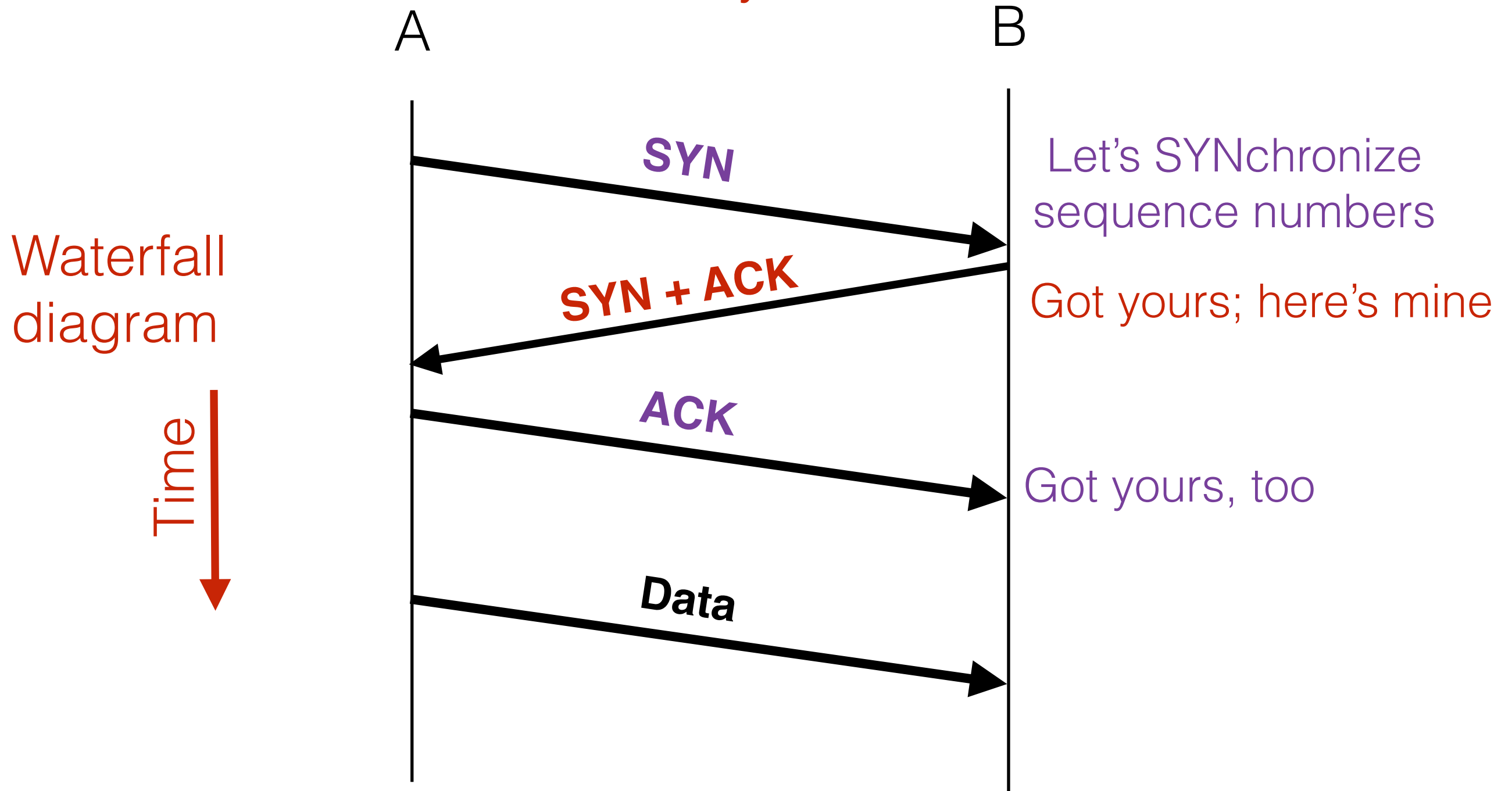
Setting up a connection

Three-way handshake



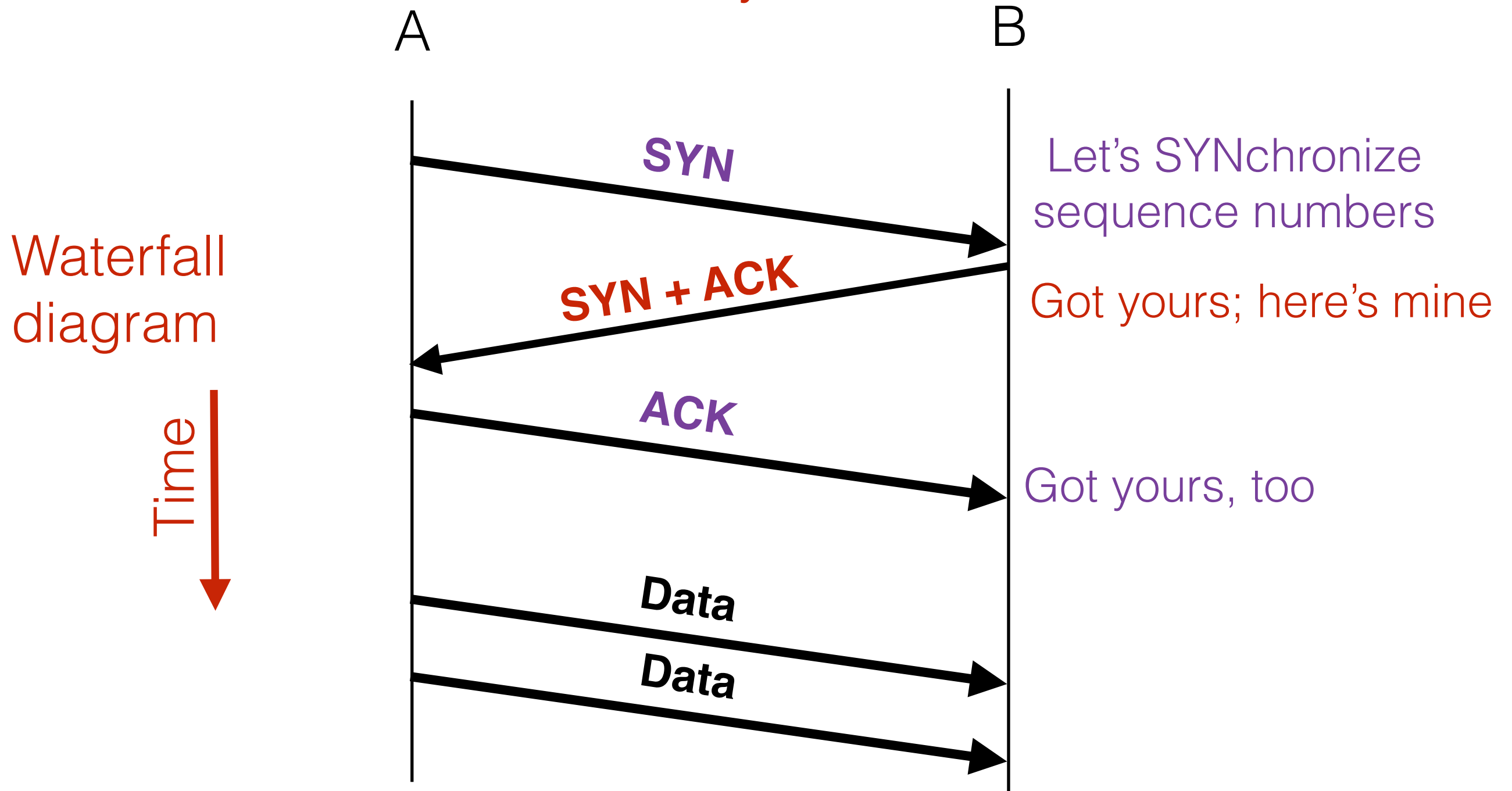
Setting up a connection

Three-way handshake



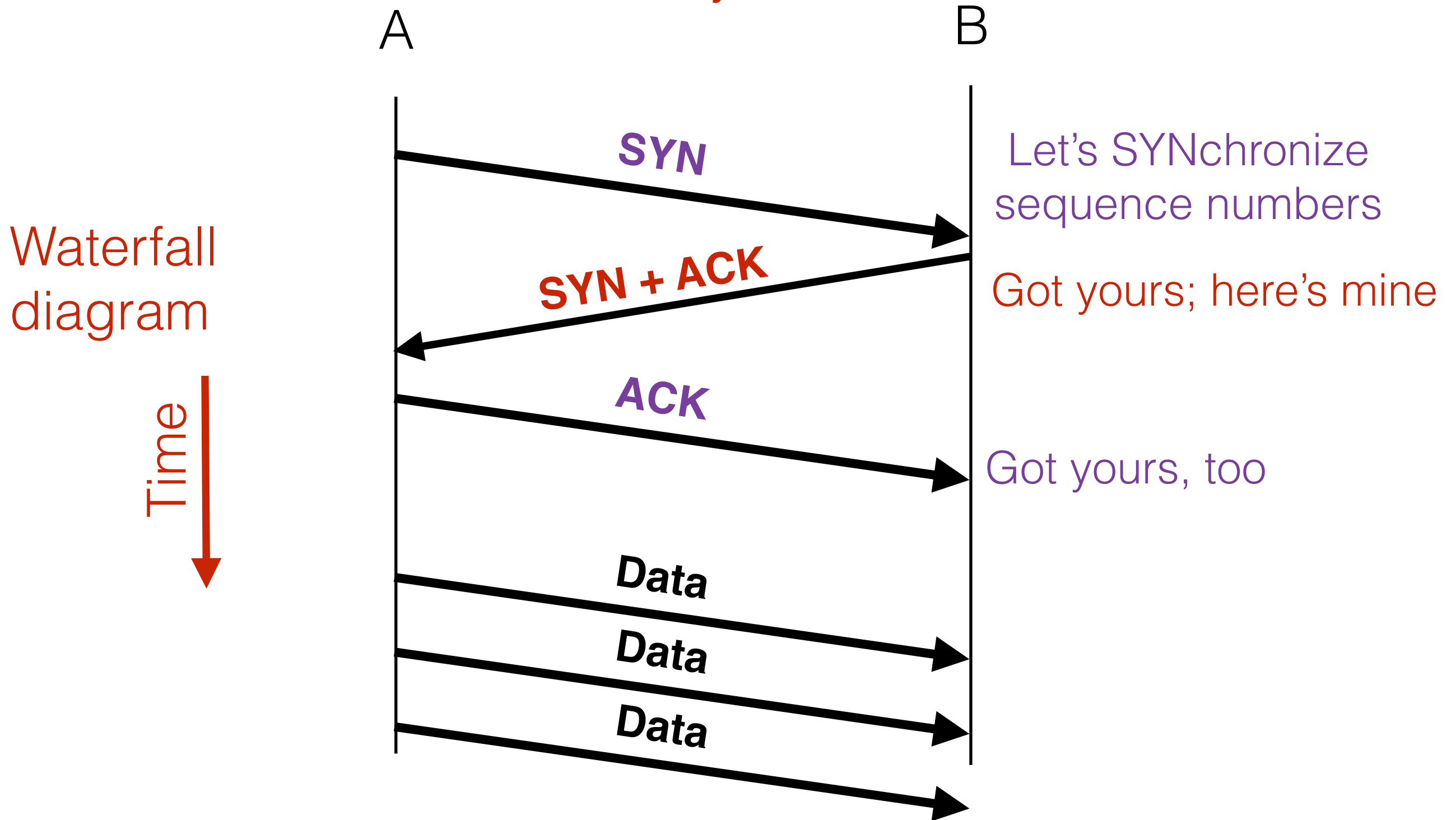
Setting up a connection

Three-way handshake



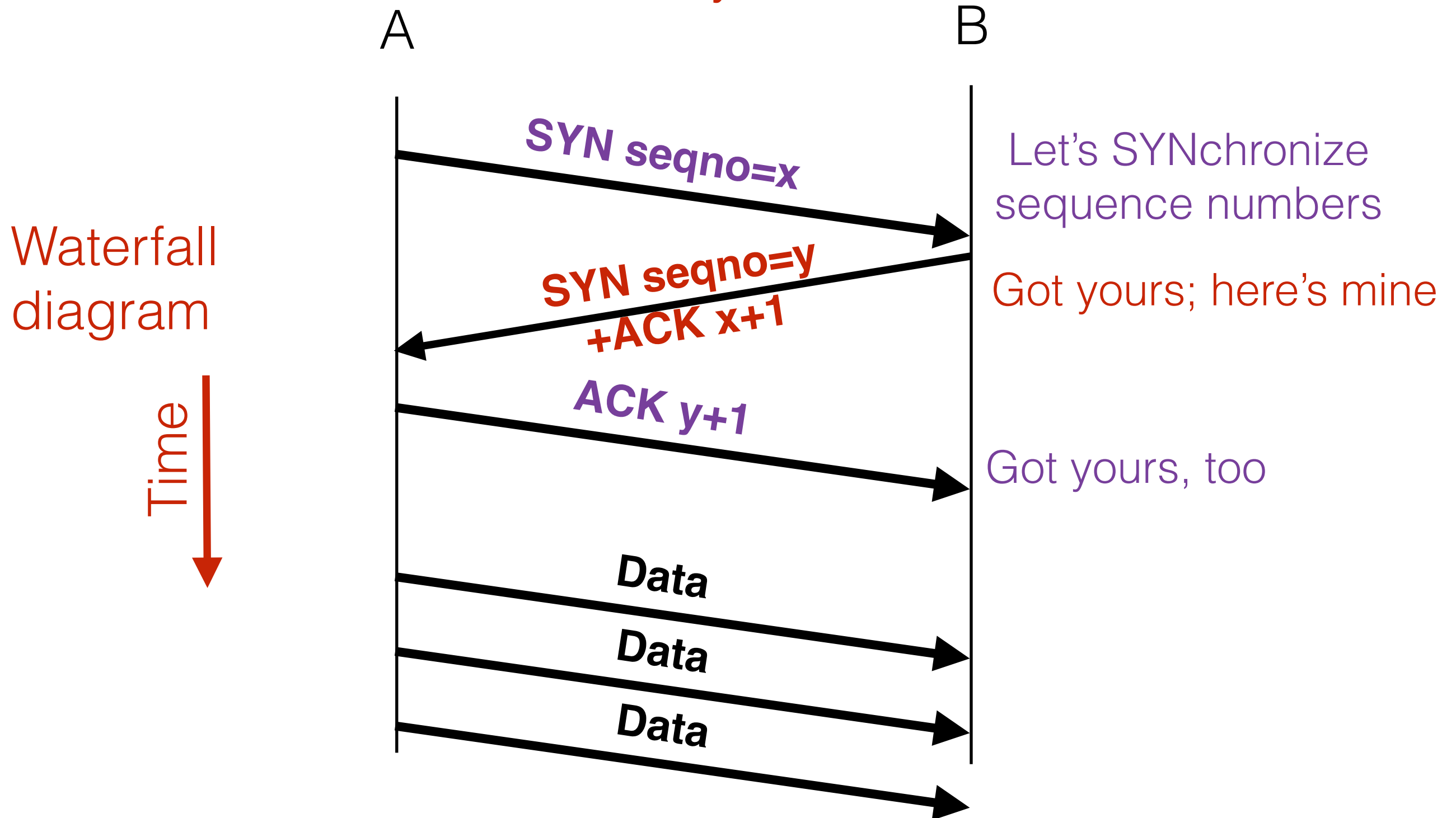
Setting up a connection

Three-way handshake



Setting up a connection

Three-way handshake



TCP flags

- SYN
- ACK
- FIN: Let's shut this down (two-way)
 - FIN
 - FIN+ACK
- RST: I'm shutting you down
 - Says "delete all your local state, because I don't know what you're talking about"

Attacks

- SYN flooding
- Injection attacks
- Opt-ack attack

SYN flooding

SYN flooding

Recall the three-way handshake:

A

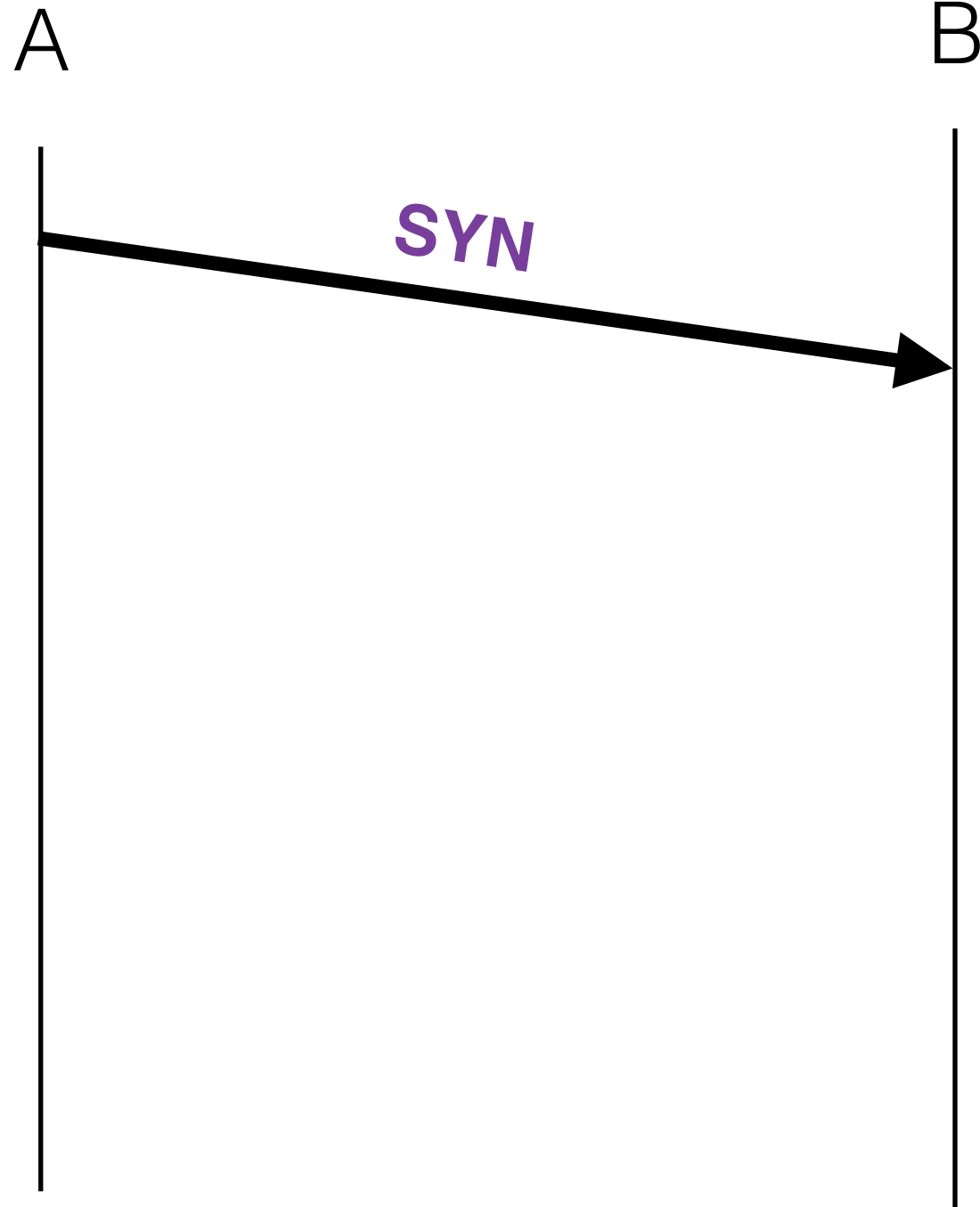
B

Waterfall
diagram



SYN flooding

Recall the three-way handshake:



Waterfall
diagram



SYN flooding

Recall the three-way handshake:

A

B

SYN

Waterfall
diagram

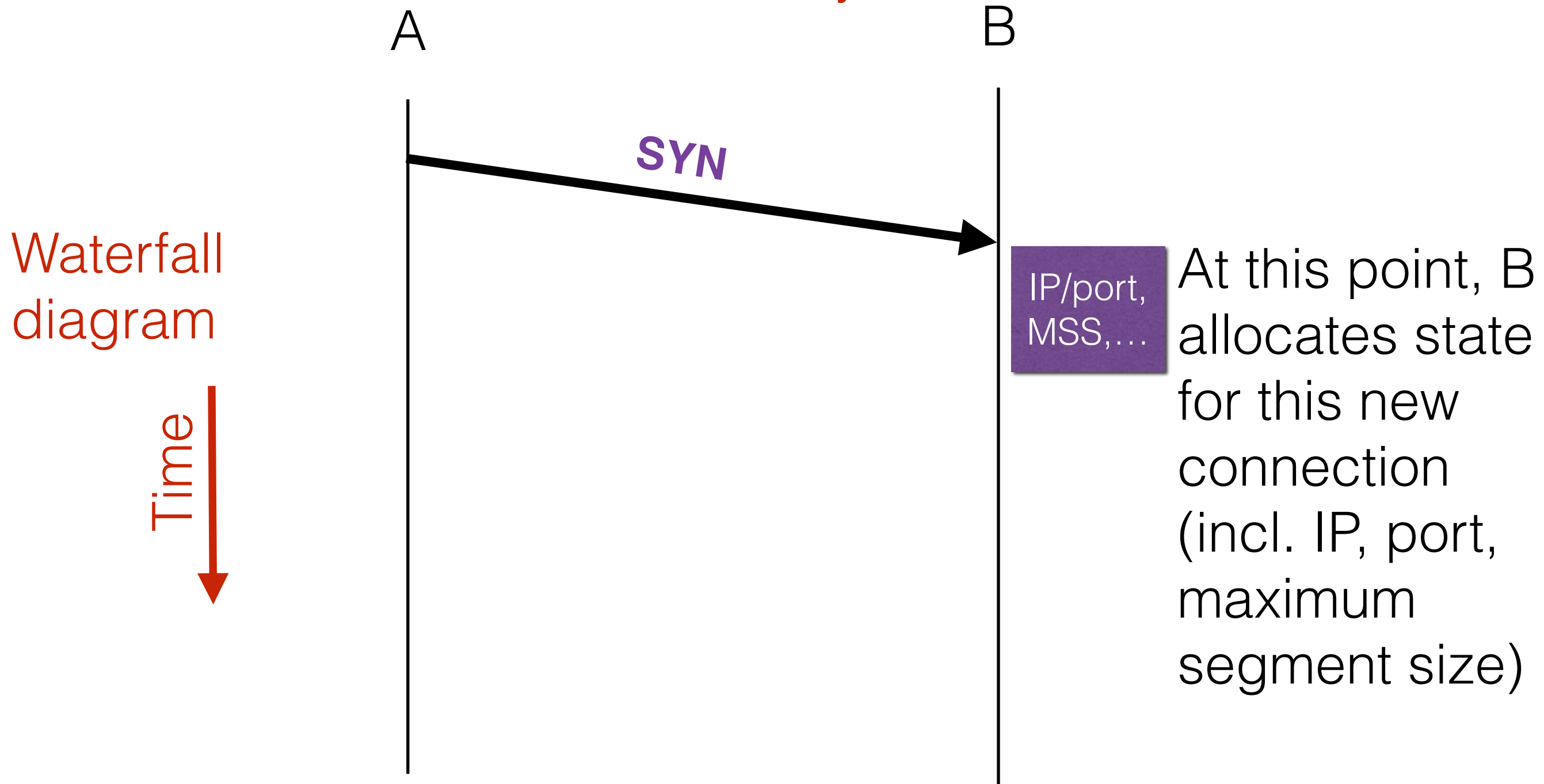
Time



At this point, B
allocates state
for this new
connection
(incl. IP, port,
maximum
segment size)

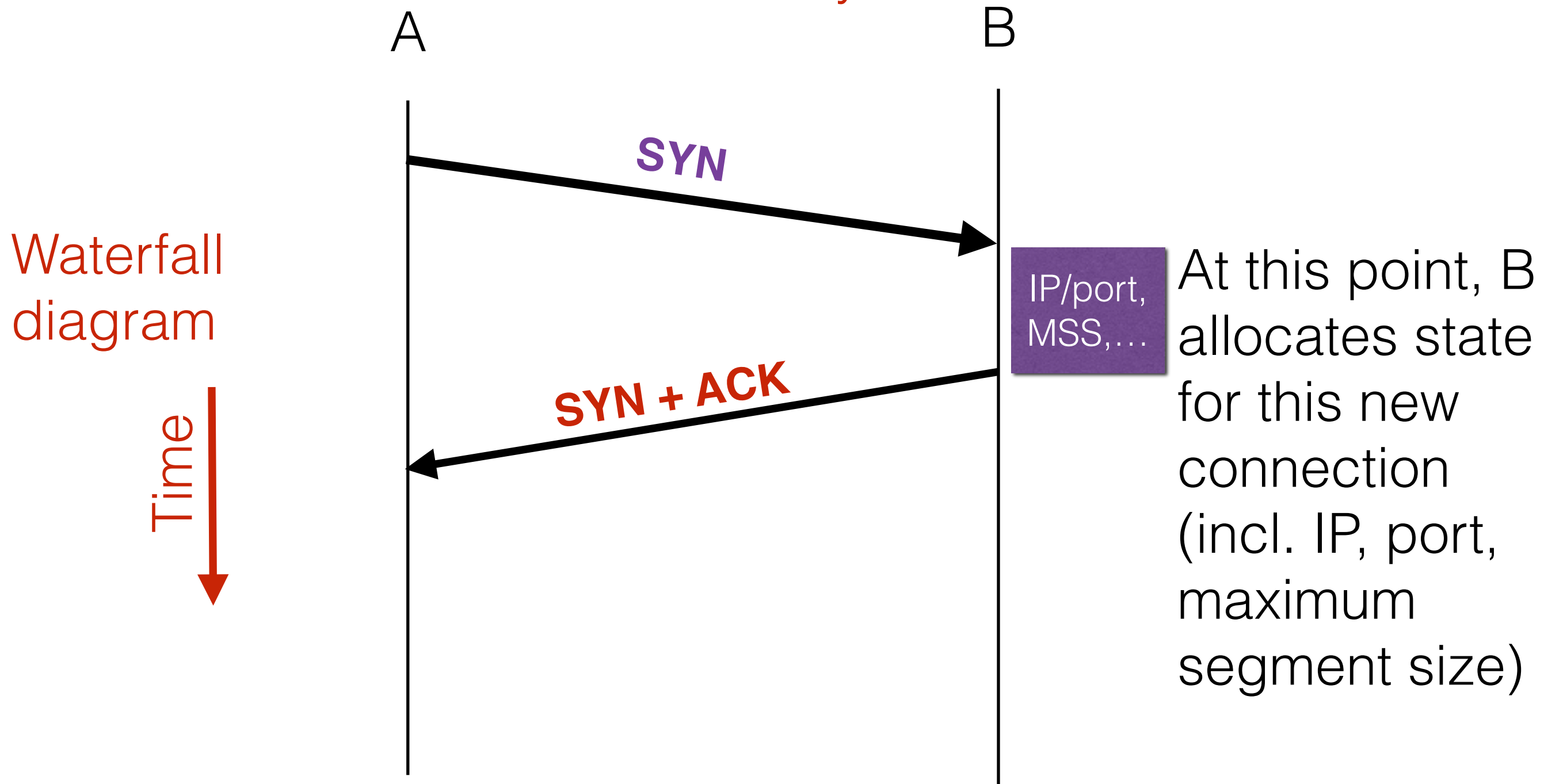
SYN flooding

Recall the three-way handshake:



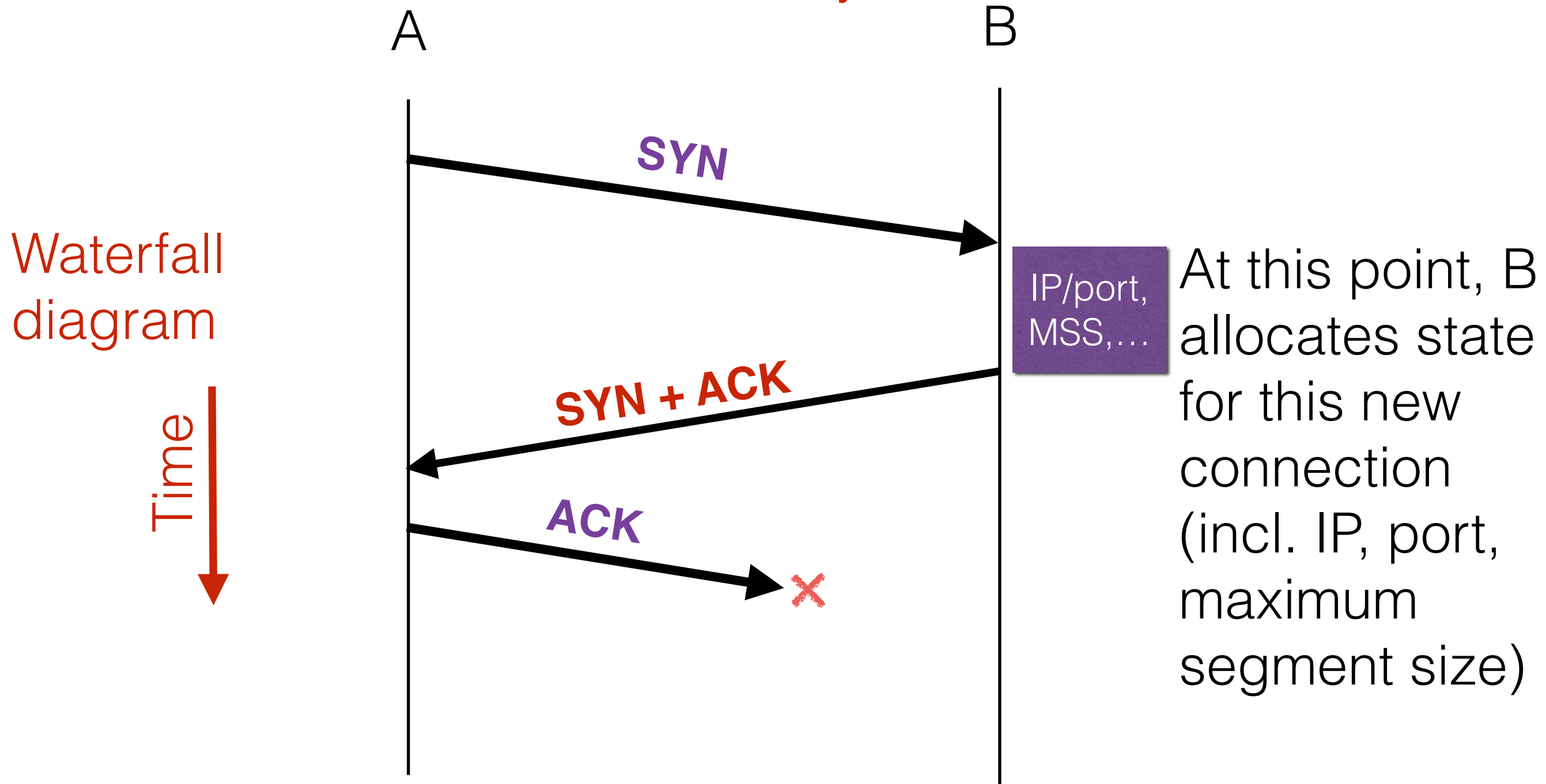
SYN flooding

Recall the three-way handshake:



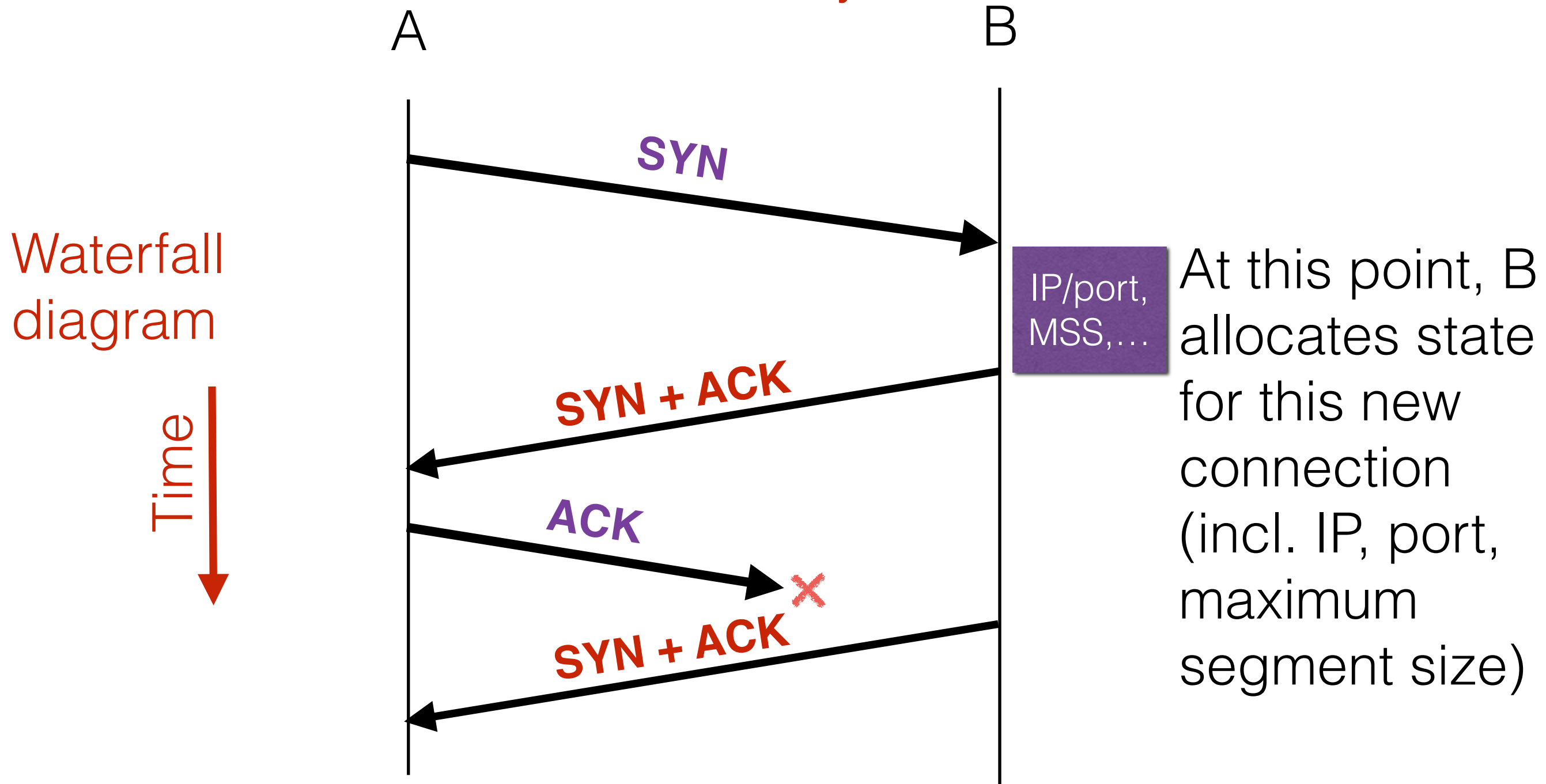
SYN flooding

Recall the three-way handshake:



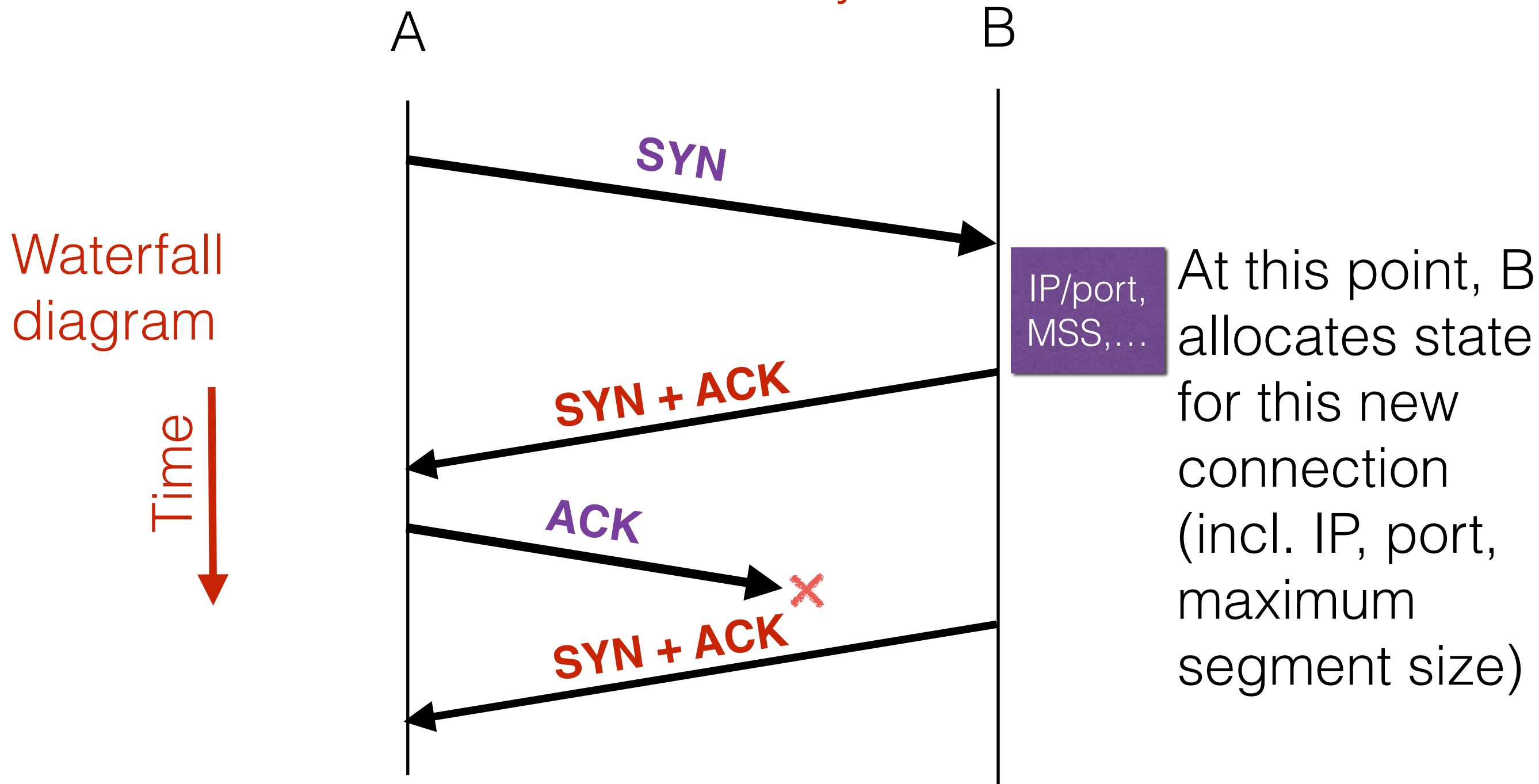
SYN flooding

Recall the three-way handshake:



SYN flooding

Recall the three-way handshake:



B will hold onto this **local state** and retransmit SYN+ACK's until it hears back or times out (up to 63 sec).

SYN flooding

The attack

A

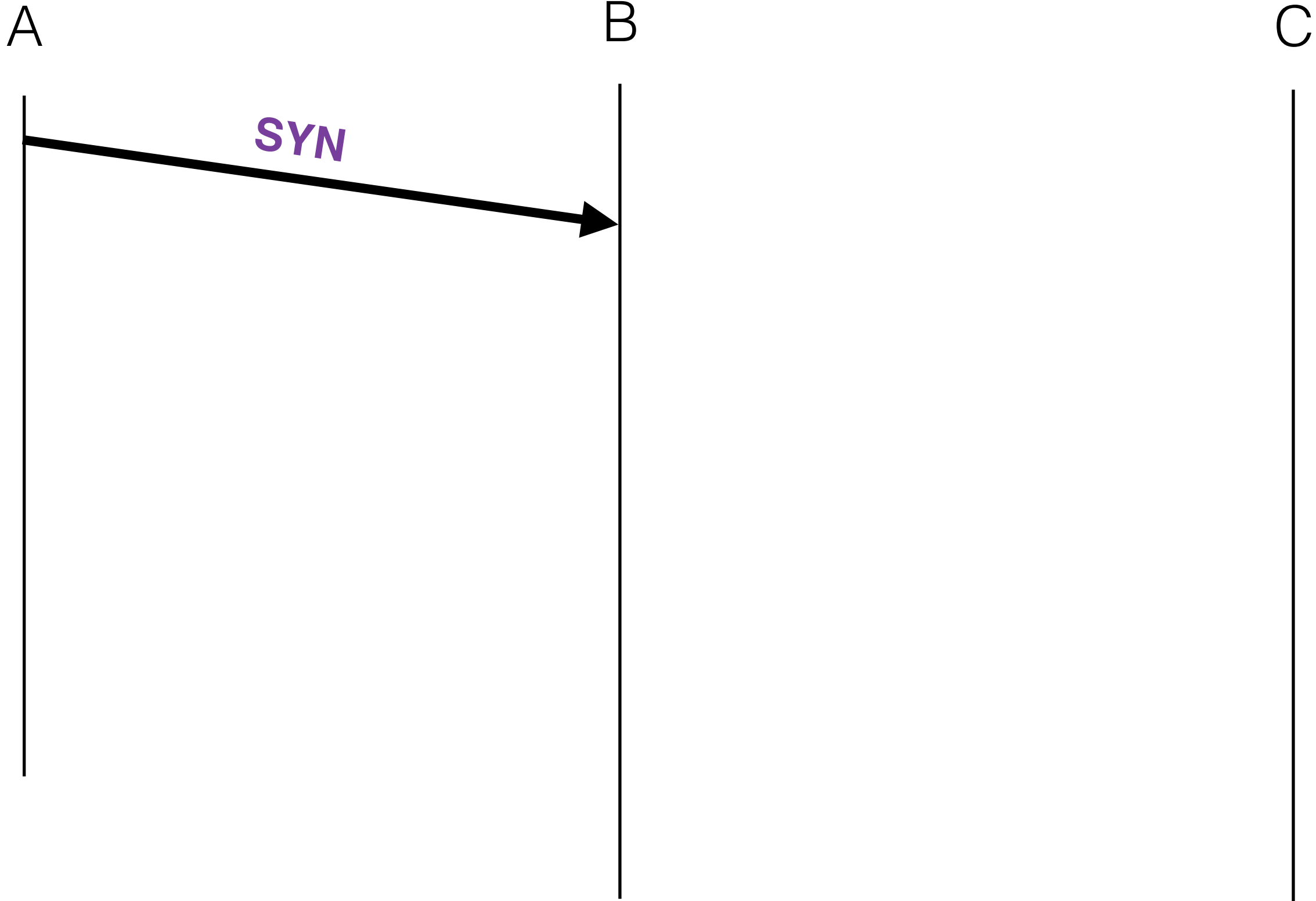
B

C



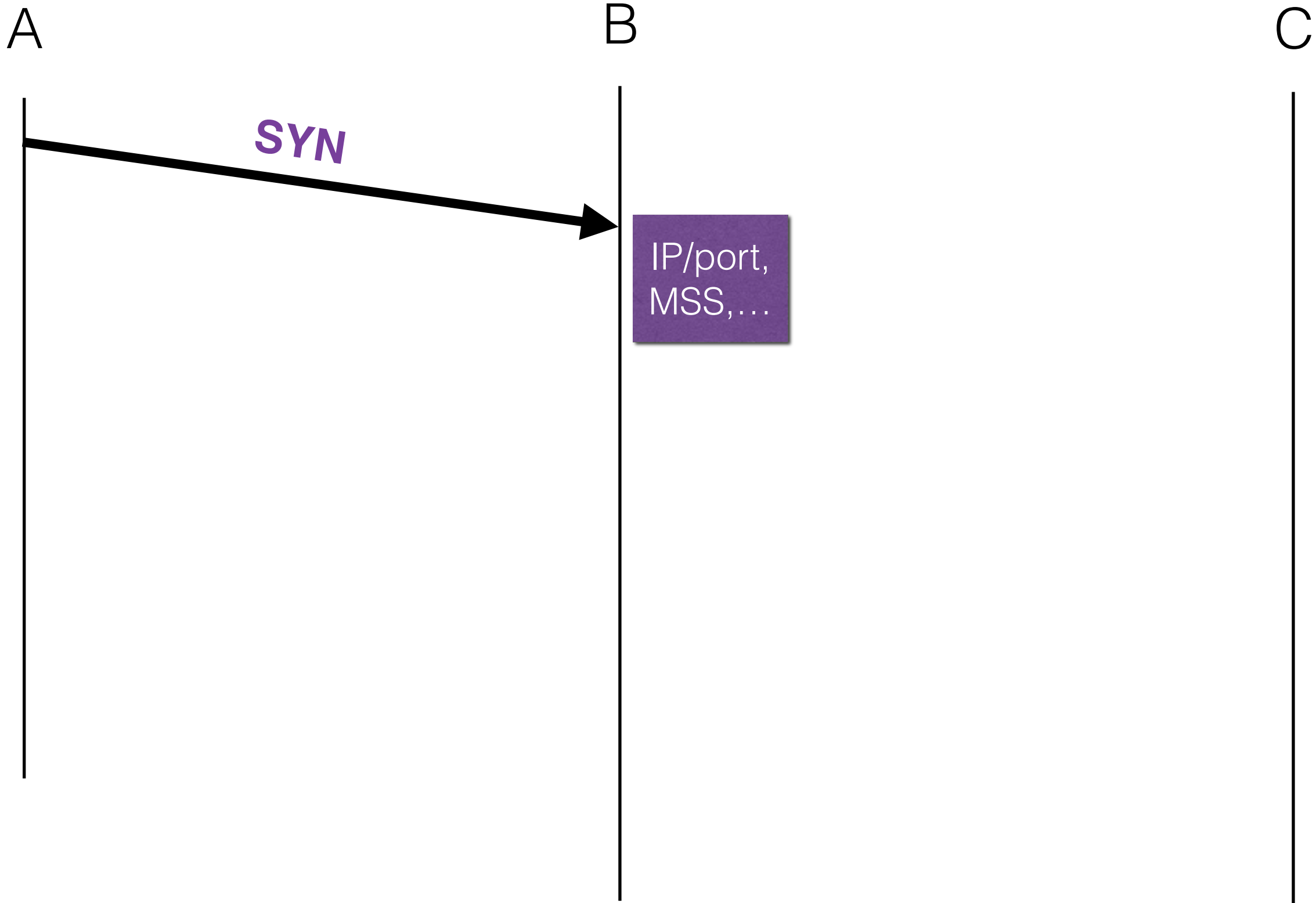
SYN flooding

The attack



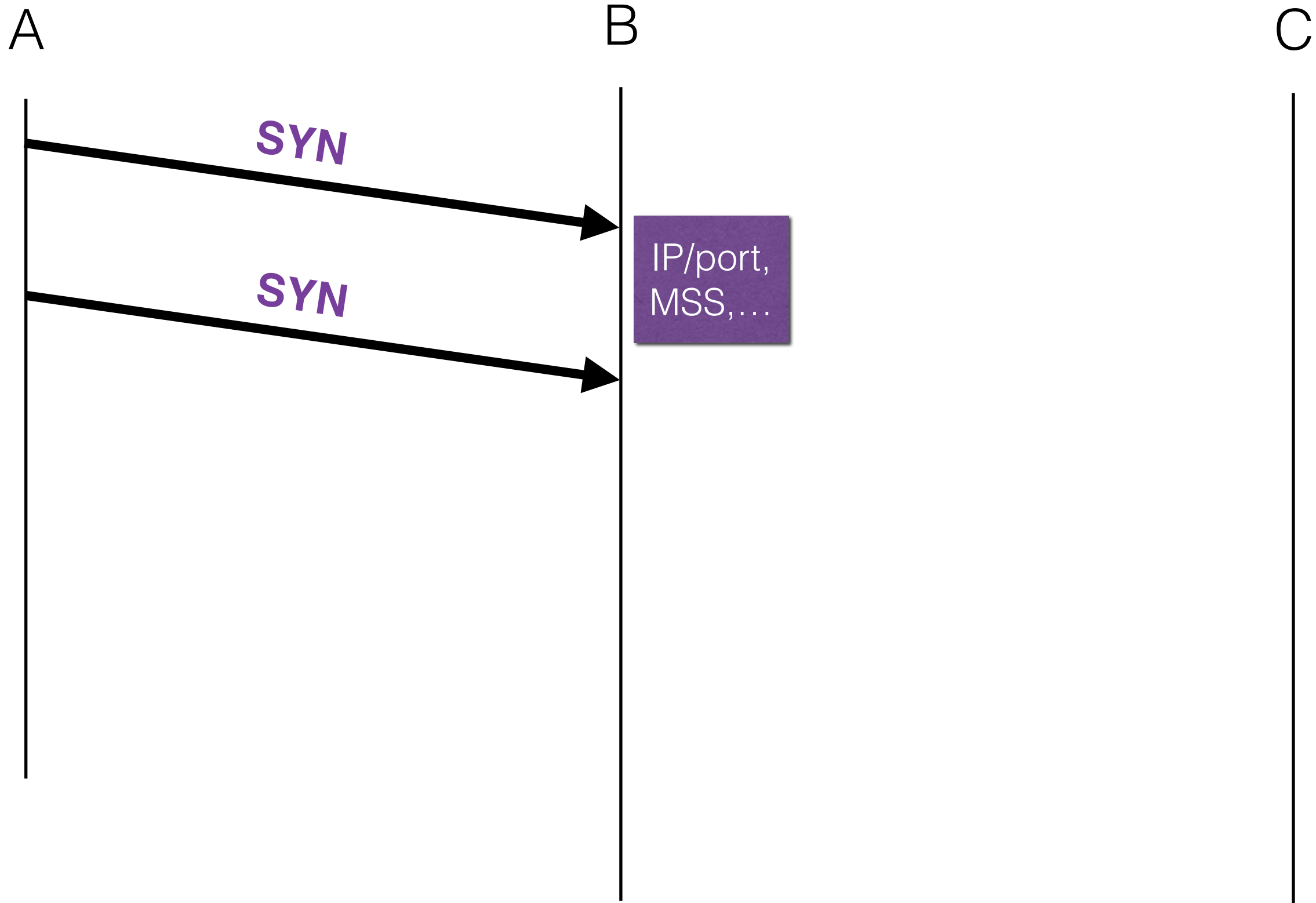
SYN flooding

The attack



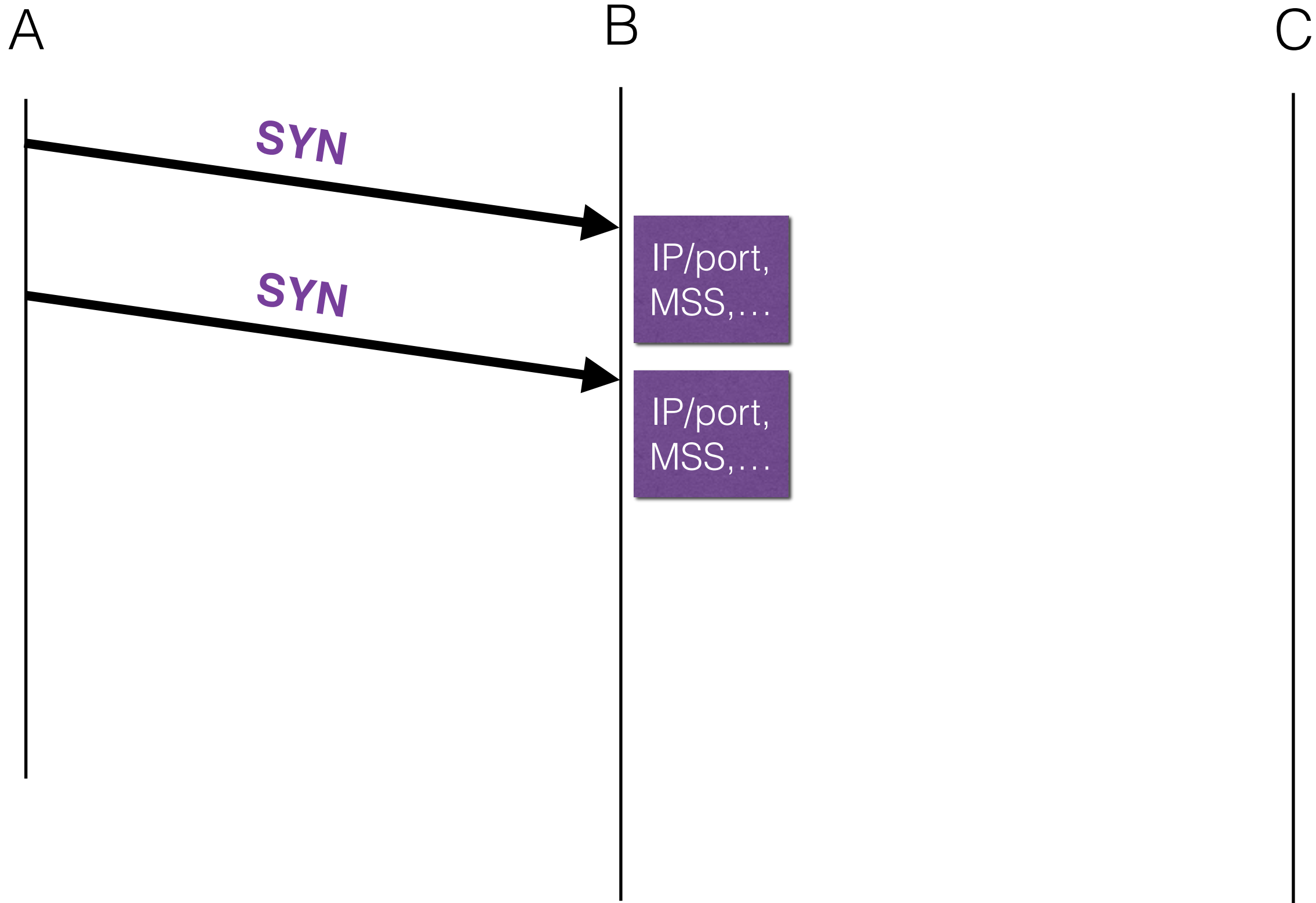
SYN flooding

The attack



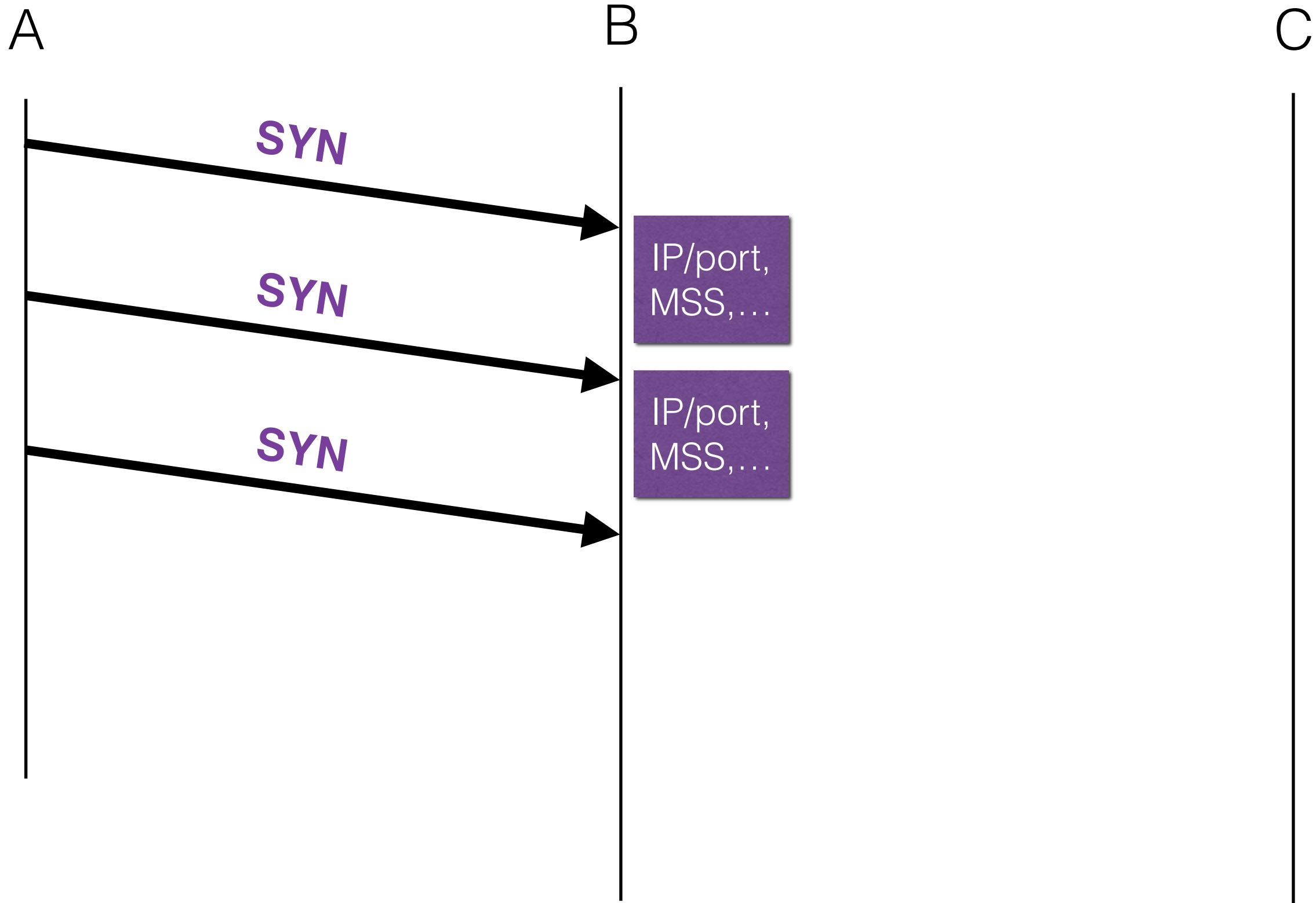
SYN flooding

The attack



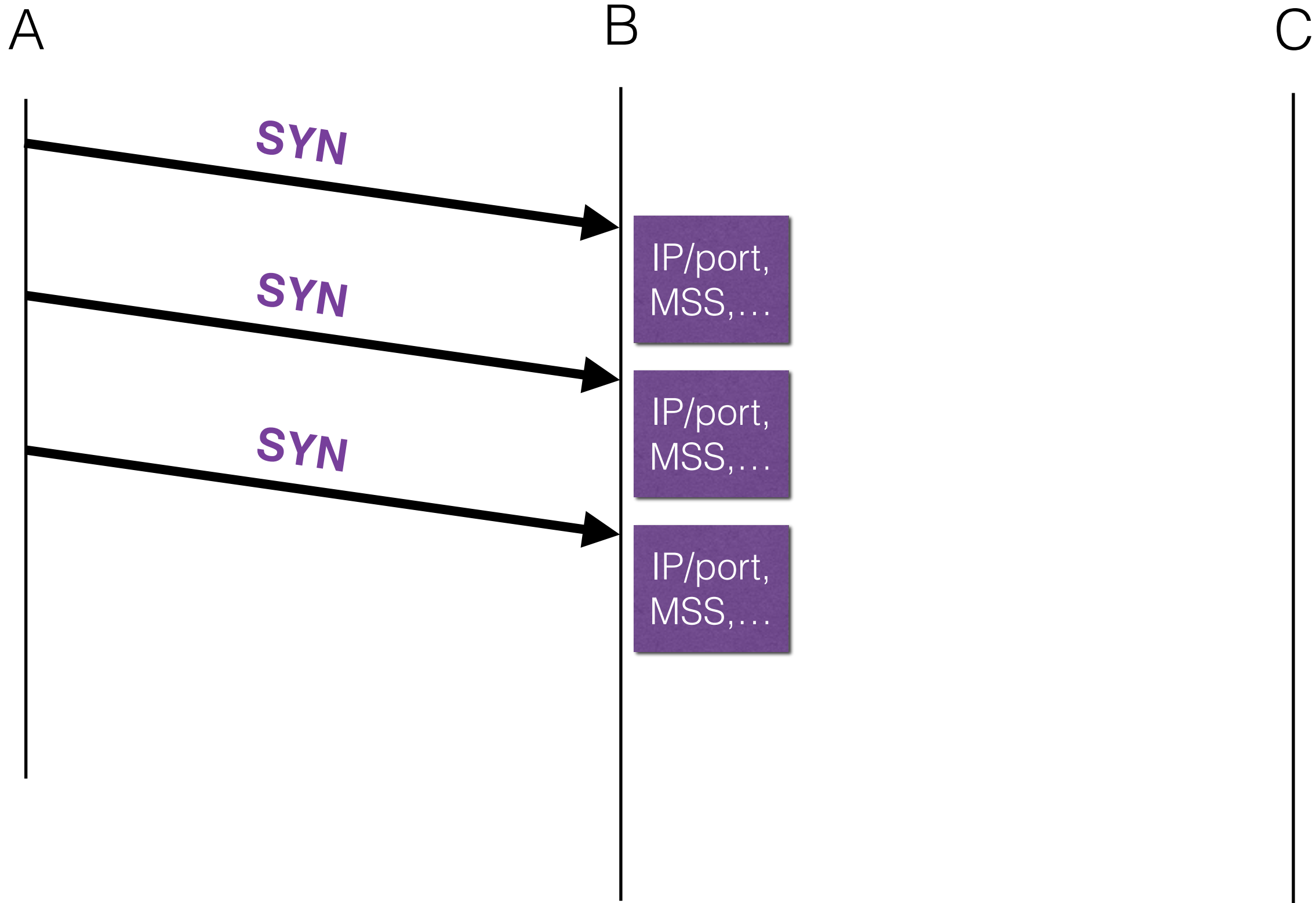
SYN flooding

The attack



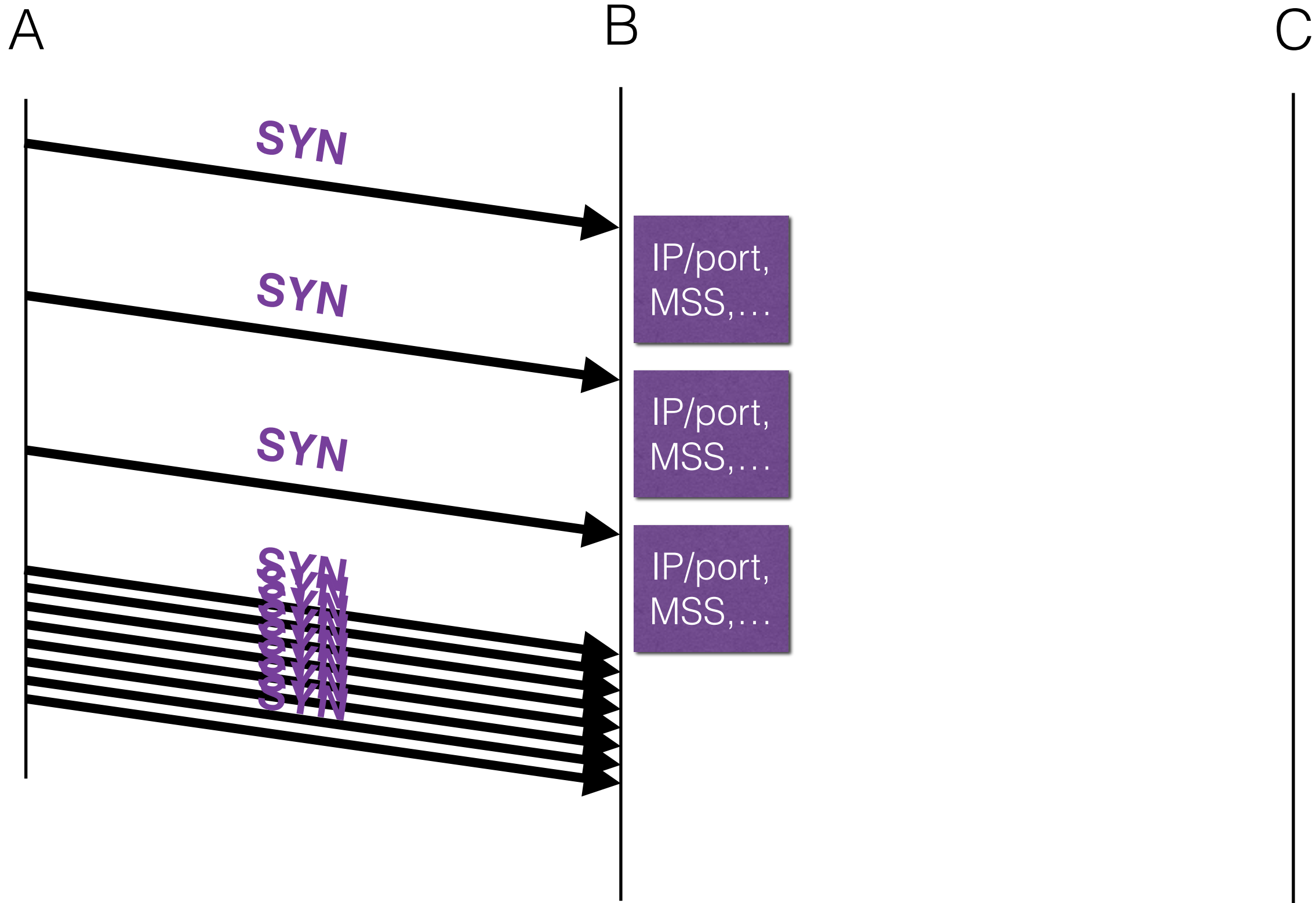
SYN flooding

The attack



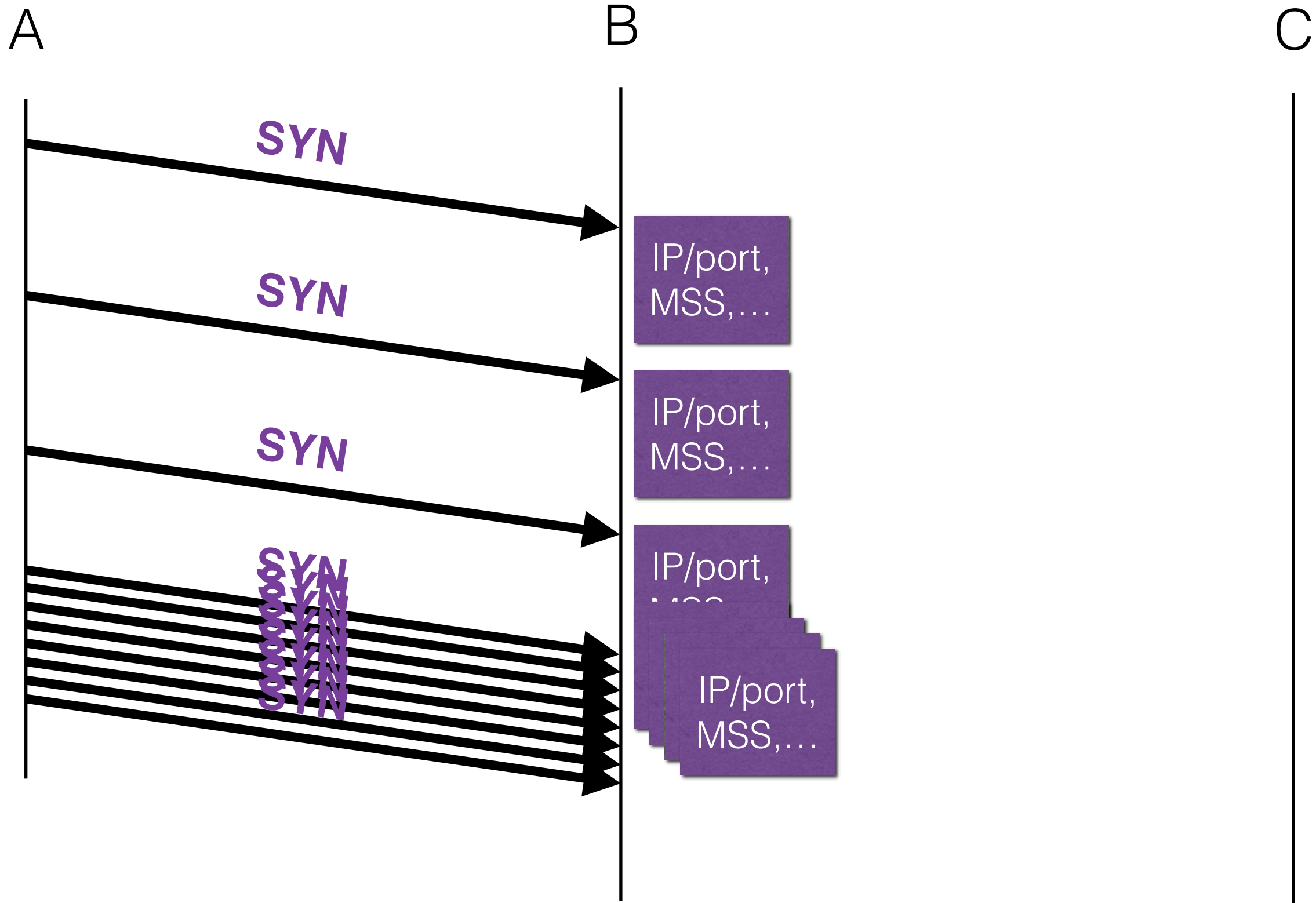
SYN flooding

The attack



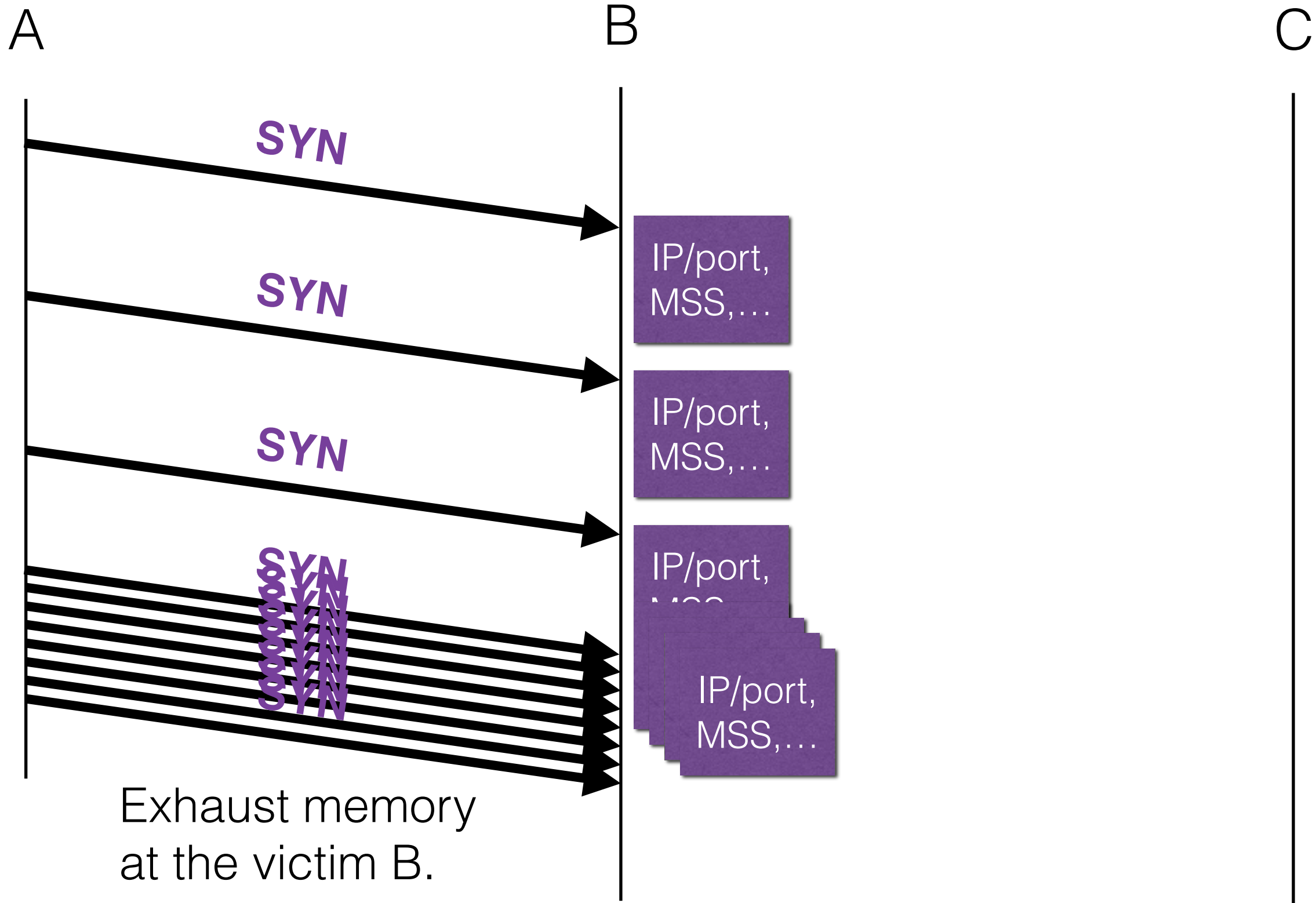
SYN flooding

The attack



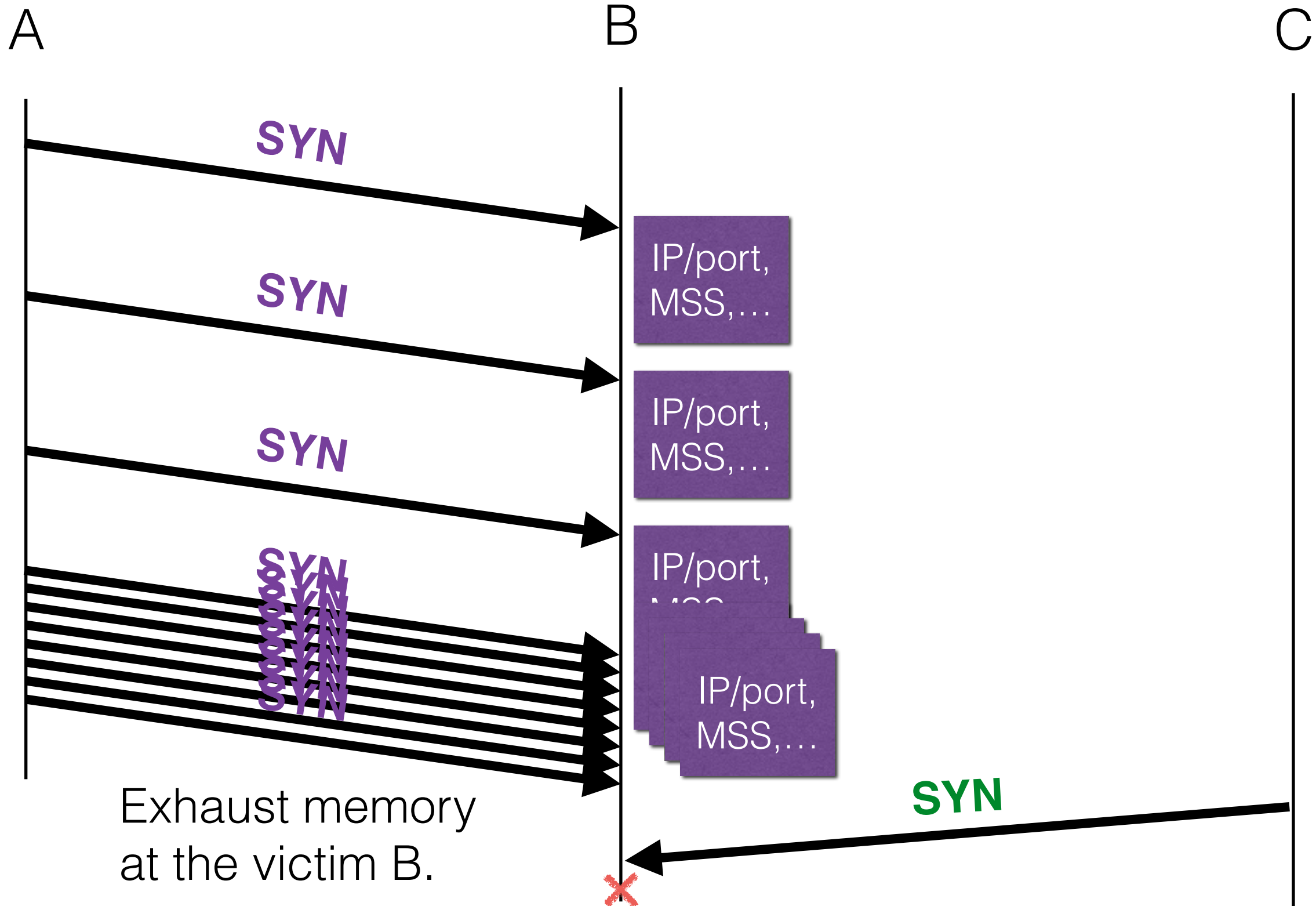
SYN flooding

The attack



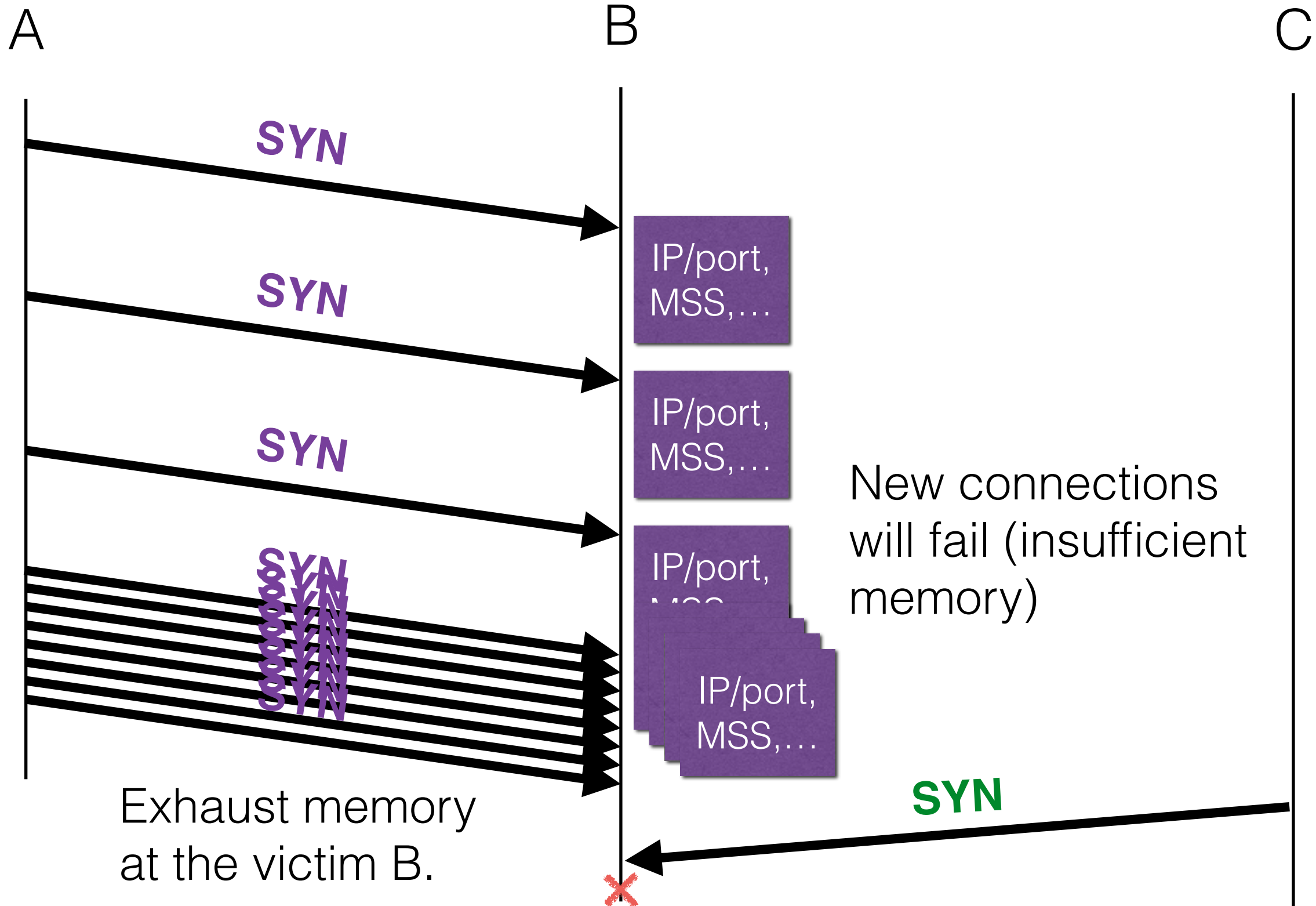
SYN flooding

The attack



SYN flooding

The attack



SYN flooding details

- Easy to detect many incomplete handshakes from a single IP address
- *Spoof* the source IP address
 - It's just a field in a header: set it to whatever you like
- Problem: the host who really owns that spoofed IP address may respond to the SYN+ACK with a RST, deleting the local state at the victim
- Ideally, spoof an IP address of a host you know won't respond

SYN cookies

The defense

A

B



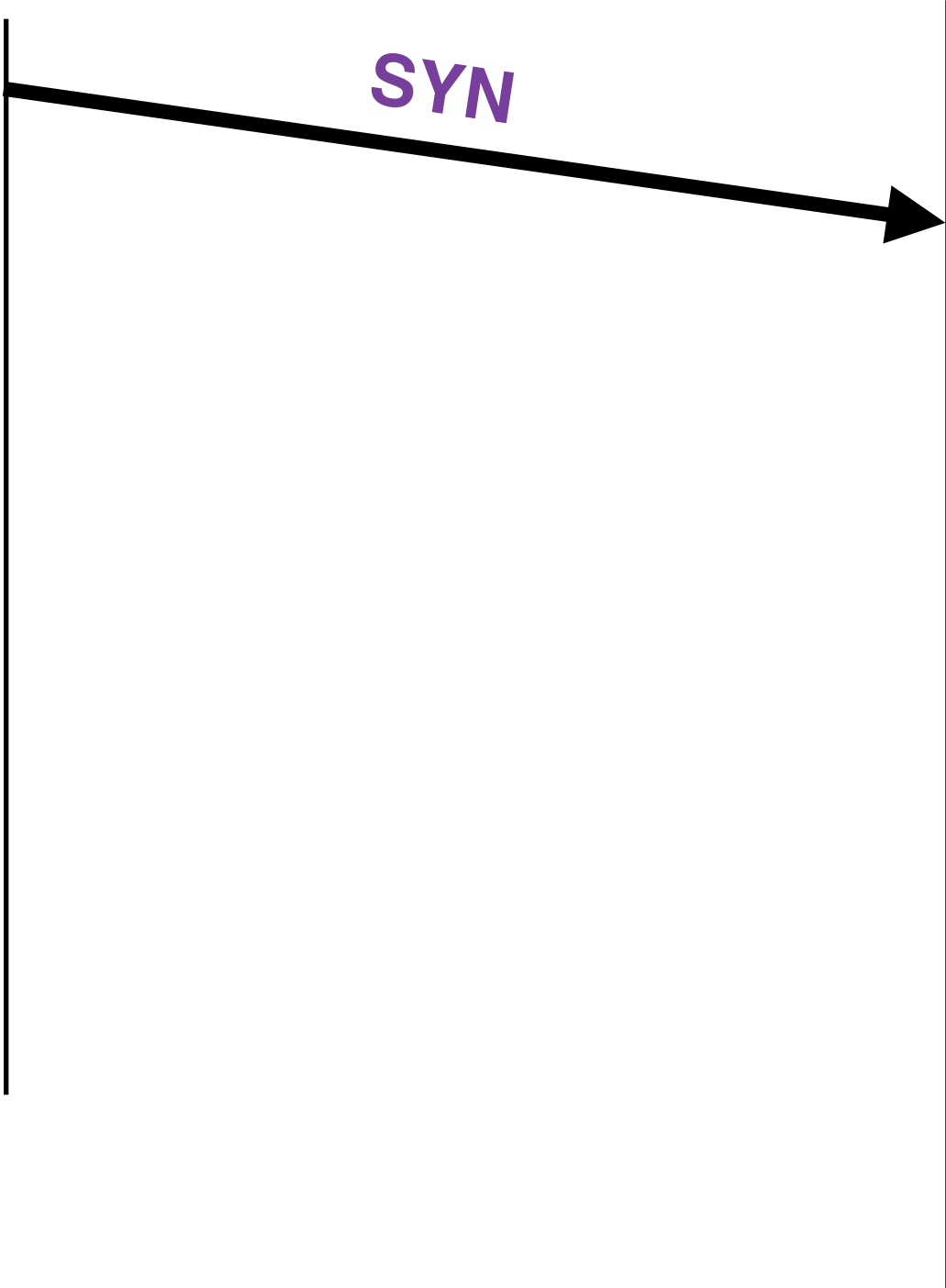
SYN cookies

The defense

A

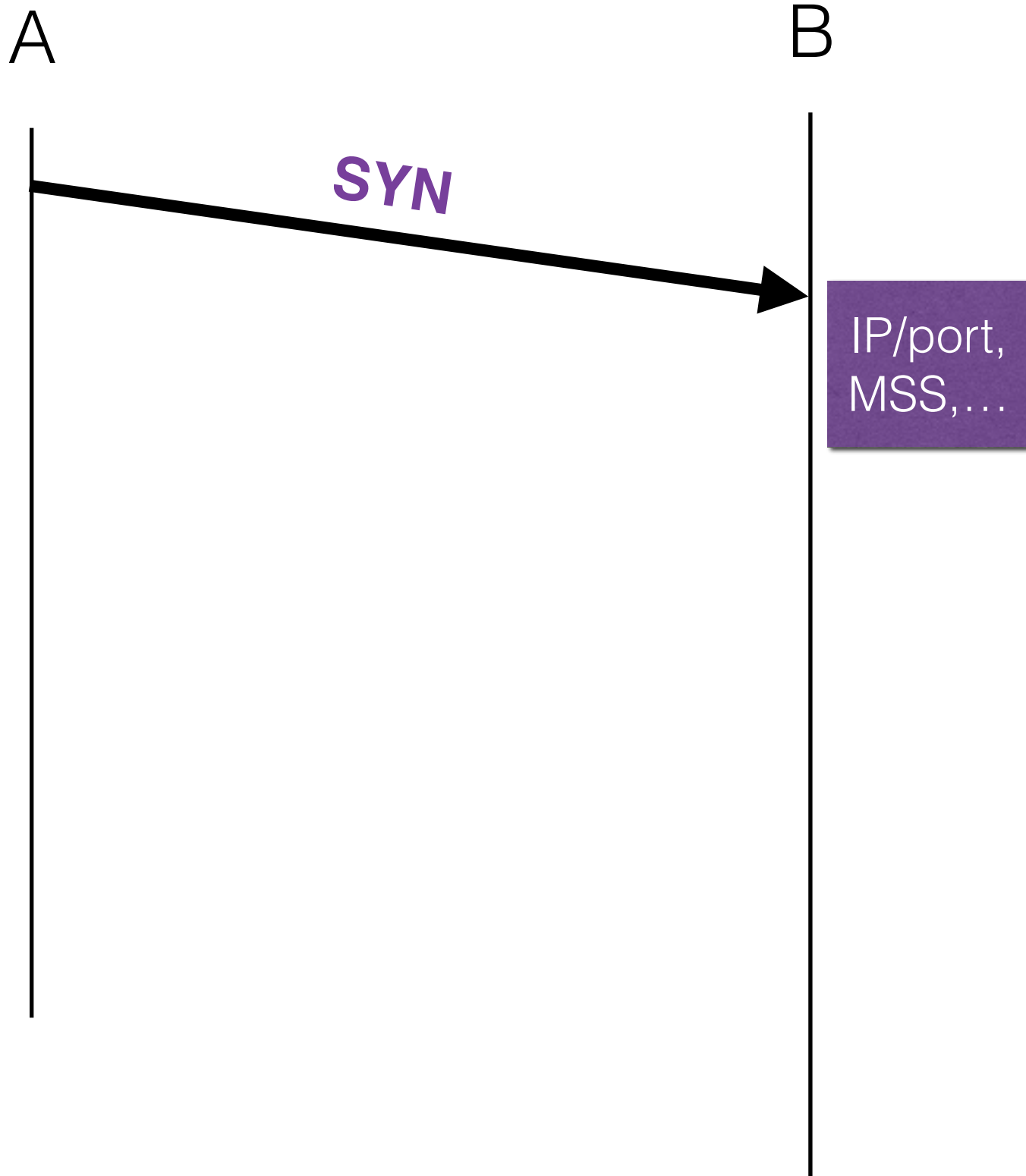
B

SYN



SYN cookies

The defense



SYN cookies

The defense

A

B

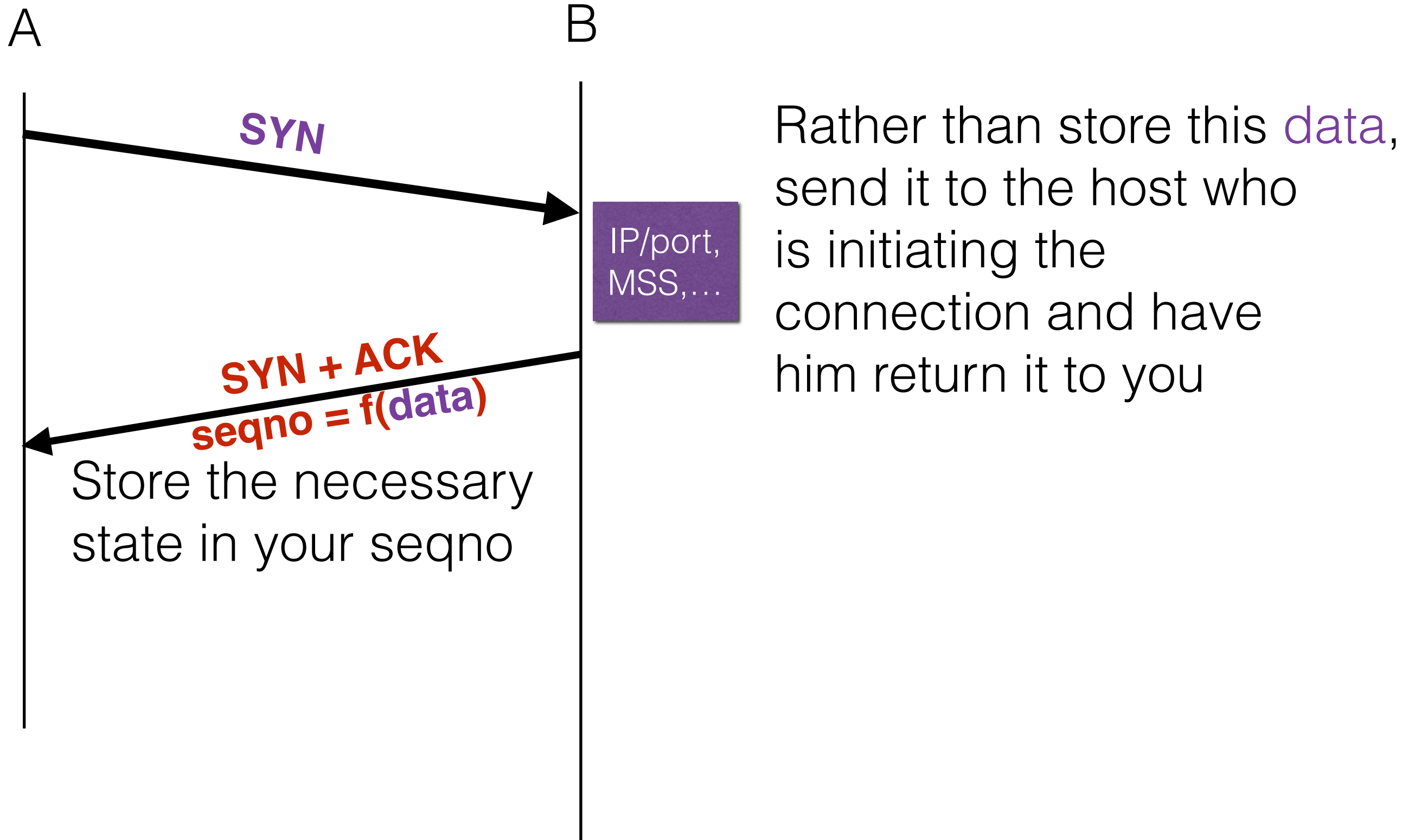
SYN

IP/port,
MSS,...

Rather than store this **data**,
send it to the host who
is initiating the
connection and have
him return it to you

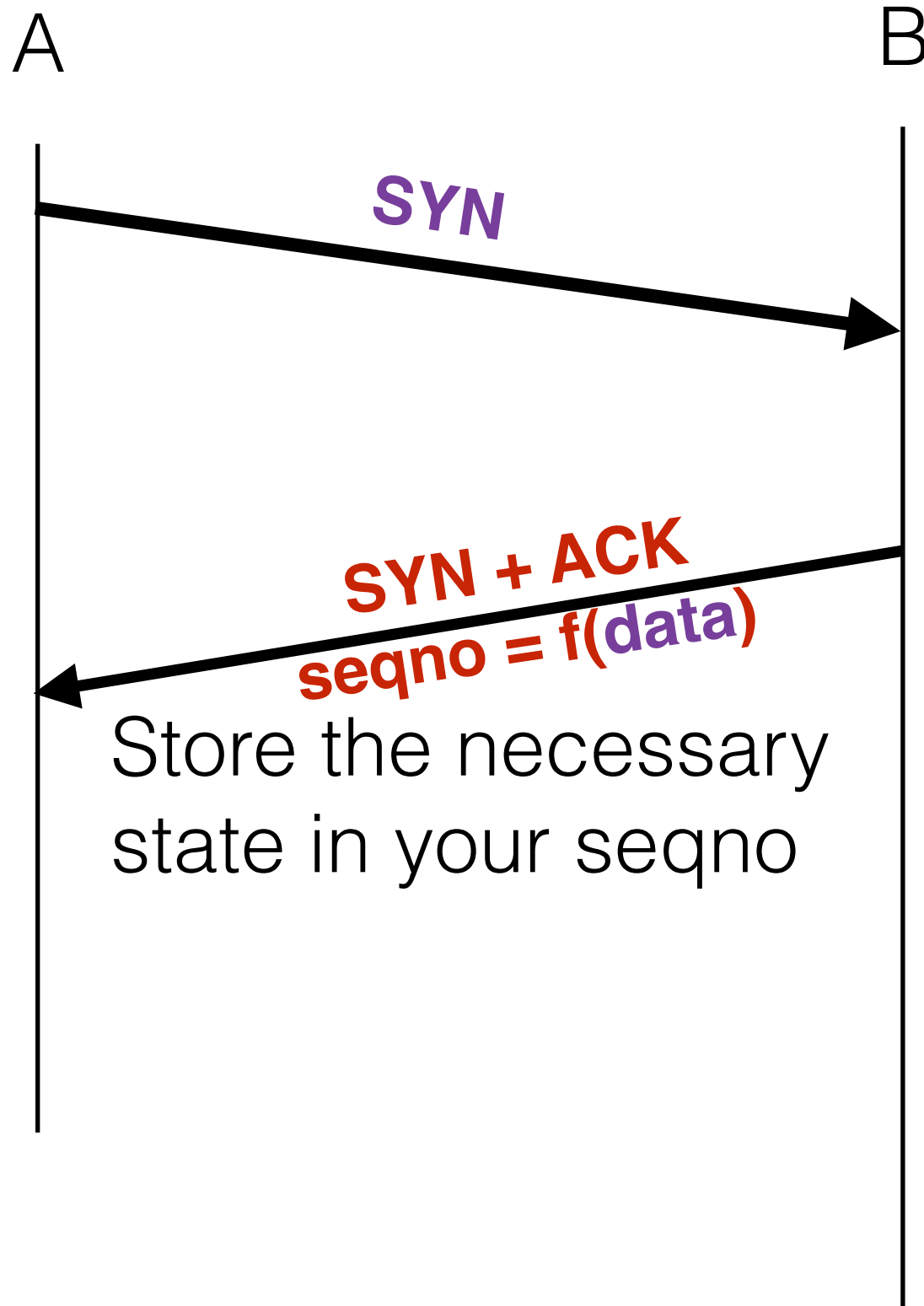
SYN cookies

The defense



SYN cookies

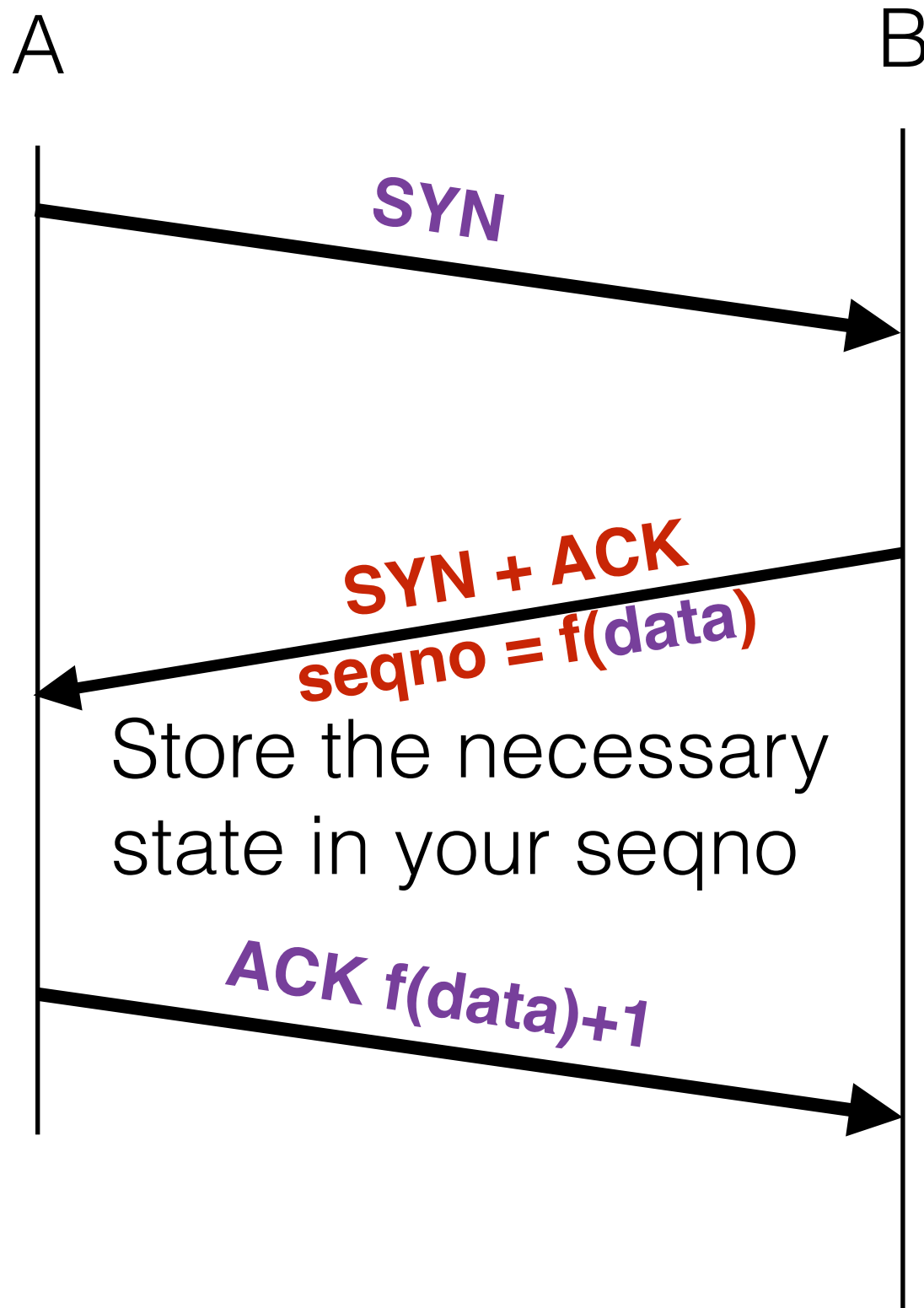
The defense



Rather than store this **data**, send it to the host who is initiating the connection and have him return it to you

SYN cookies

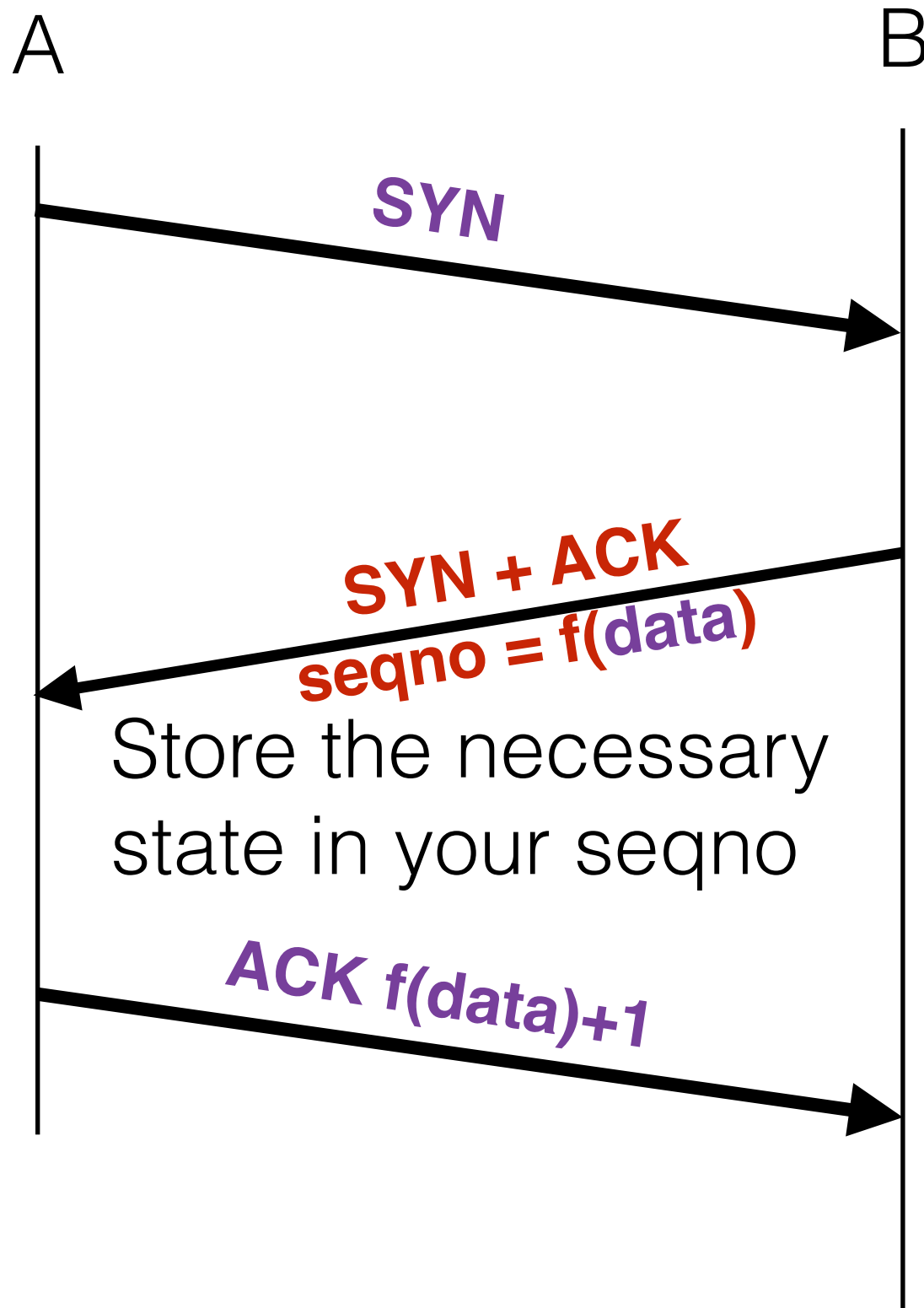
The defense



Rather than store this **data**, send it to the host who is initiating the connection and have him return it to you

SYN cookies

The defense

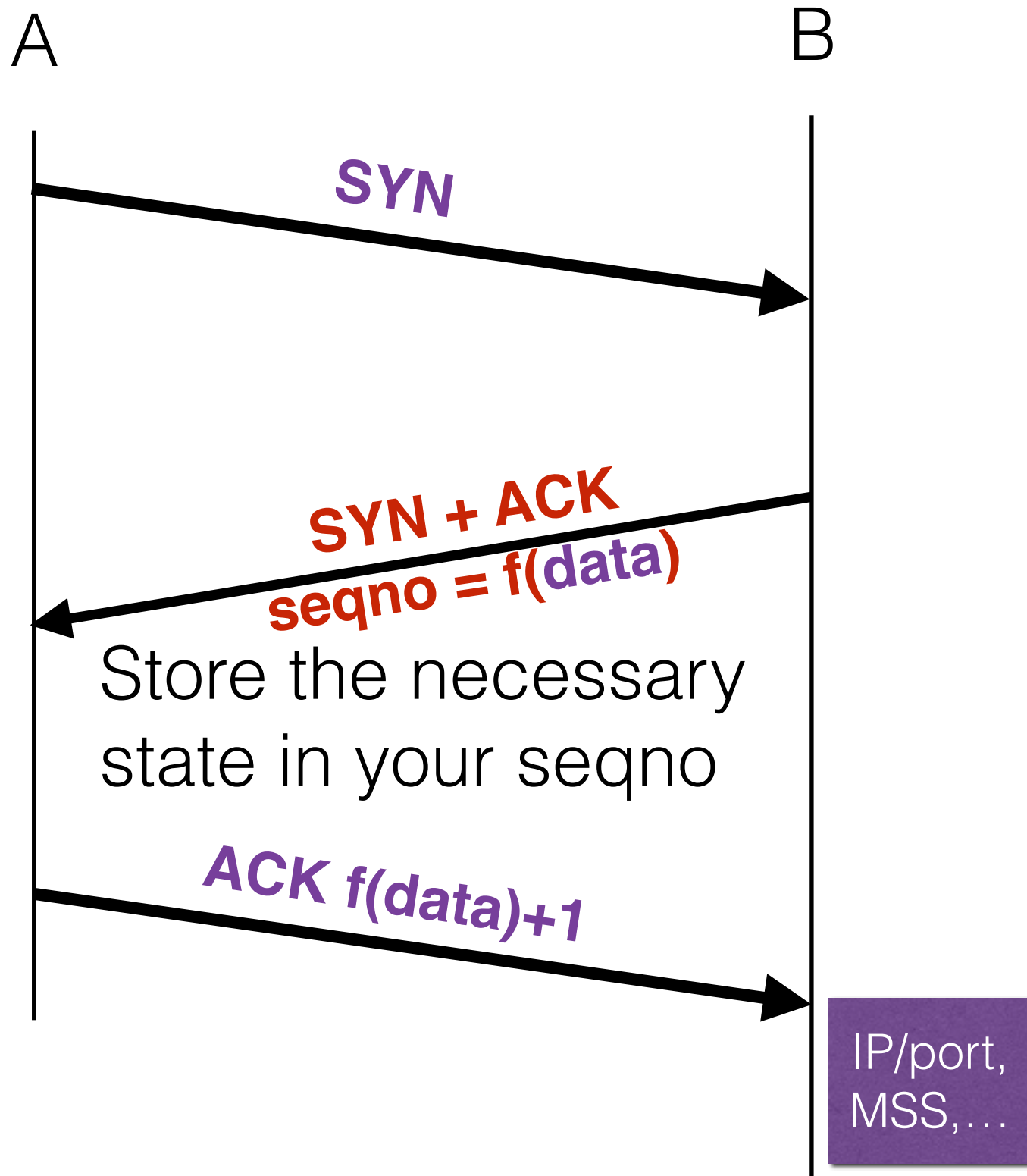


Rather than store this **data**, send it to the host who is initiating the connection and have him return it to you

Check that $f(\text{data})$ is valid for this connection. Only at that point do you allocate state.

SYN cookies

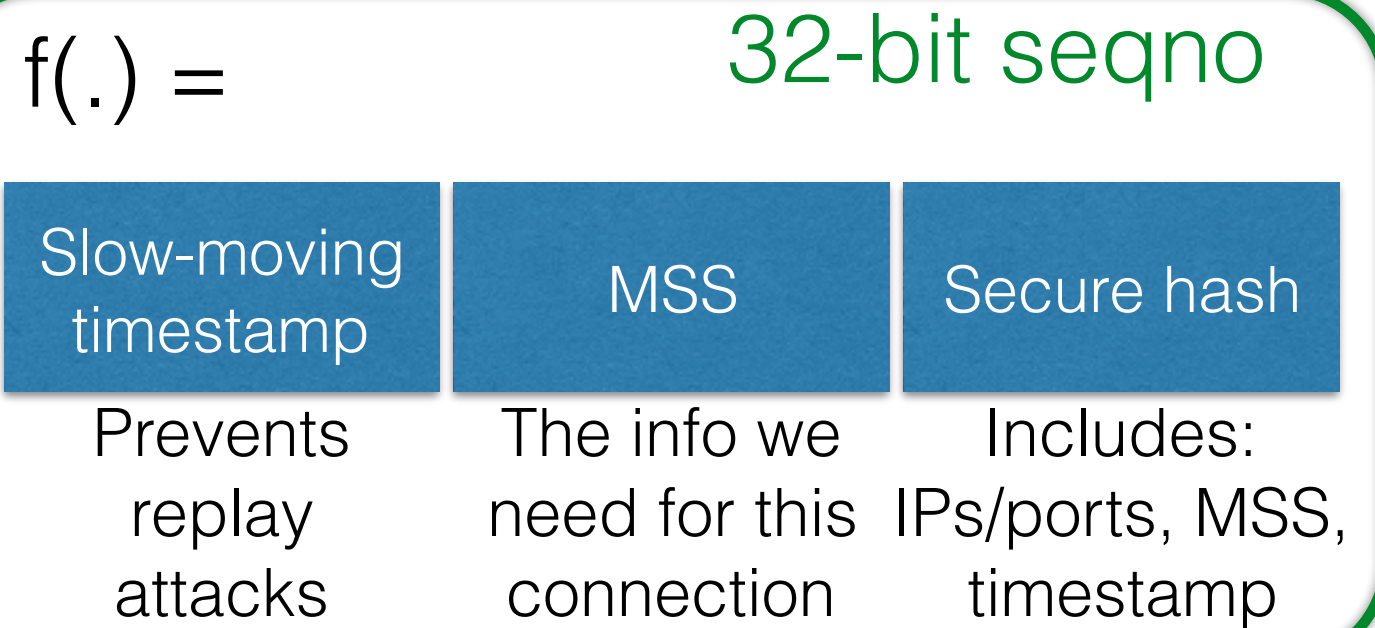
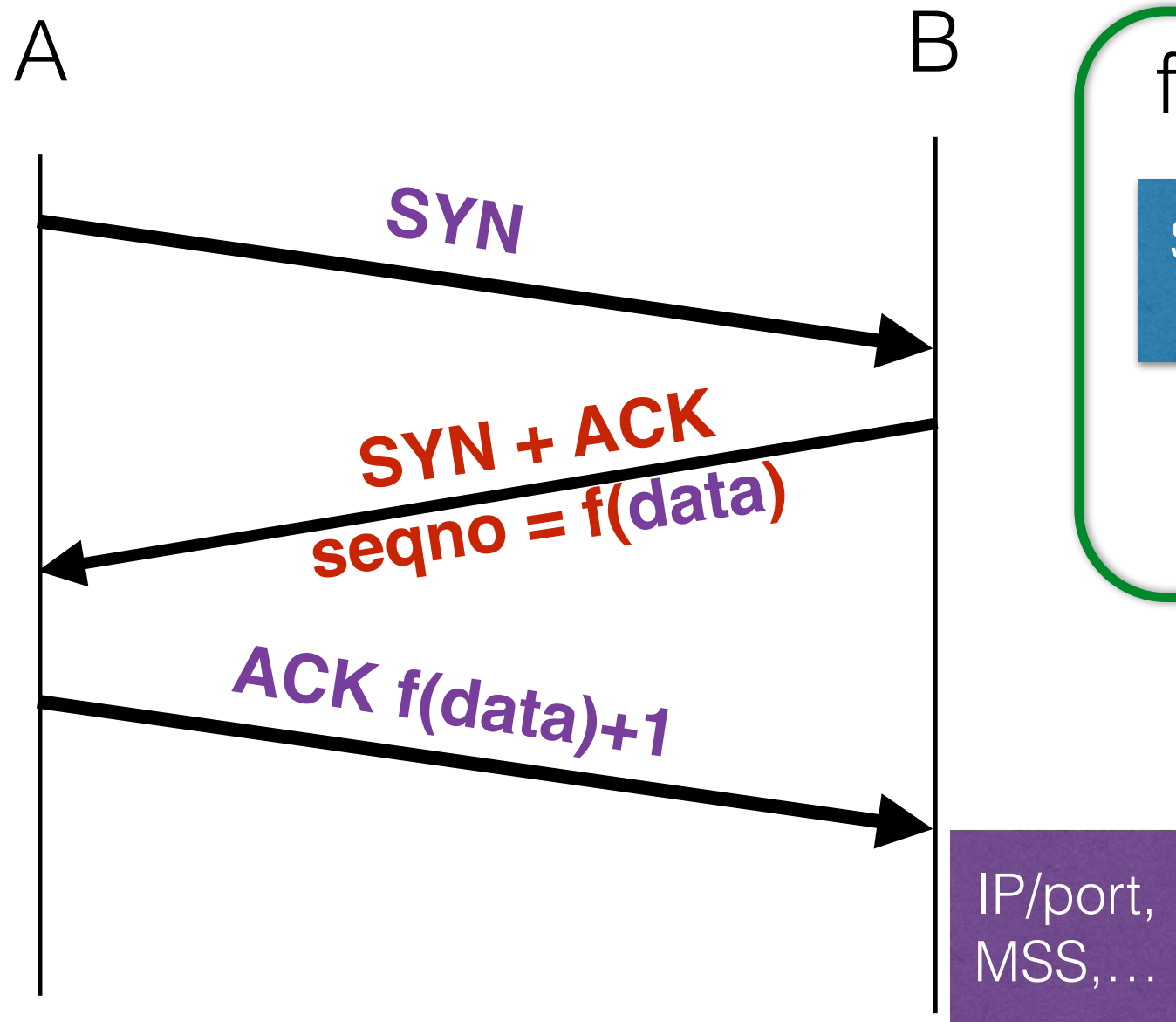
The defense



Rather than store this **data**, send it to the host who is initiating the connection and have him return it to you

Check that $f(\text{data})$ is valid for this connection. Only at that point do you allocate state.

SYN cookie format



The secure hash makes it difficult for the attacker to guess what $f()$ will be, and therefore the attacker cannot guess a correct ACK if he spoofs.

Injection attacks

- Suppose you are on the path between src and dst; what can you do?
 - Trivial to inject packets with the correct sequence number
- What if you are not on the path?
 - Need to guess the sequence number
 - Is this difficult to do?

Initial sequence numbers

- Initial sequence numbers used to be deterministic
- What havoc can we wreak?
 - Send RSTs
 - Inject data packets into an existing connection (TCP veto attacks)
 - *Initiate and use an entire connection without ever hearing the other end*

Mitnick attack

X-terminal
server

Server that X-
term trusts

Any connection initiated
from this IP address is
allowed access to the
X-terminal server

Attacker

Mitnick attack

X-terminal
server

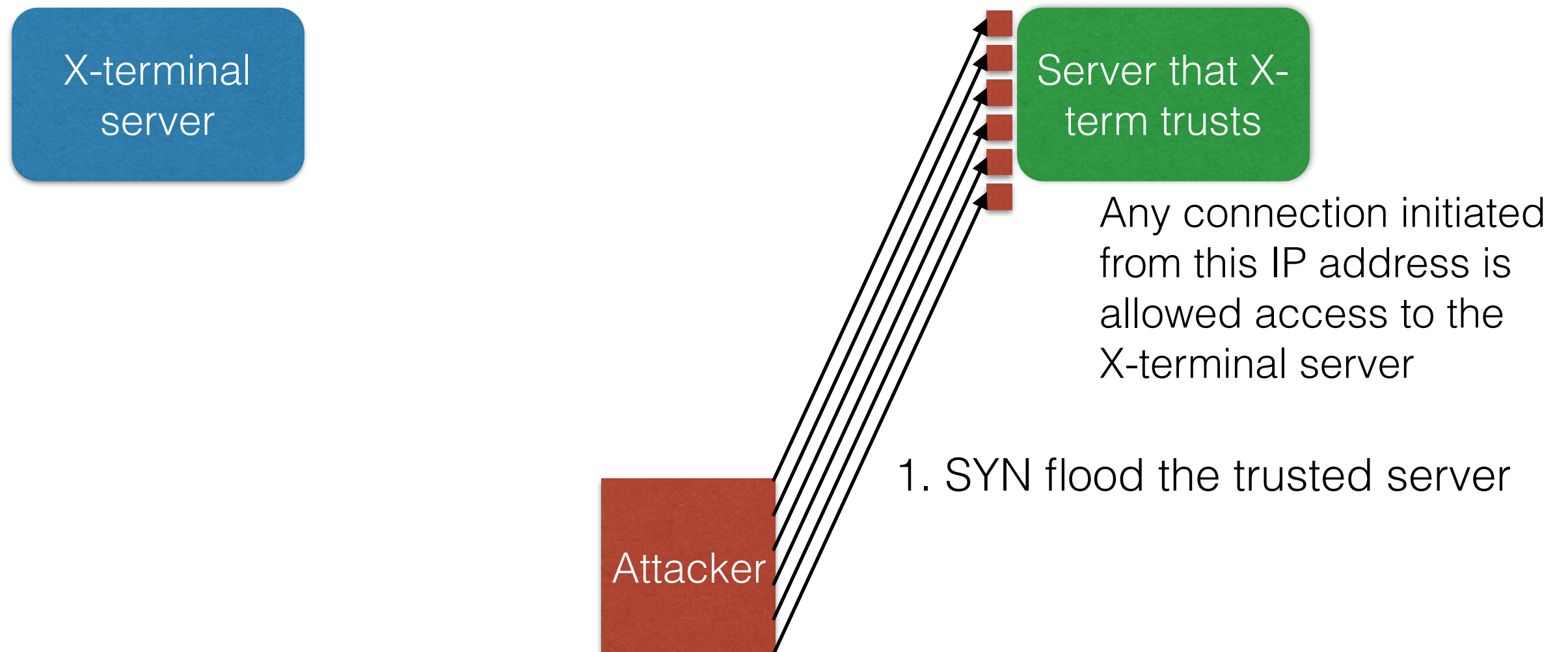
Server that X-
term trusts

Any connection initiated
from this IP address is
allowed access to the
X-terminal server

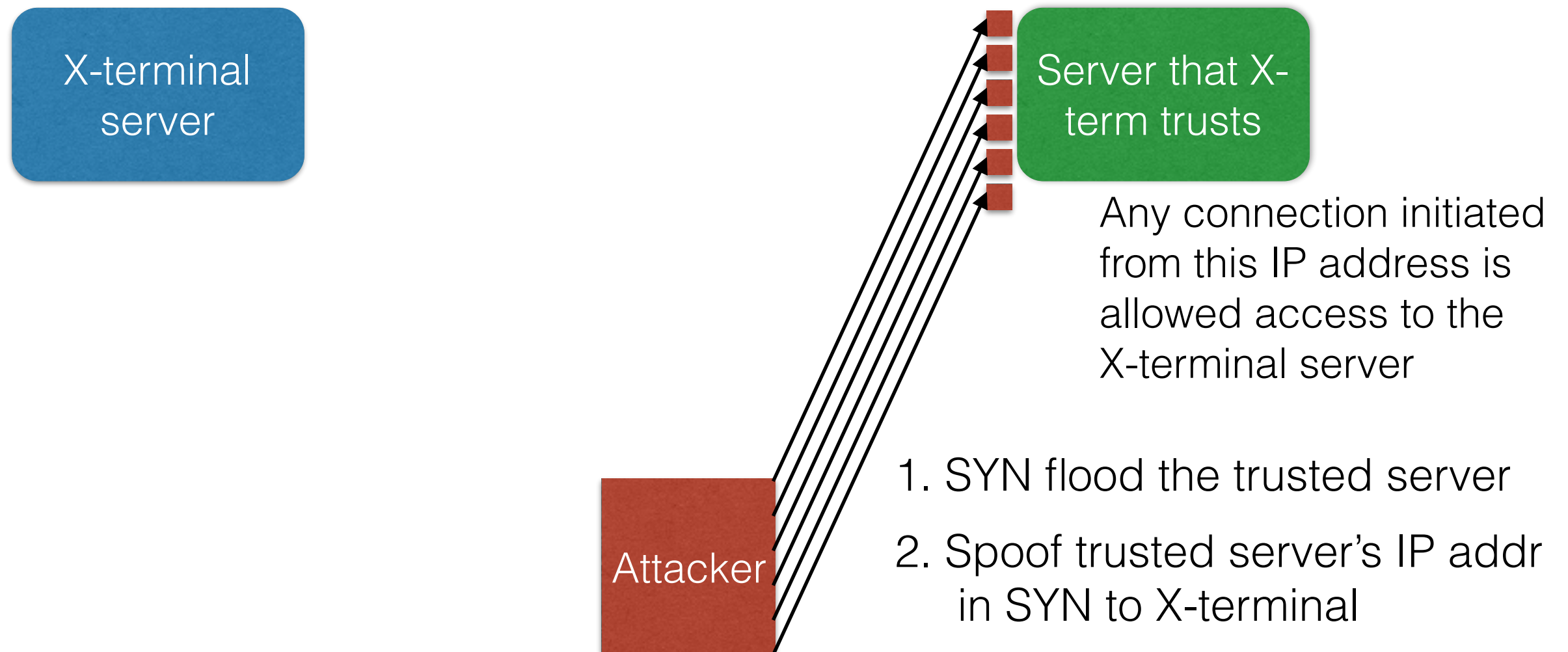
Attacker

1. SYN flood the trusted server

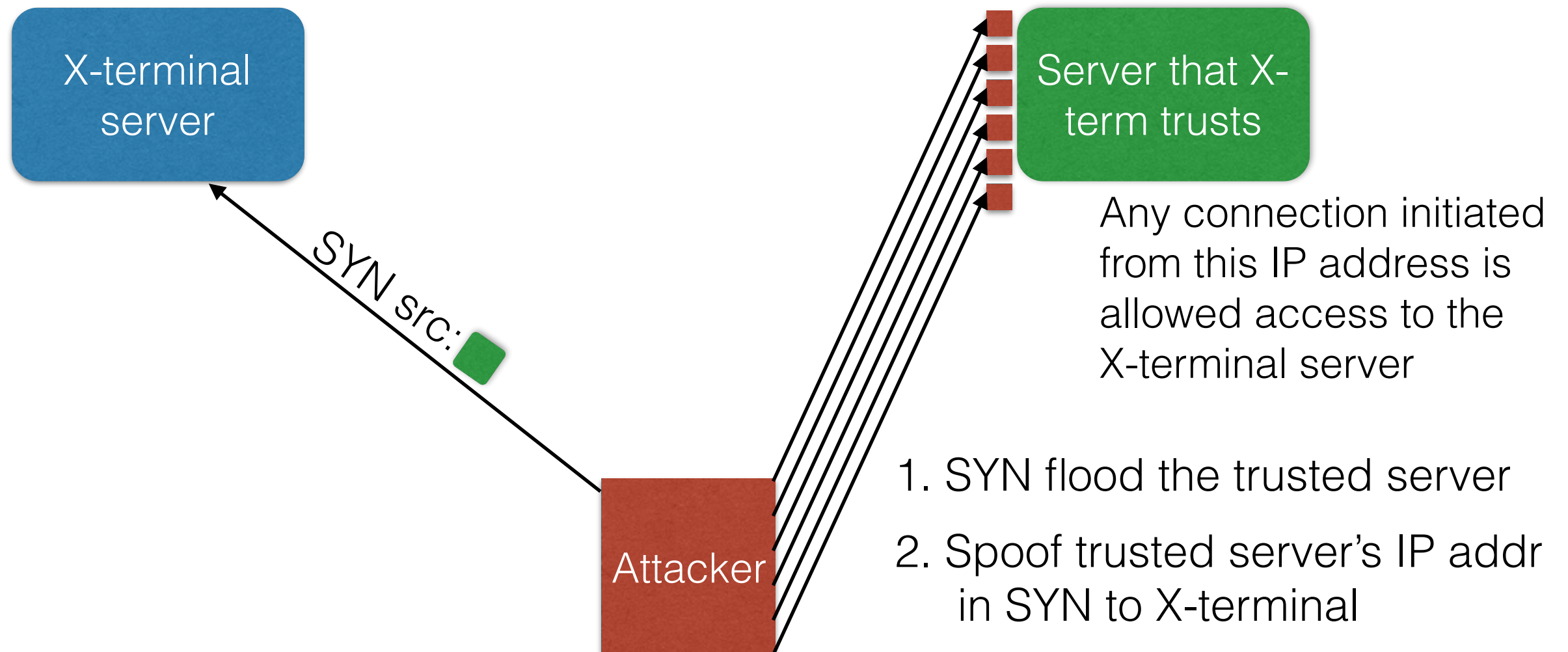
Mitnick attack



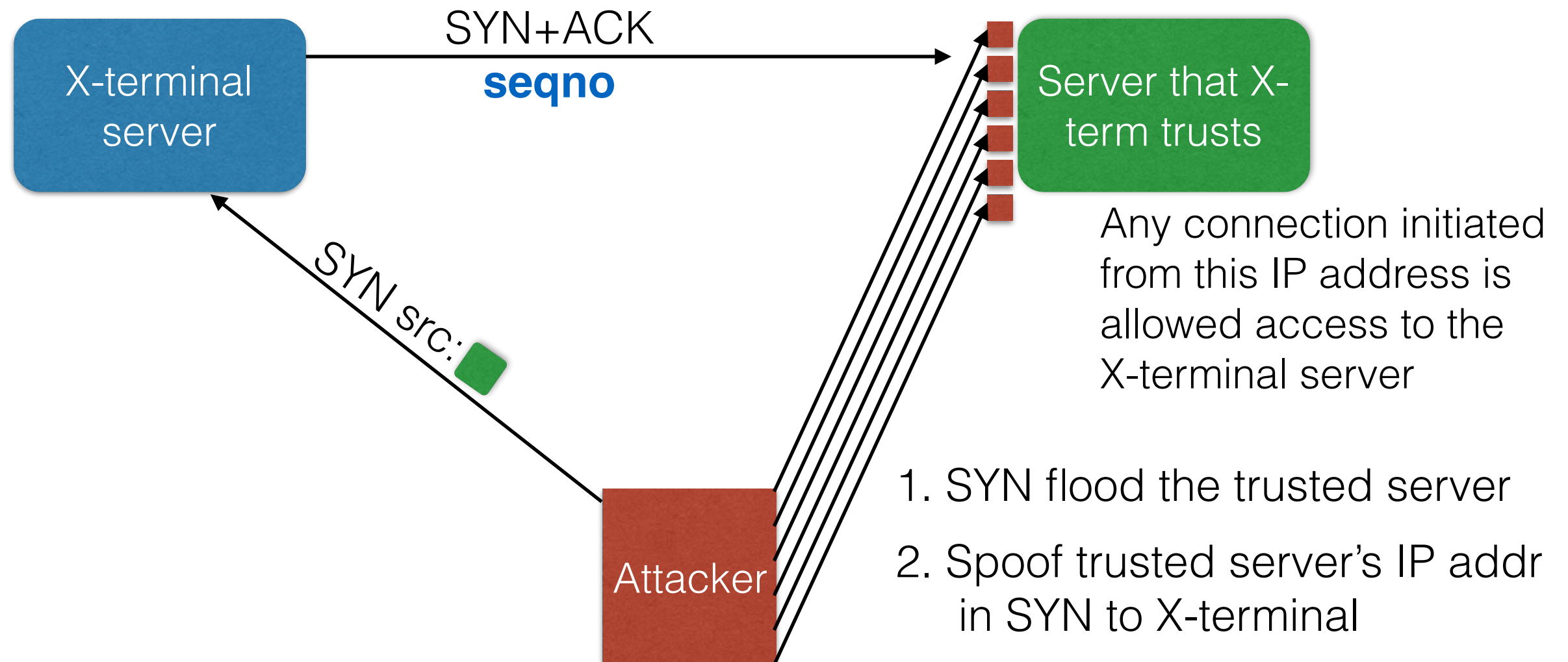
Mitnick attack



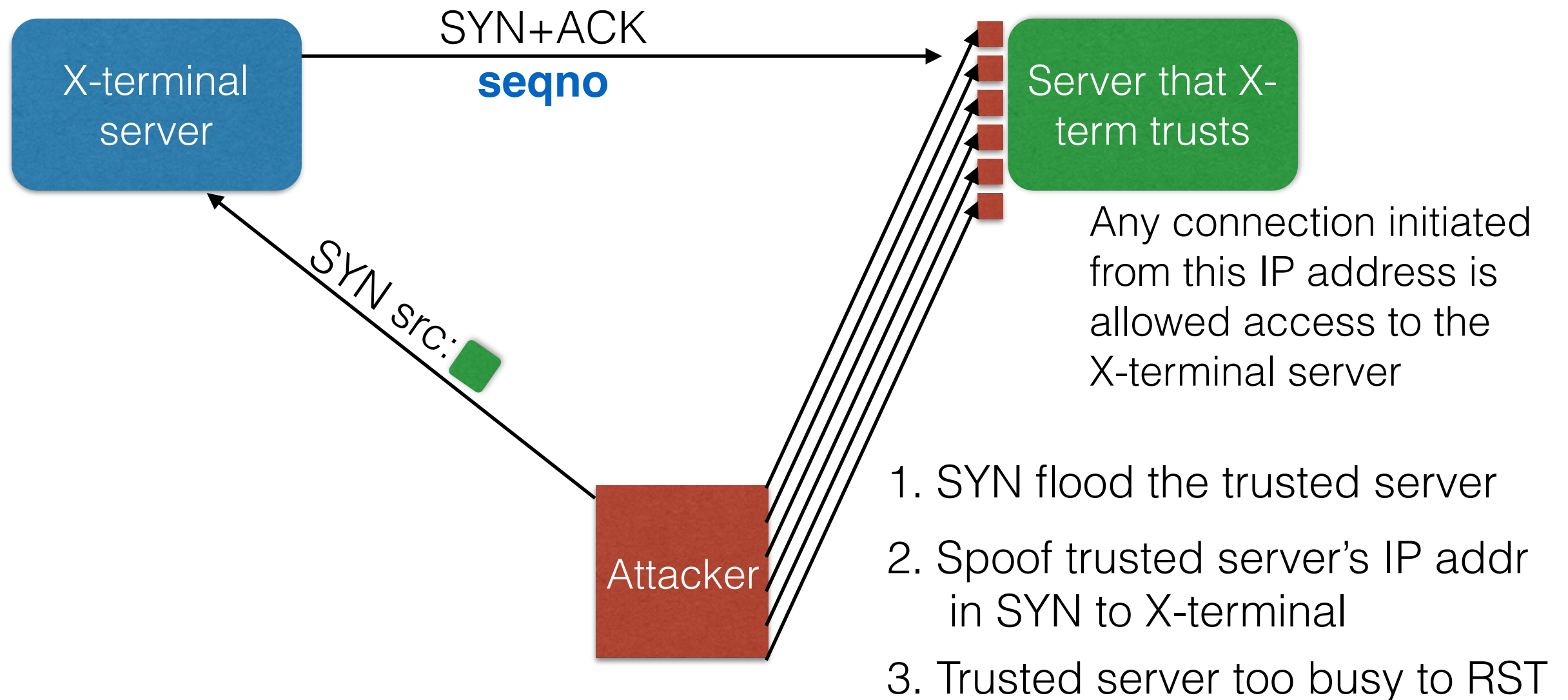
Mitnick attack



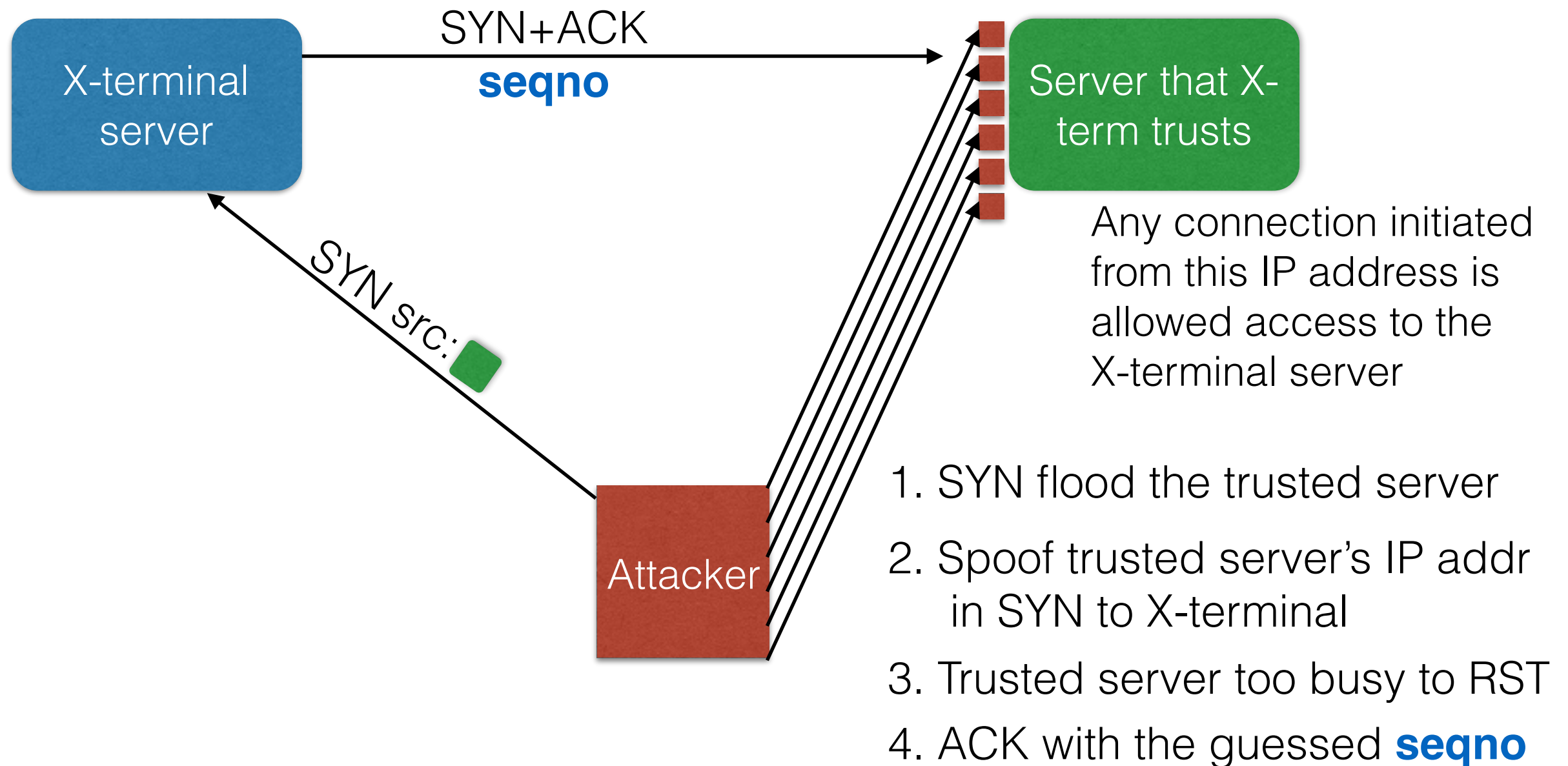
Mitnick attack



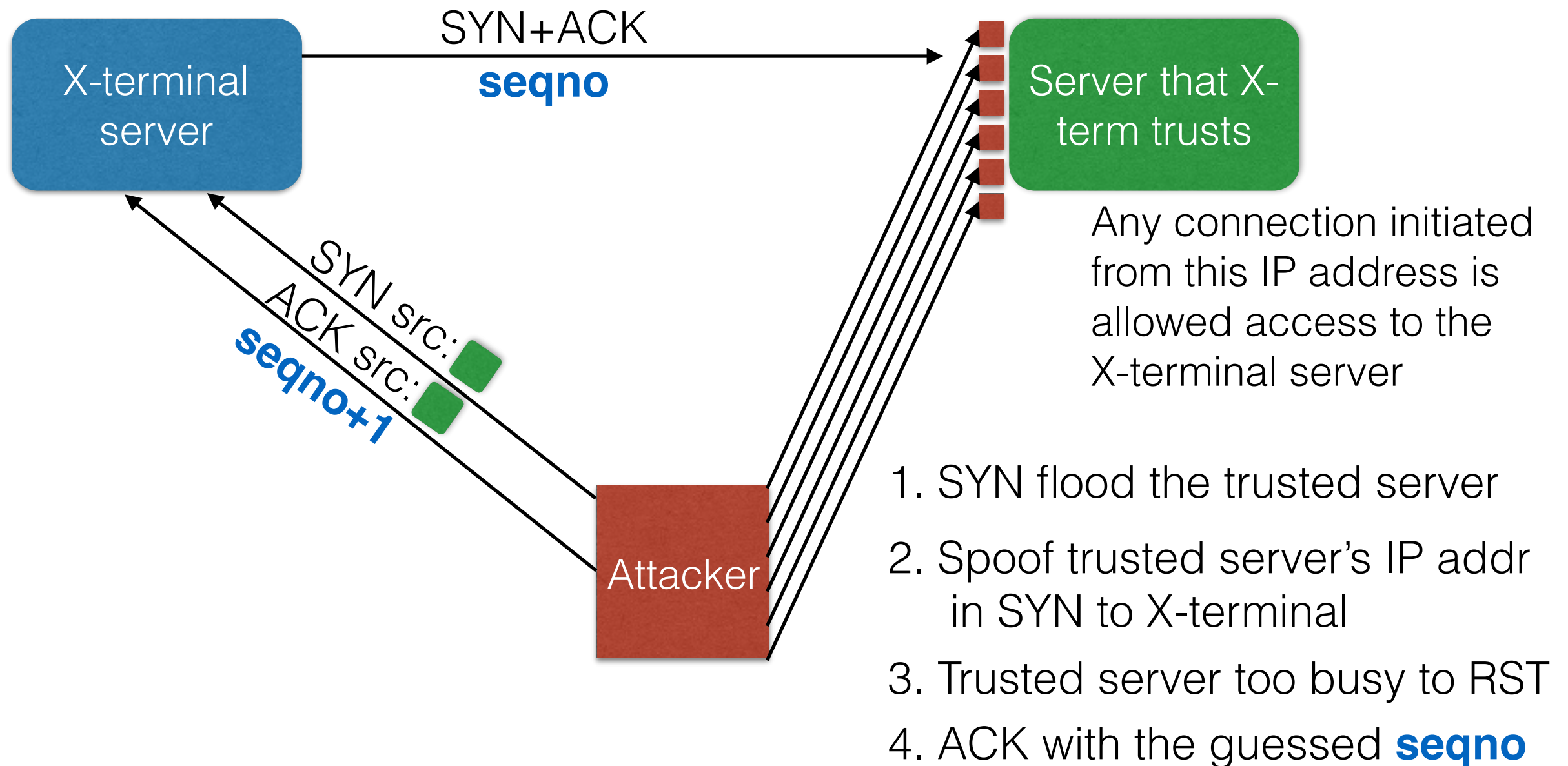
Mitnick attack



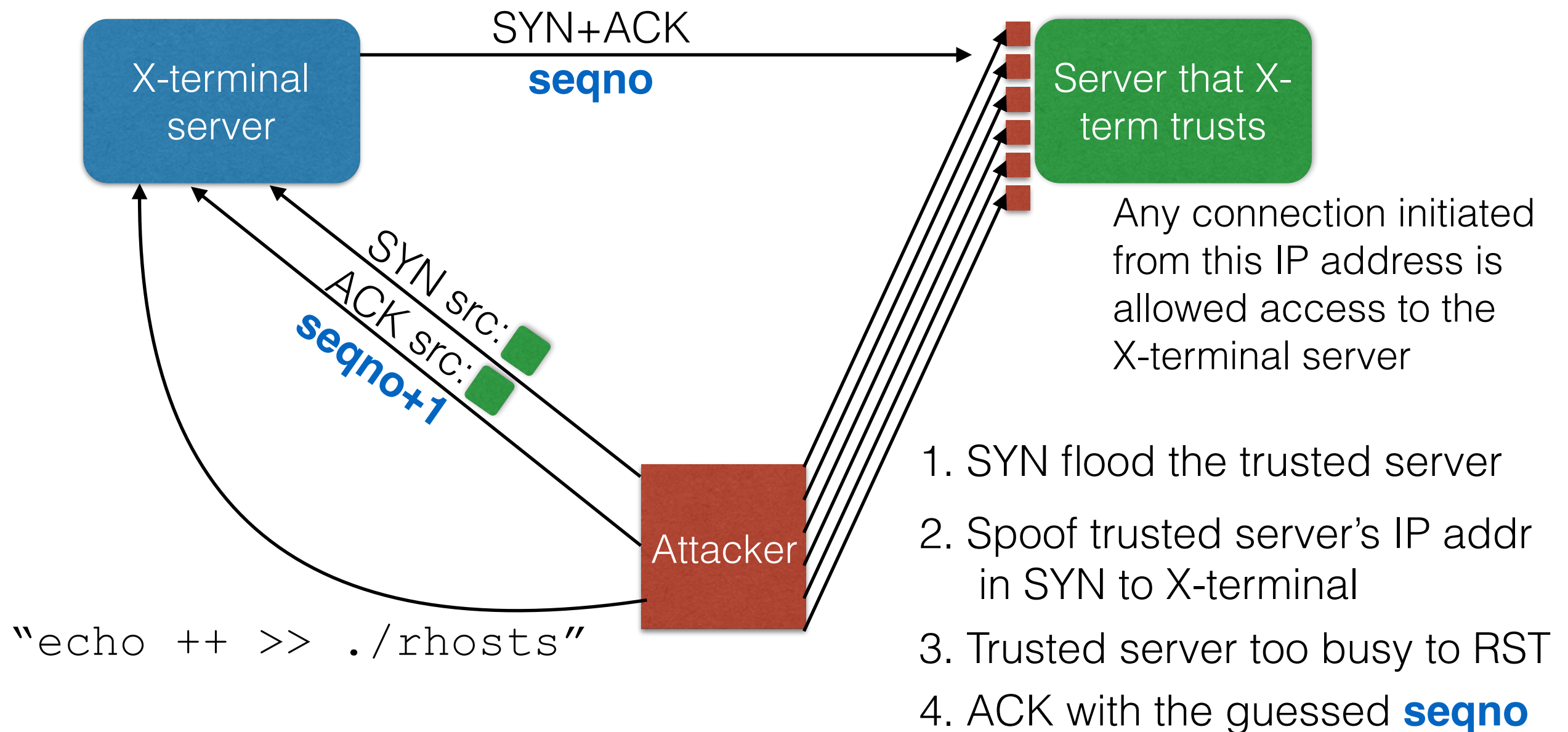
Mitnick attack



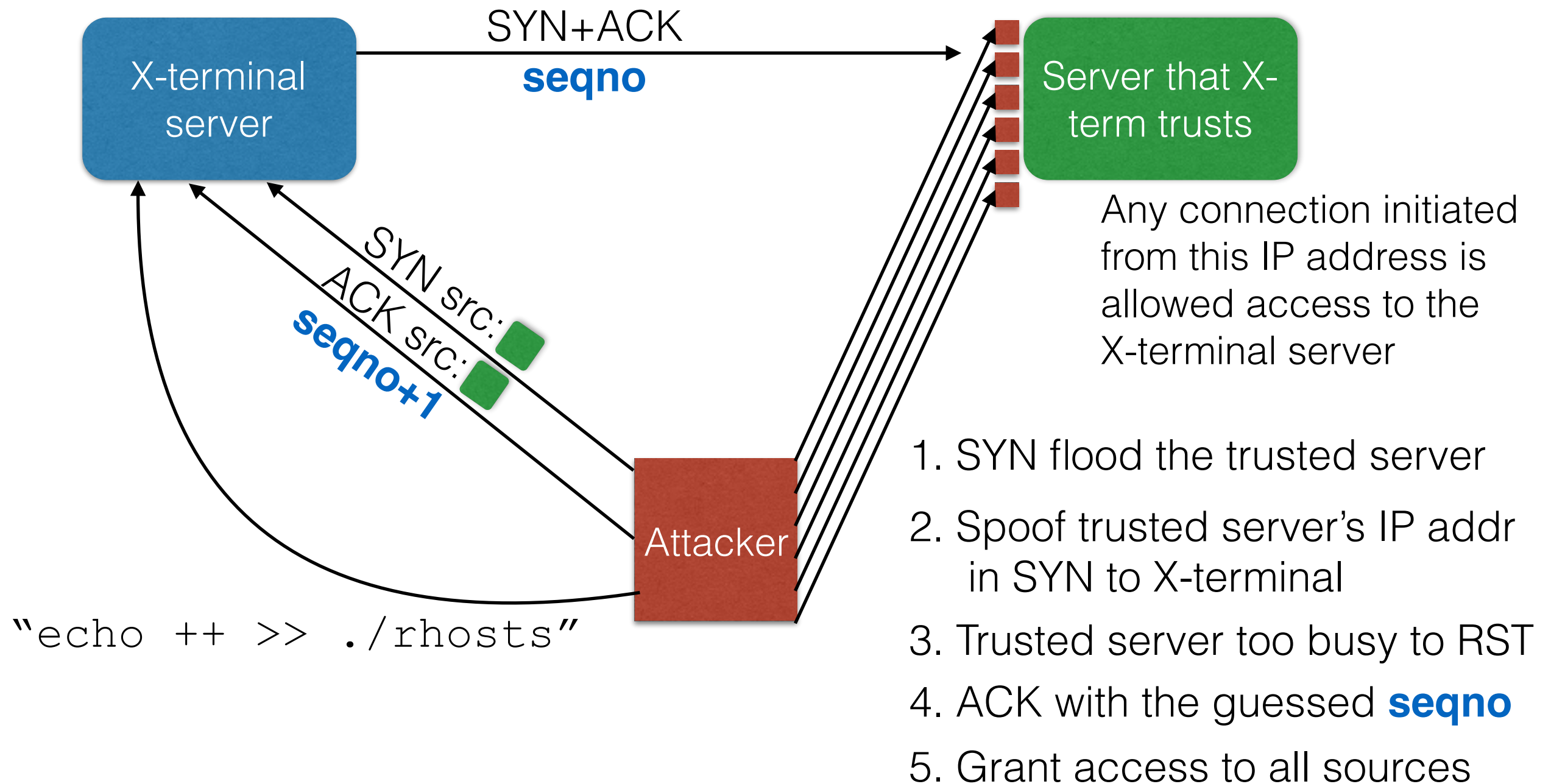
Mitnick attack



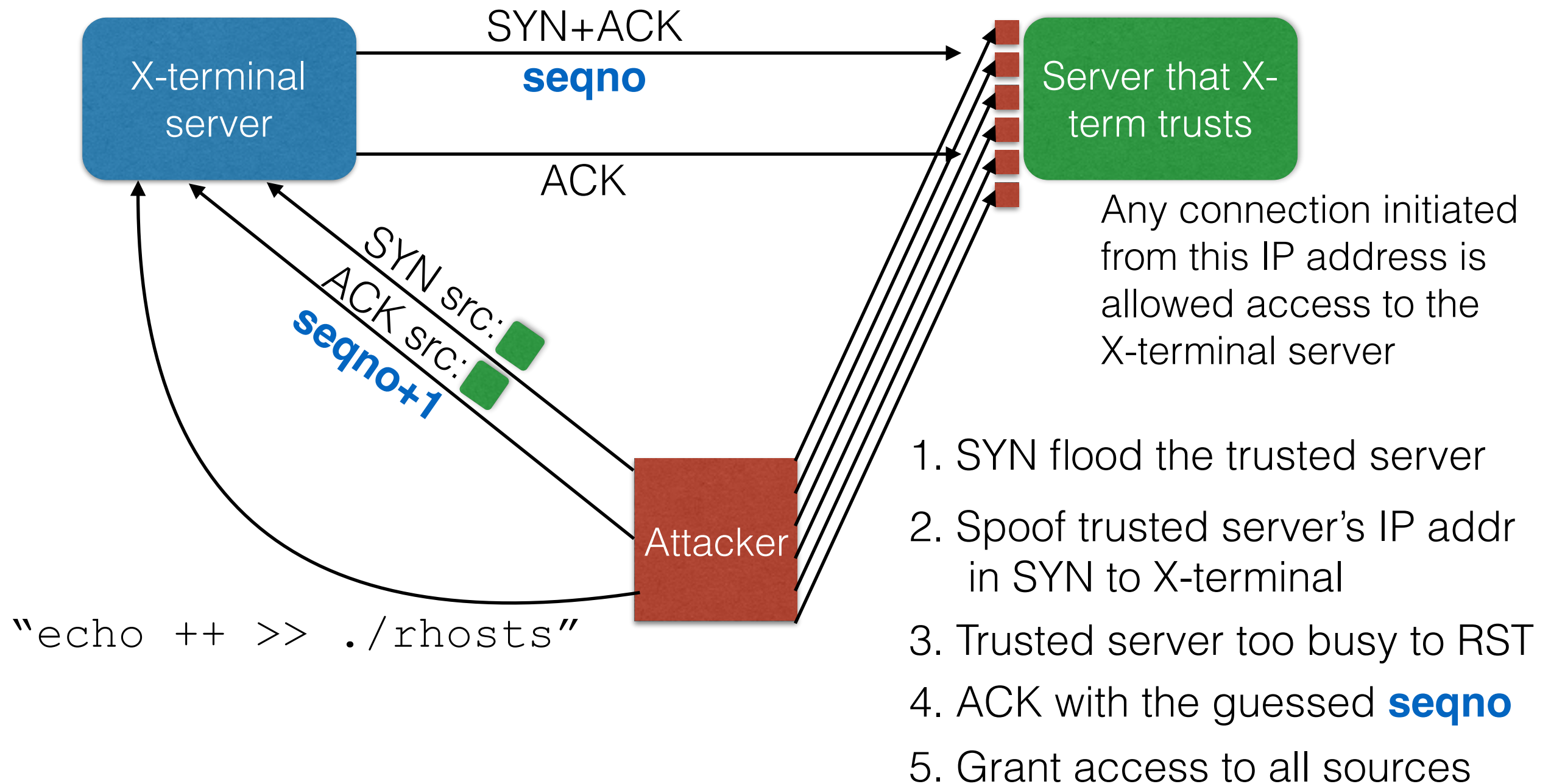
Mitnick attack



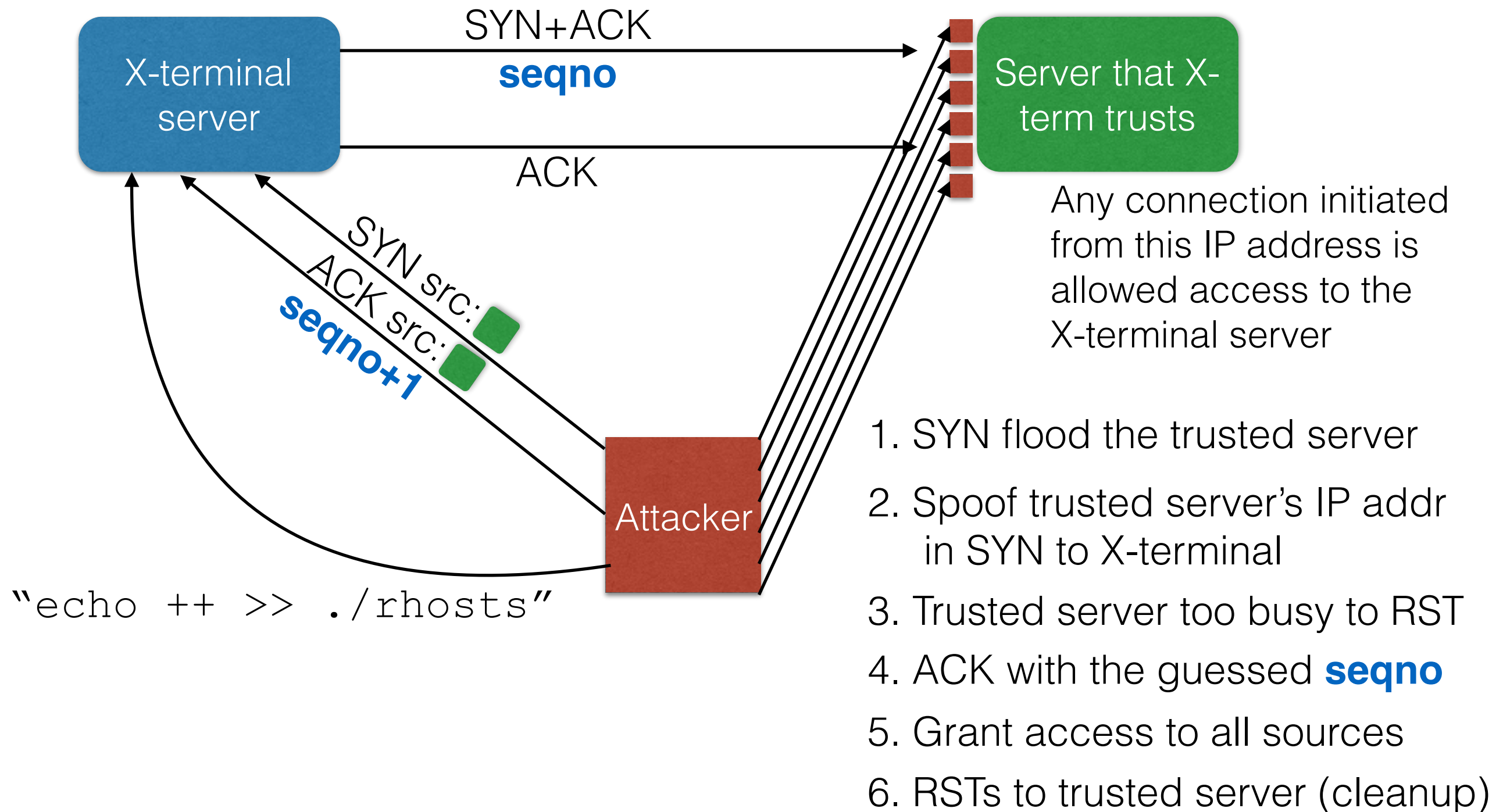
Mitnick attack



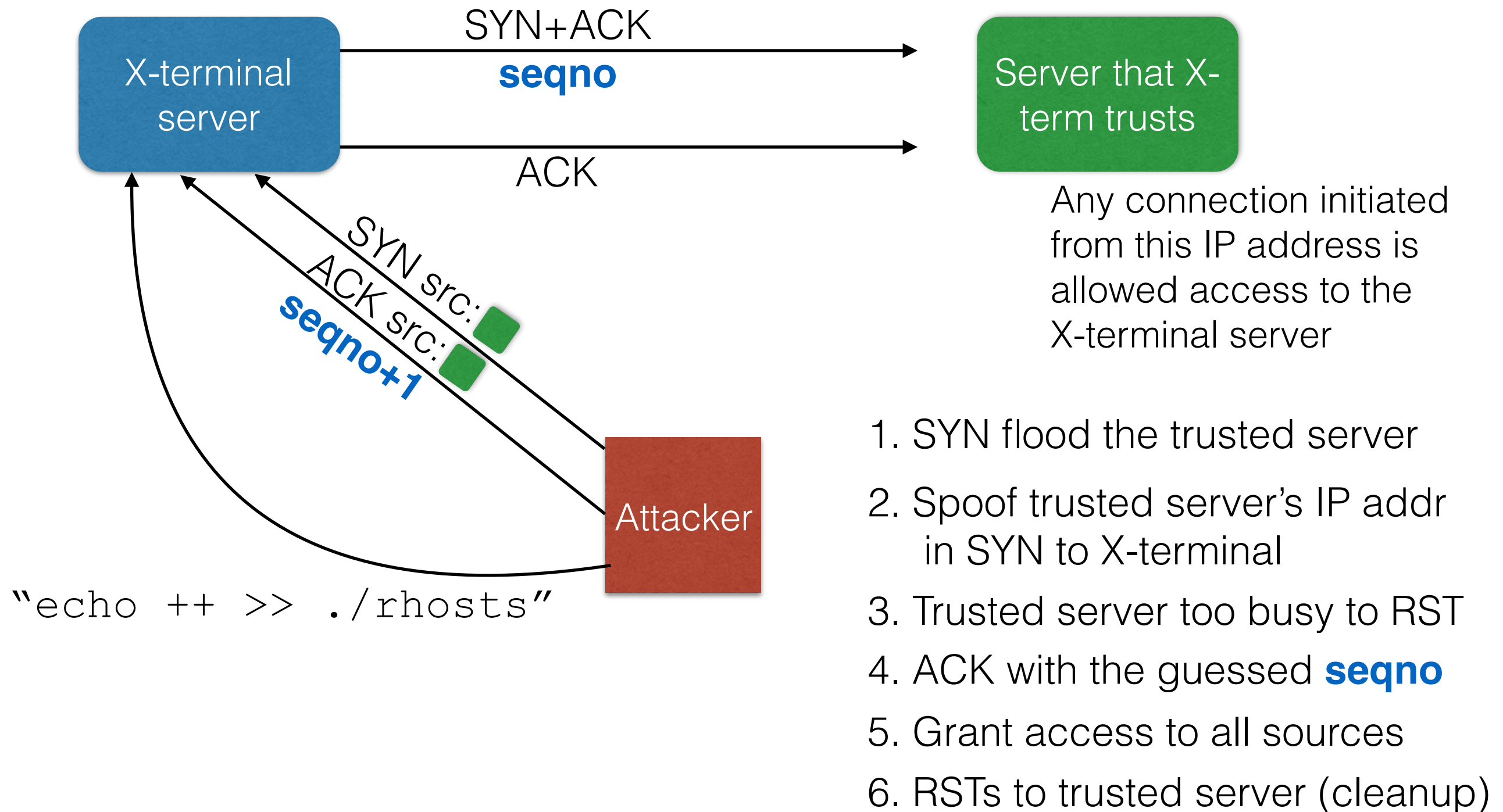
Mitnick attack



Mitnick attack



Mitnick attack



Defenses

- Initial sequence number must be difficult to predict!

Opt-ack attack

A



B



TCP uses ACKs not only for reliability, but also for
congestion control:
the more ACKs come back, the faster I can send

Opt-ack attack

A



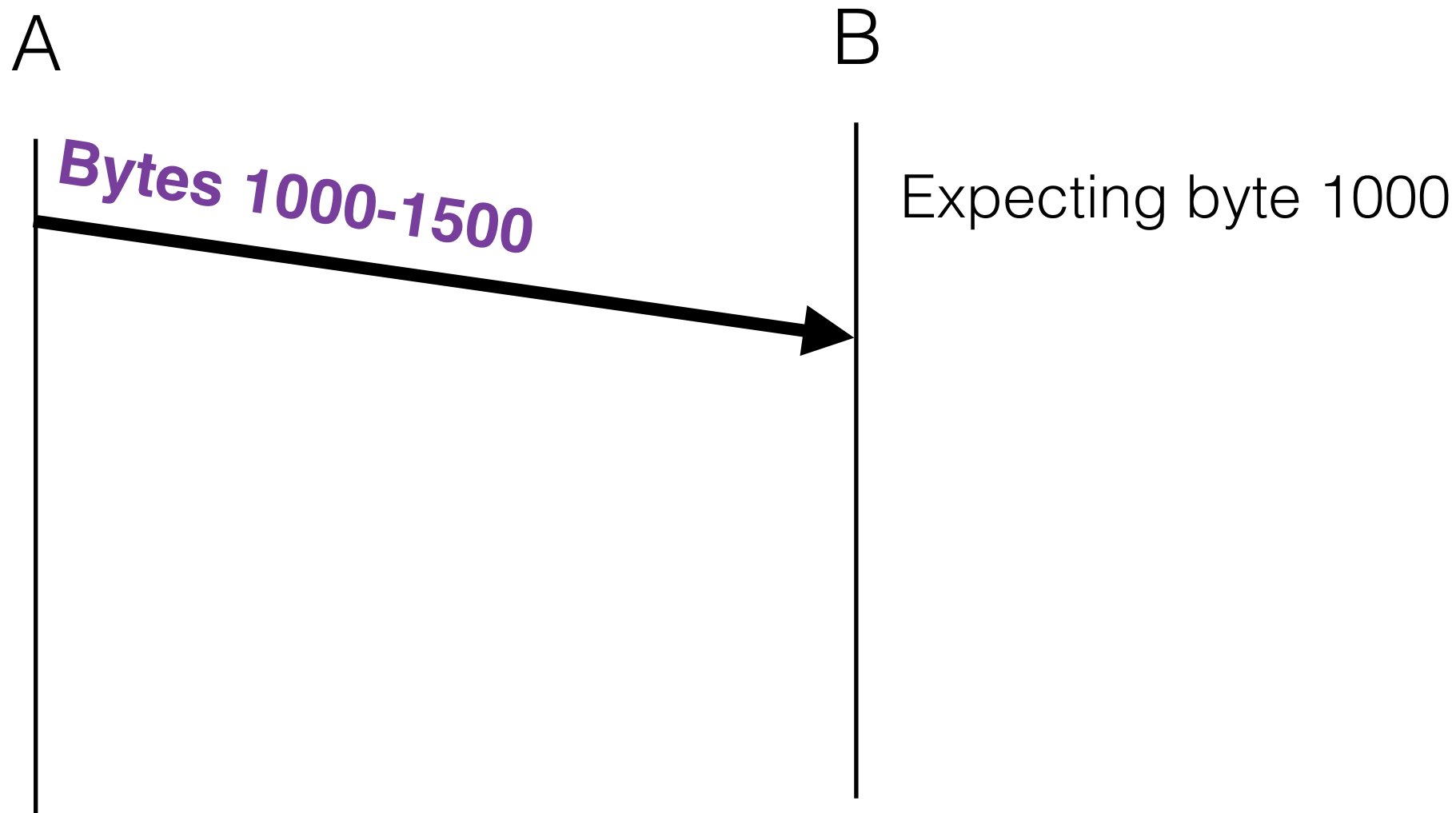
B



Expecting byte 1000

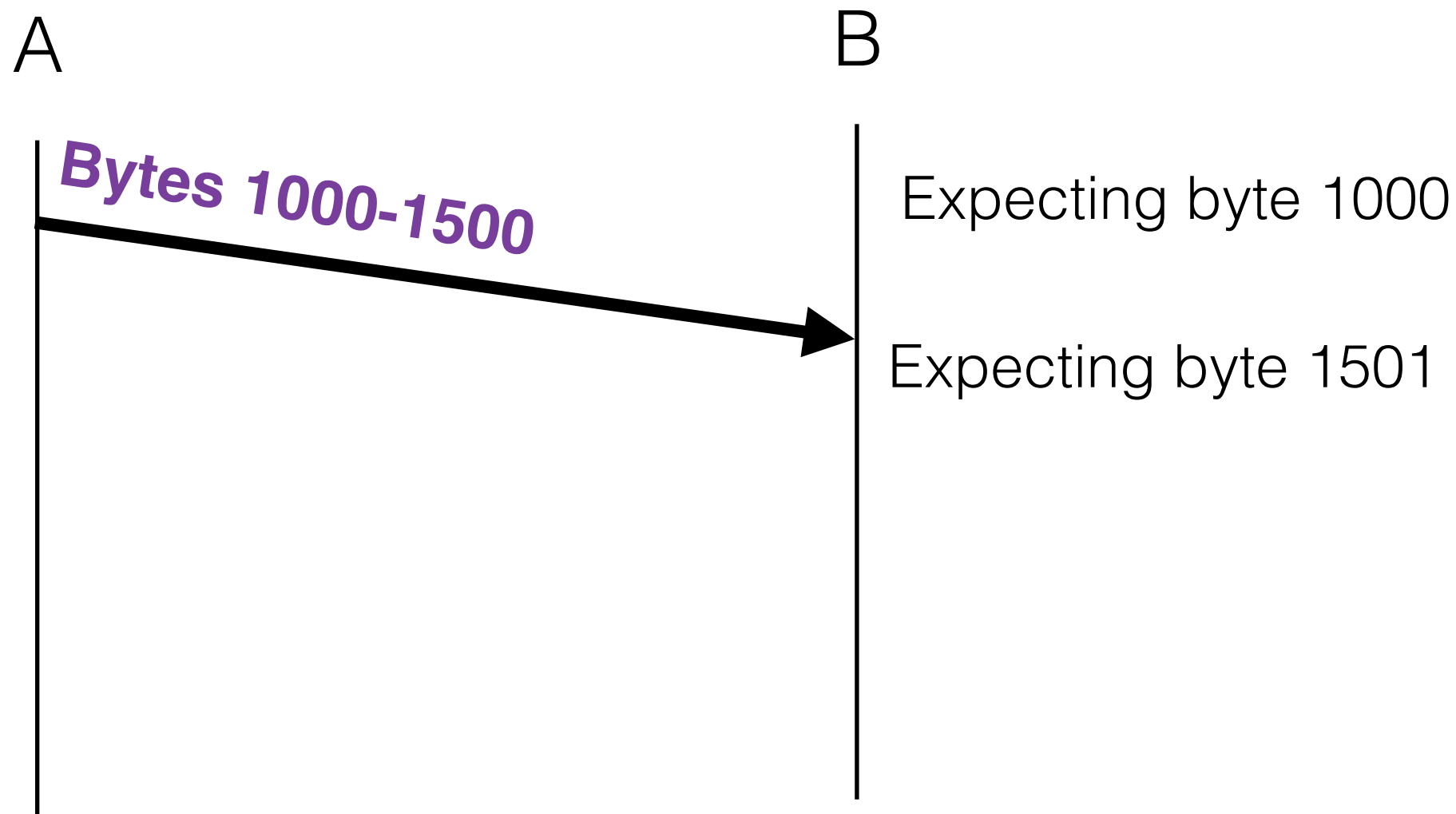
TCP uses ACKs not only for reliability, but also for
congestion control:
the more ACKs come back, the faster I can send

Opt-ack attack



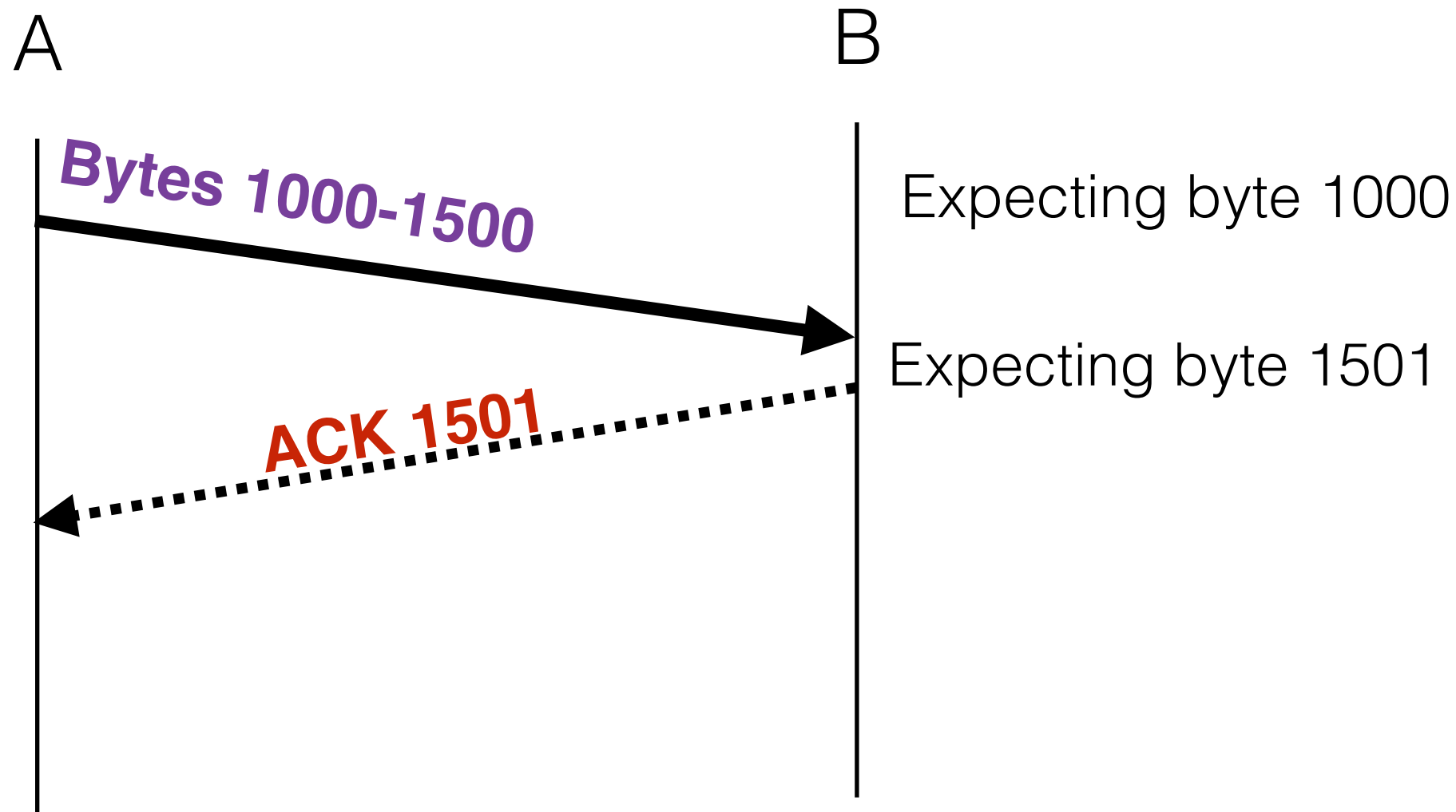
TCP uses ACKs not only for reliability, but also for **congestion control**:
the more ACKs come back, the faster I can send

Opt-ack attack



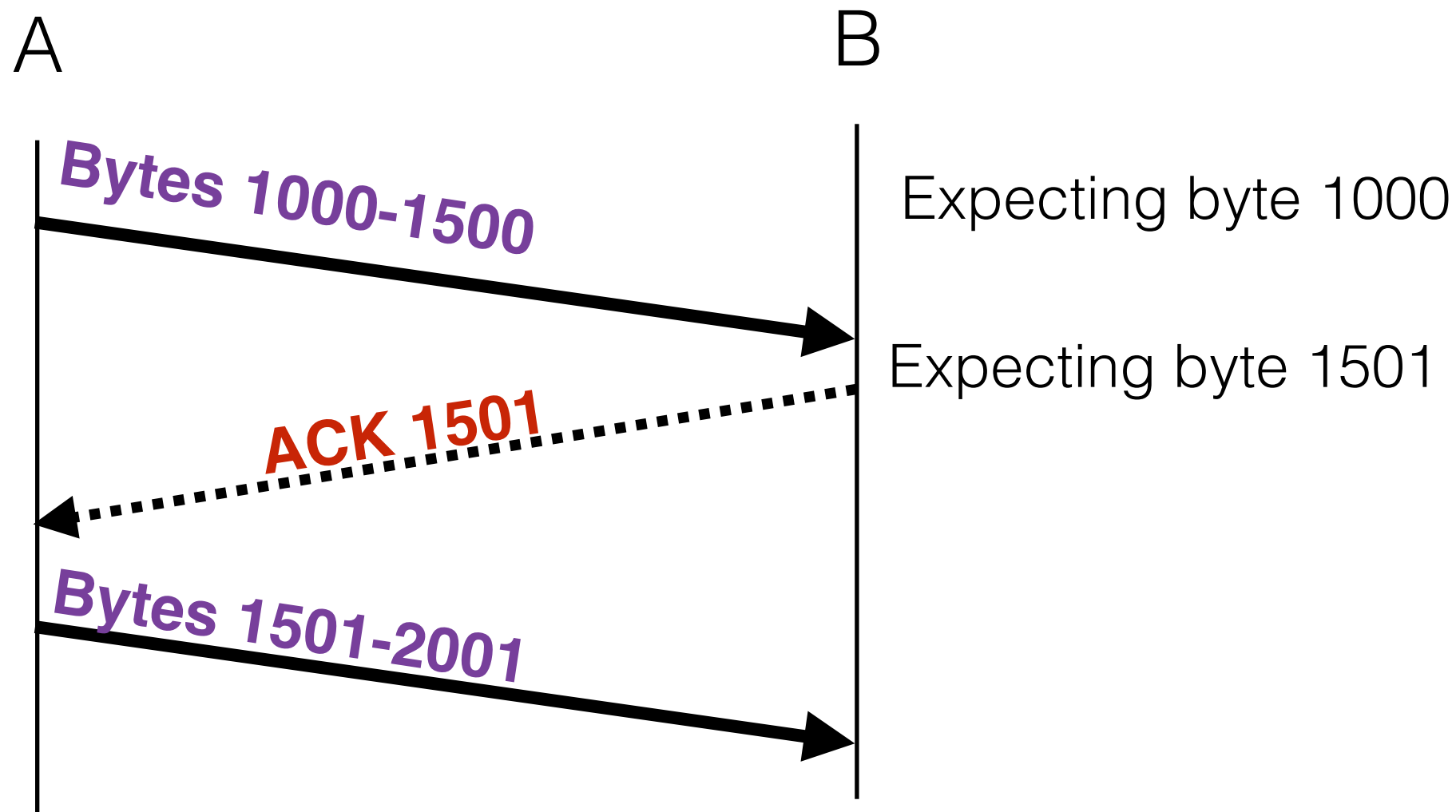
TCP uses ACKs not only for reliability, but also for **congestion control**:
the more ACKs come back, the faster I can send

Opt-ack attack



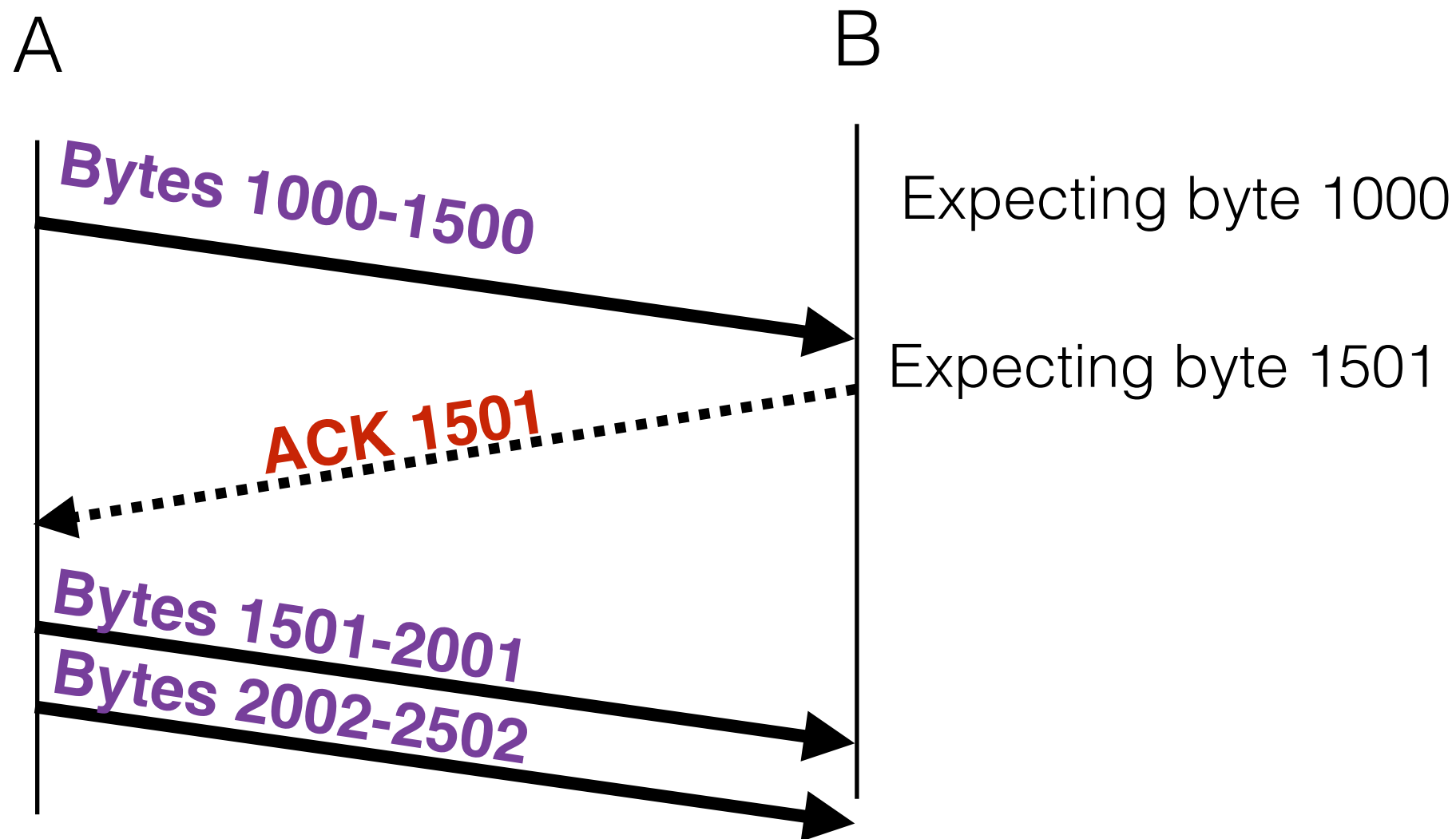
TCP uses ACKs not only for reliability, but also for **congestion control**:
the more ACKs come back, the faster I can send

Opt-ack attack



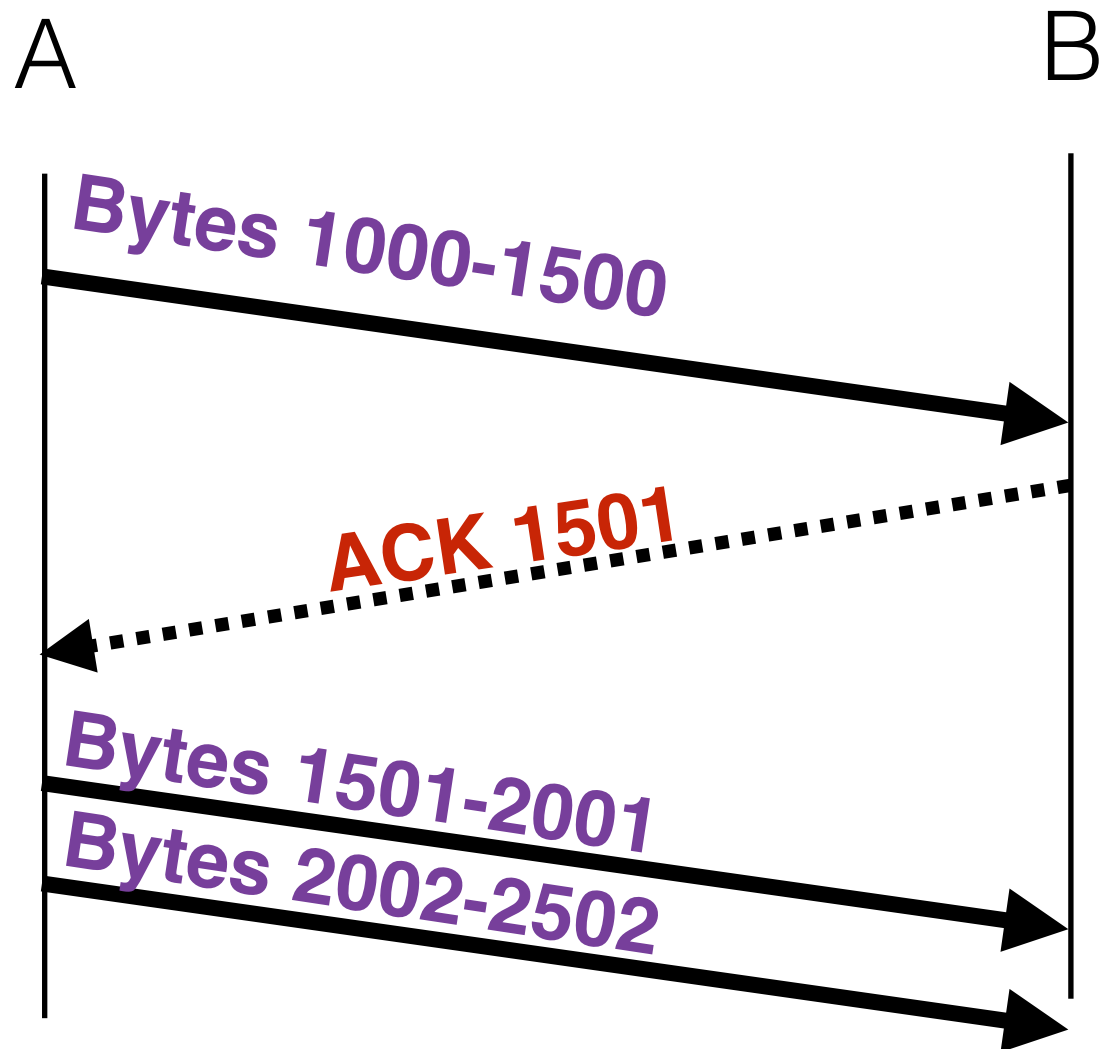
TCP uses ACKs not only for reliability, but also for **congestion control**:
the more ACKs come back, the faster I can send

Opt-ack attack

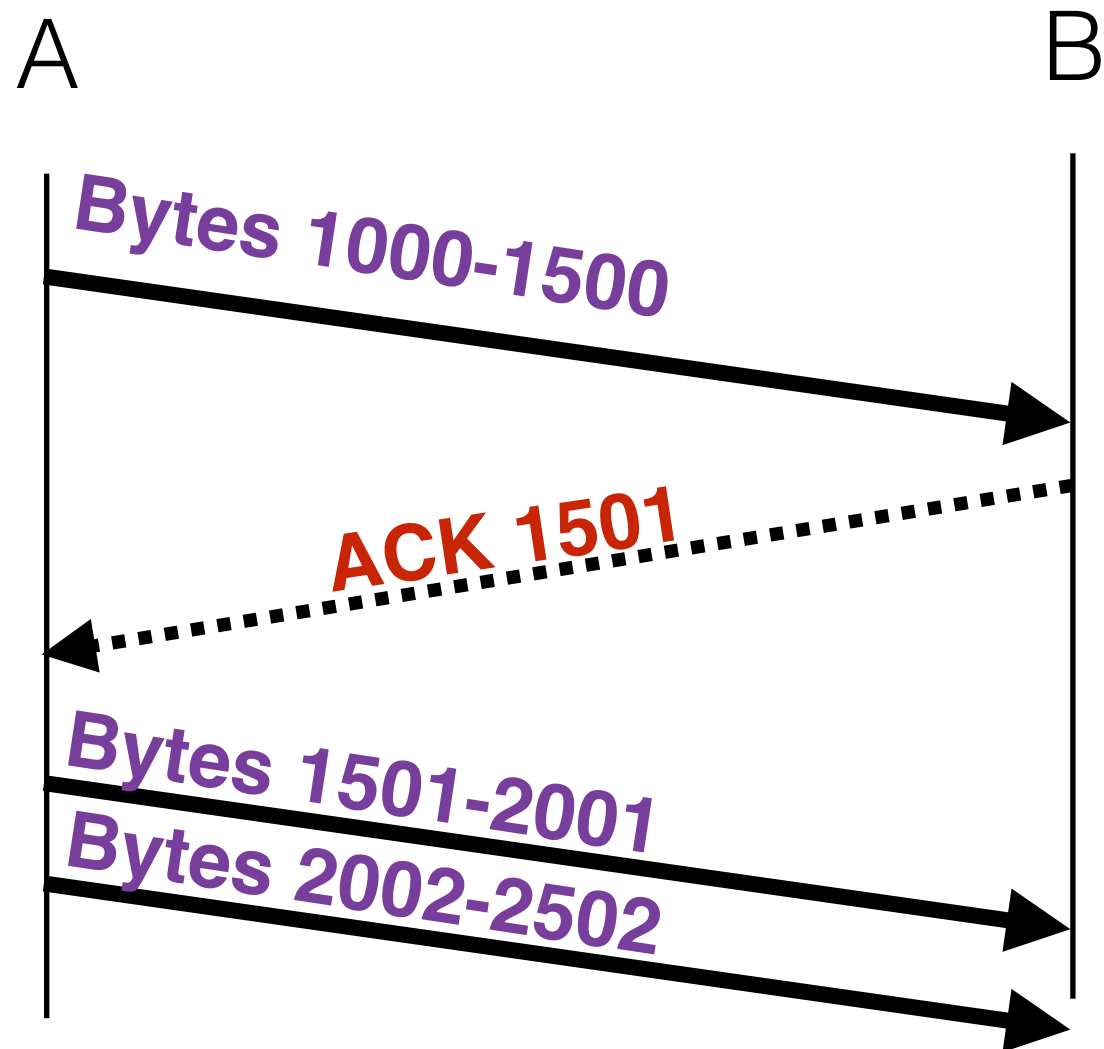


TCP uses ACKs not only for reliability, but also for **congestion control**:
the more ACKs come back, the faster I can send

Opt-ack attack

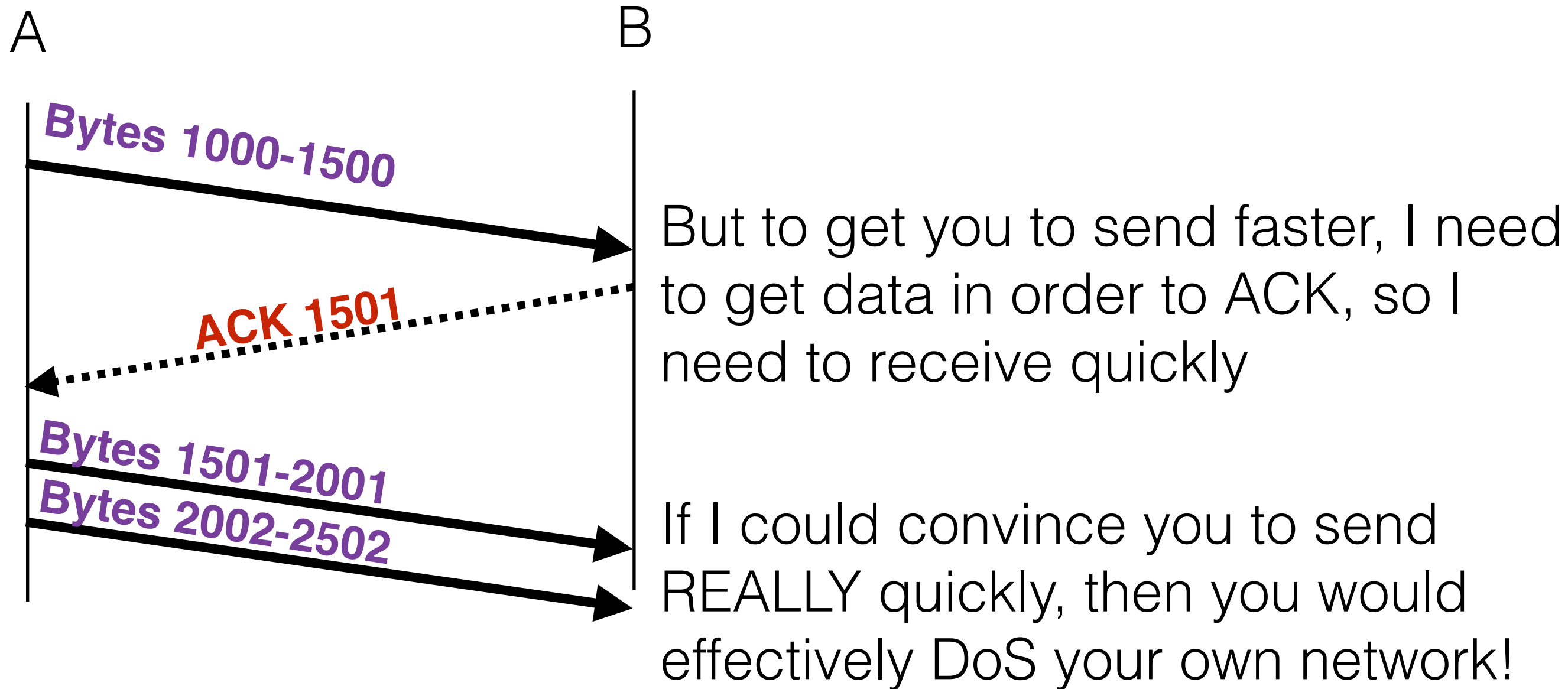


Opt-ack attack

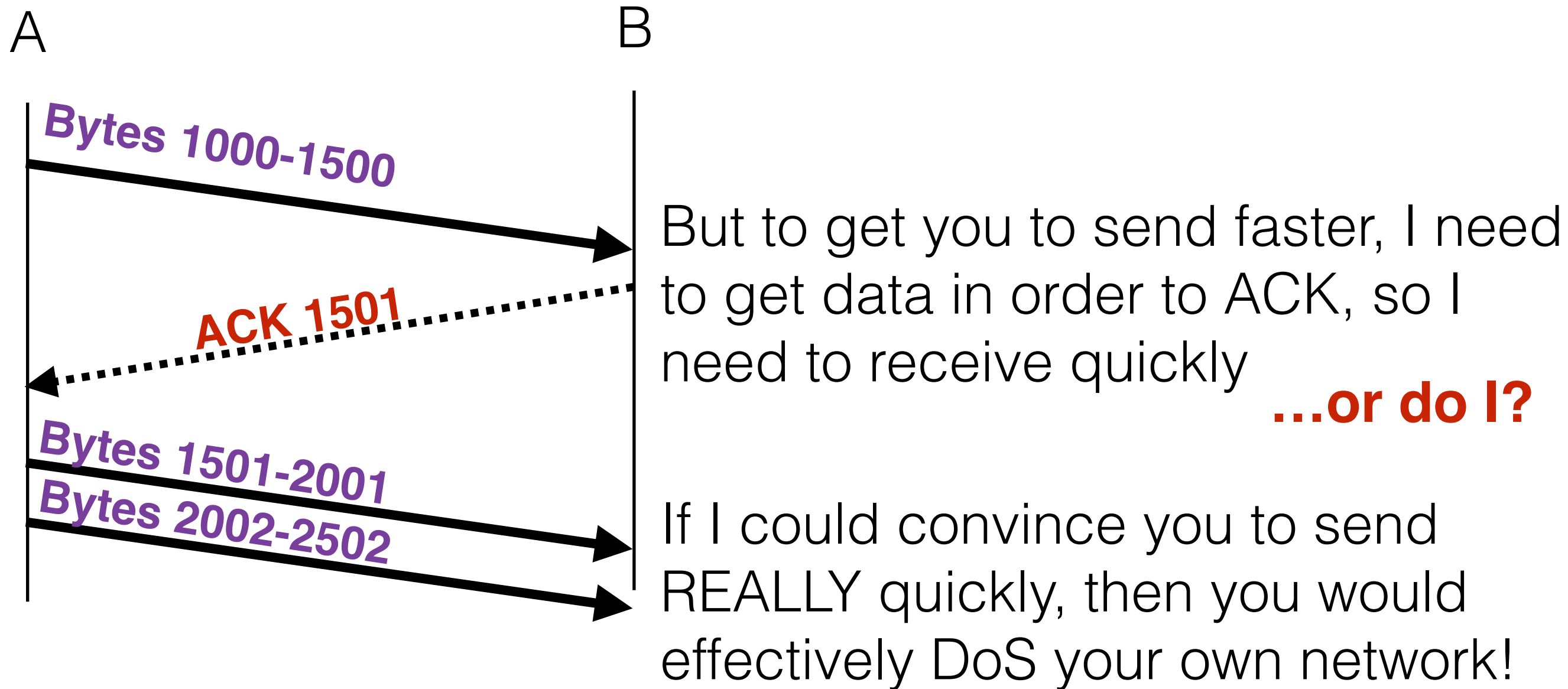


If I could convince you to send REALLY quickly, then you would effectively DoS your own network!

Opt-ack attack



Opt-ack attack



Opt-ack attack

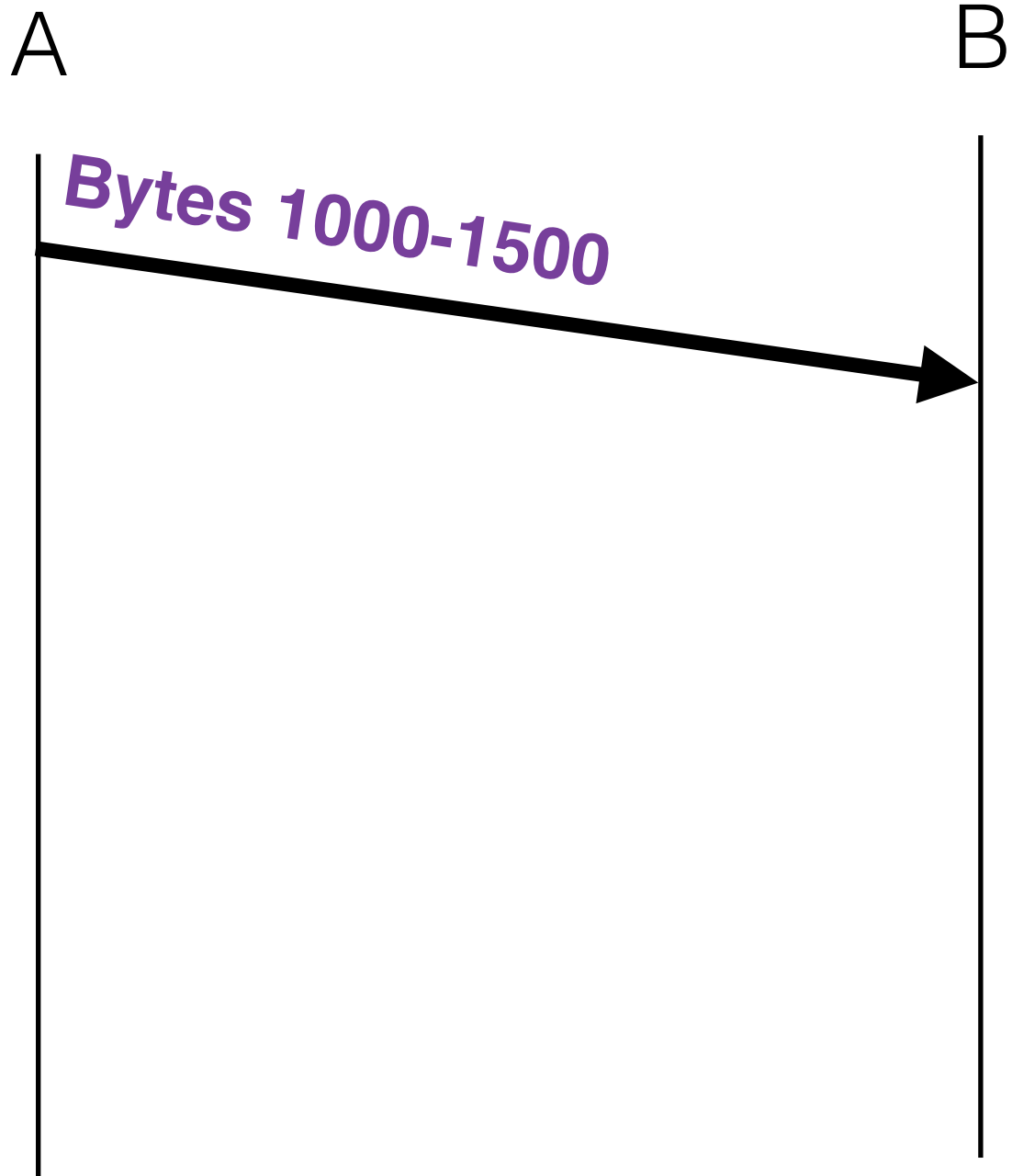
A



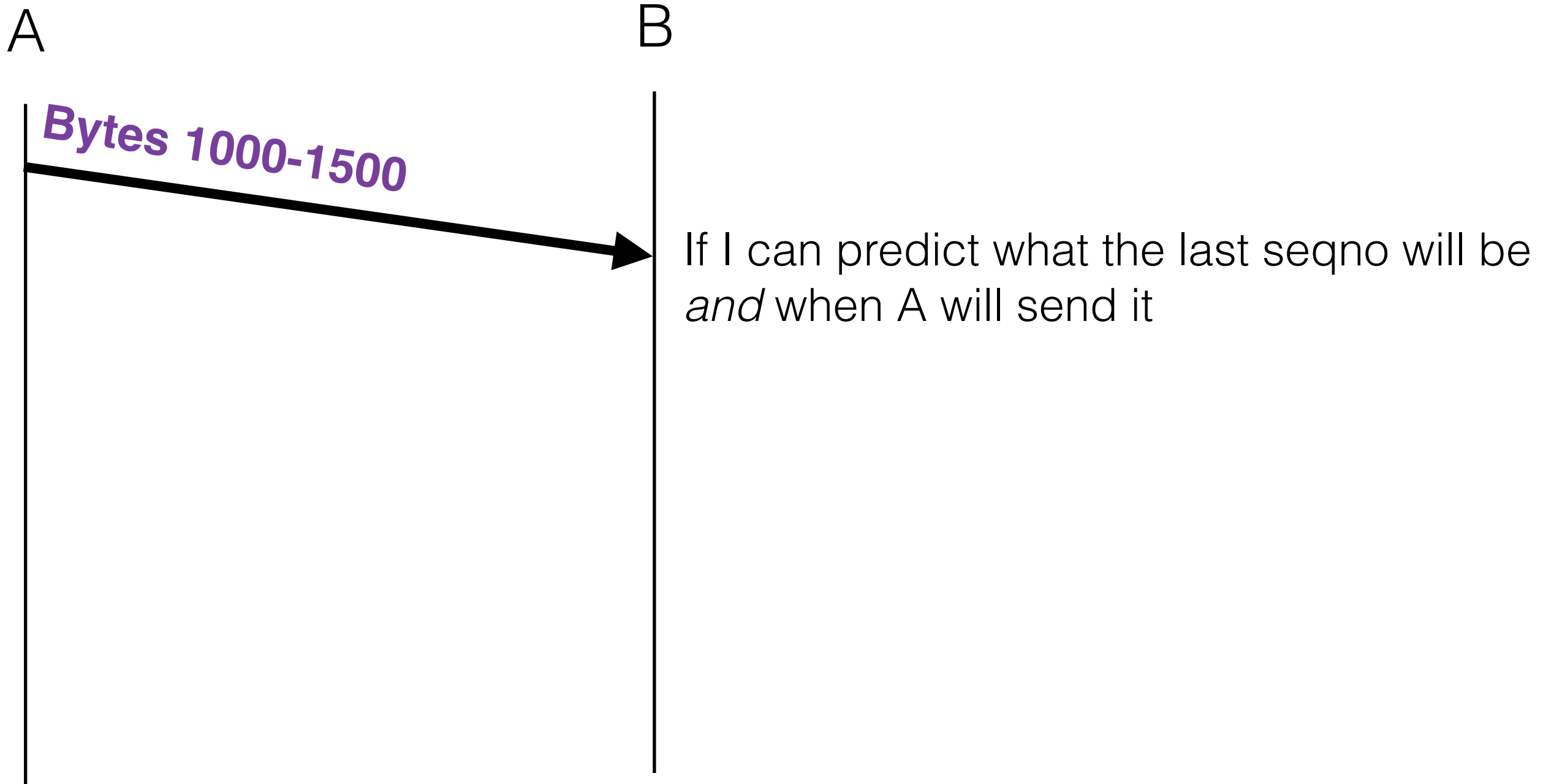
B



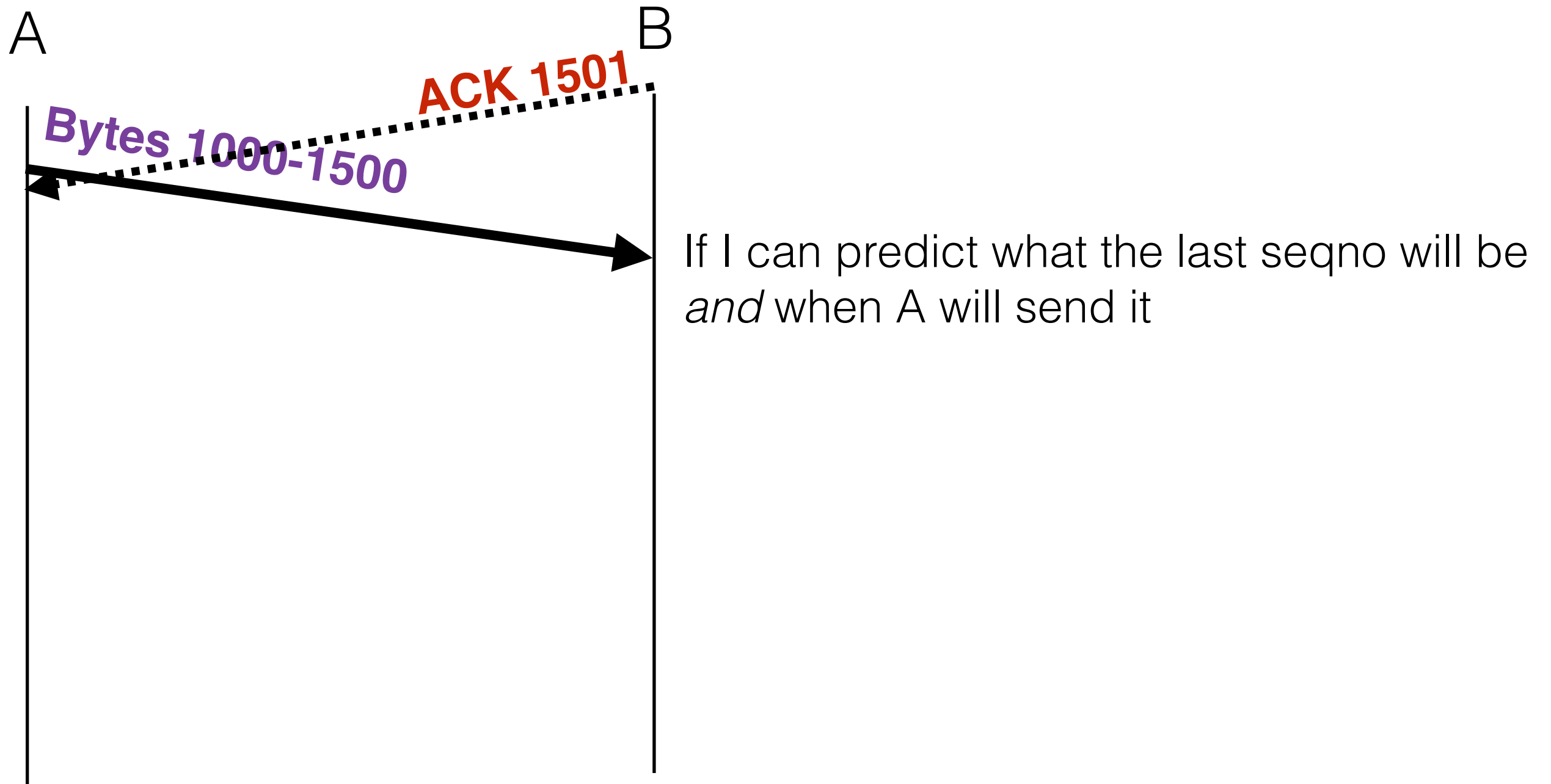
Opt-ack attack



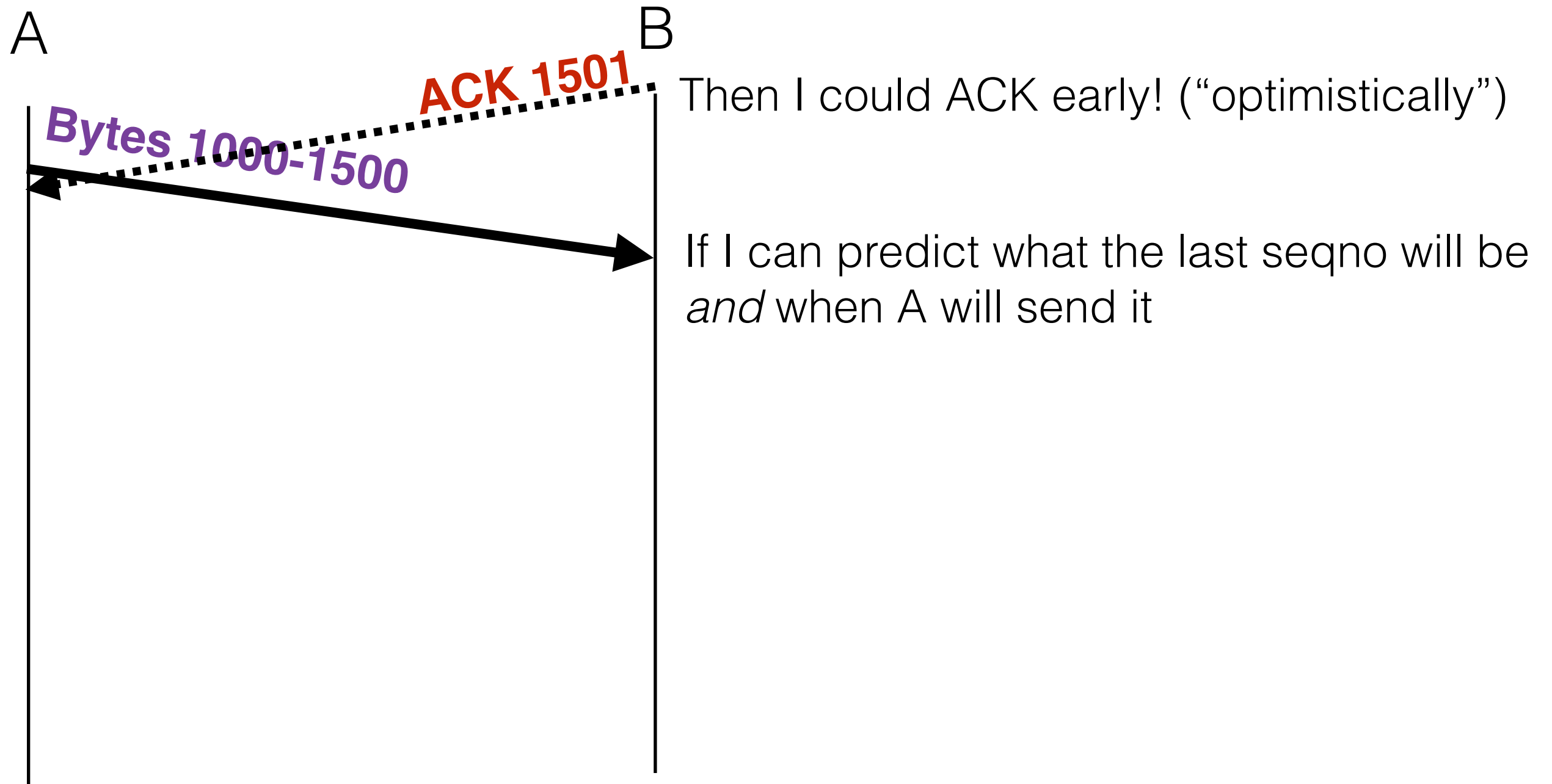
Opt-ack attack



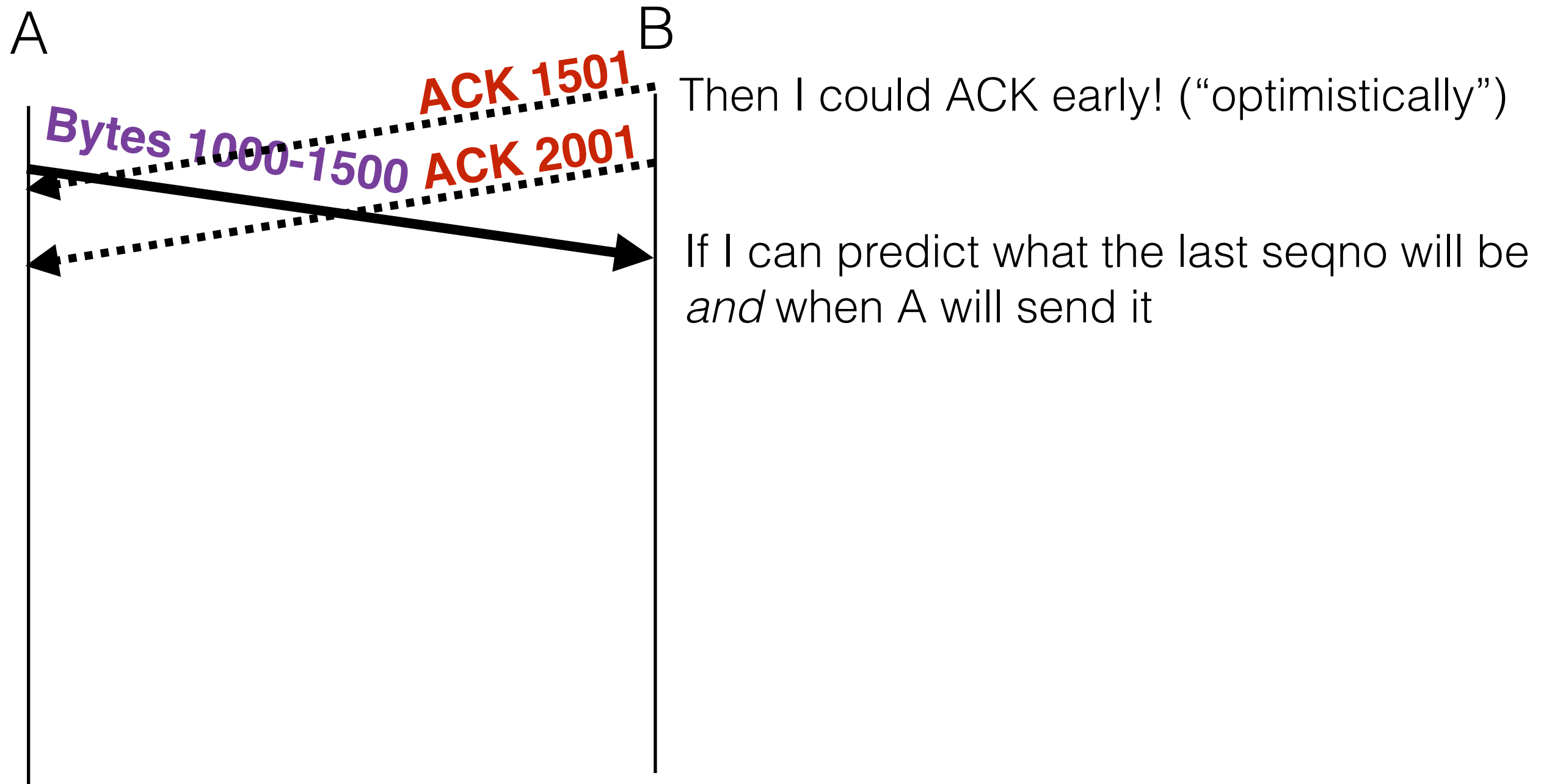
Opt-ack attack



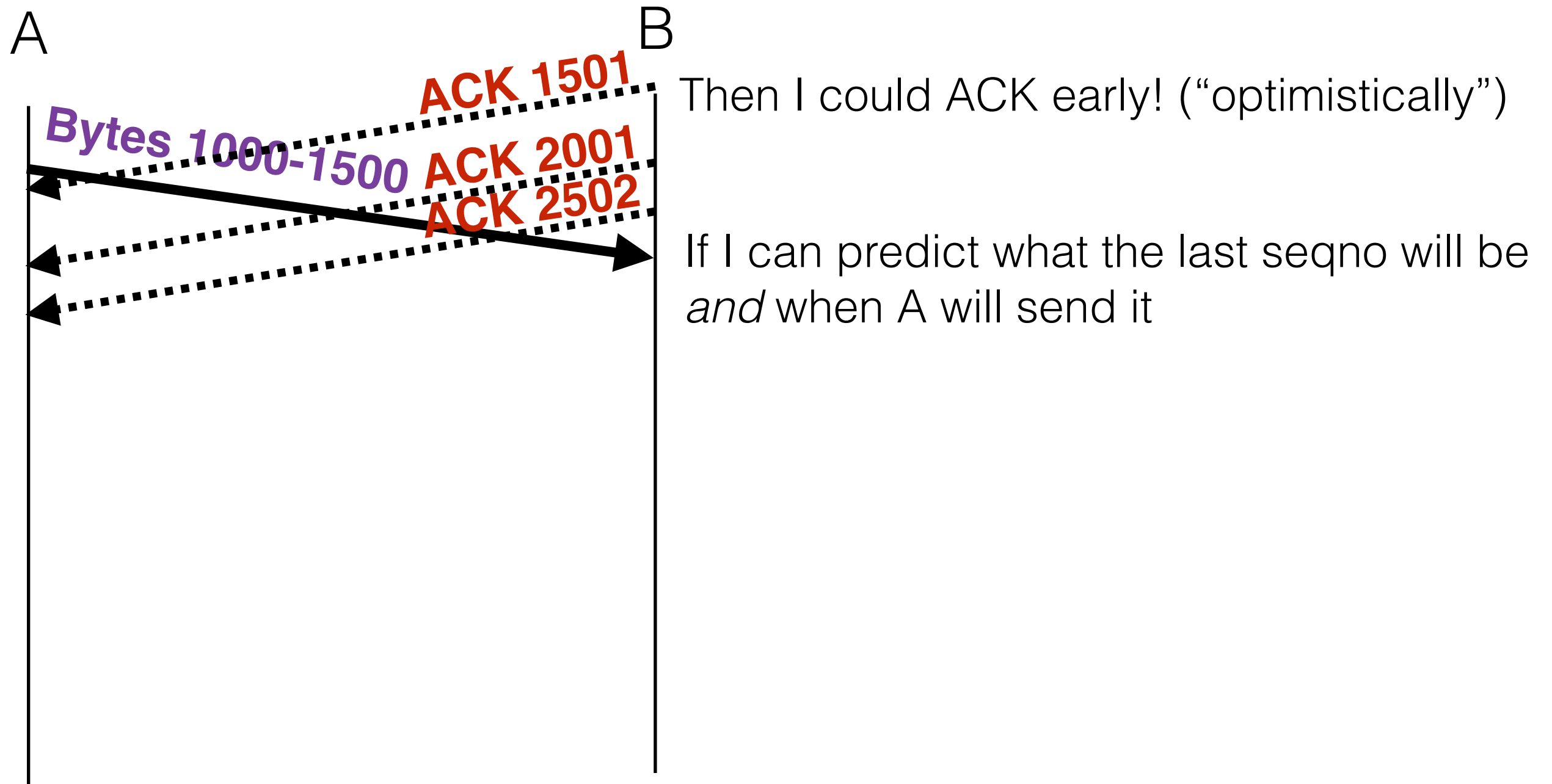
Opt-ack attack



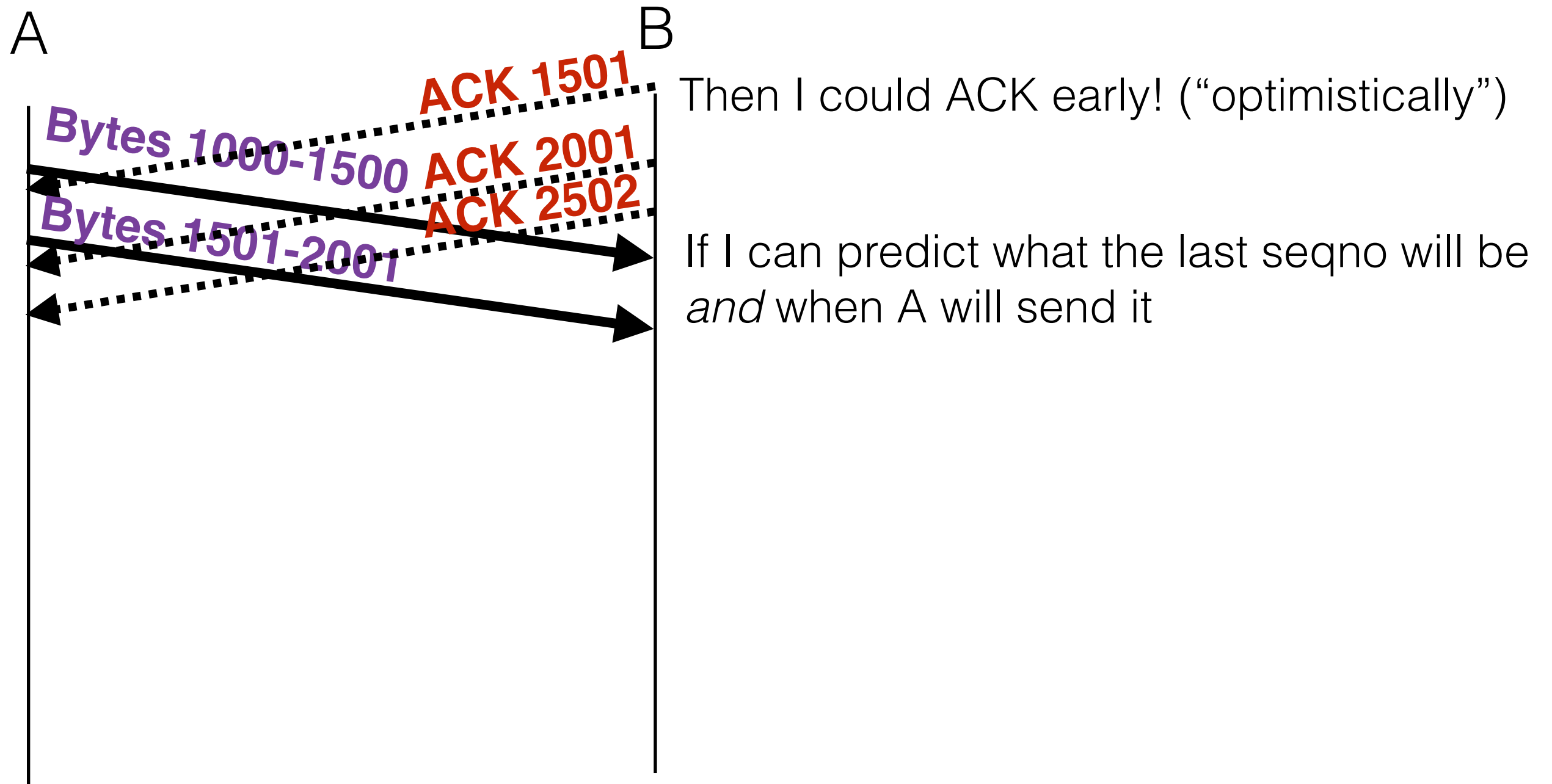
Opt-ack attack



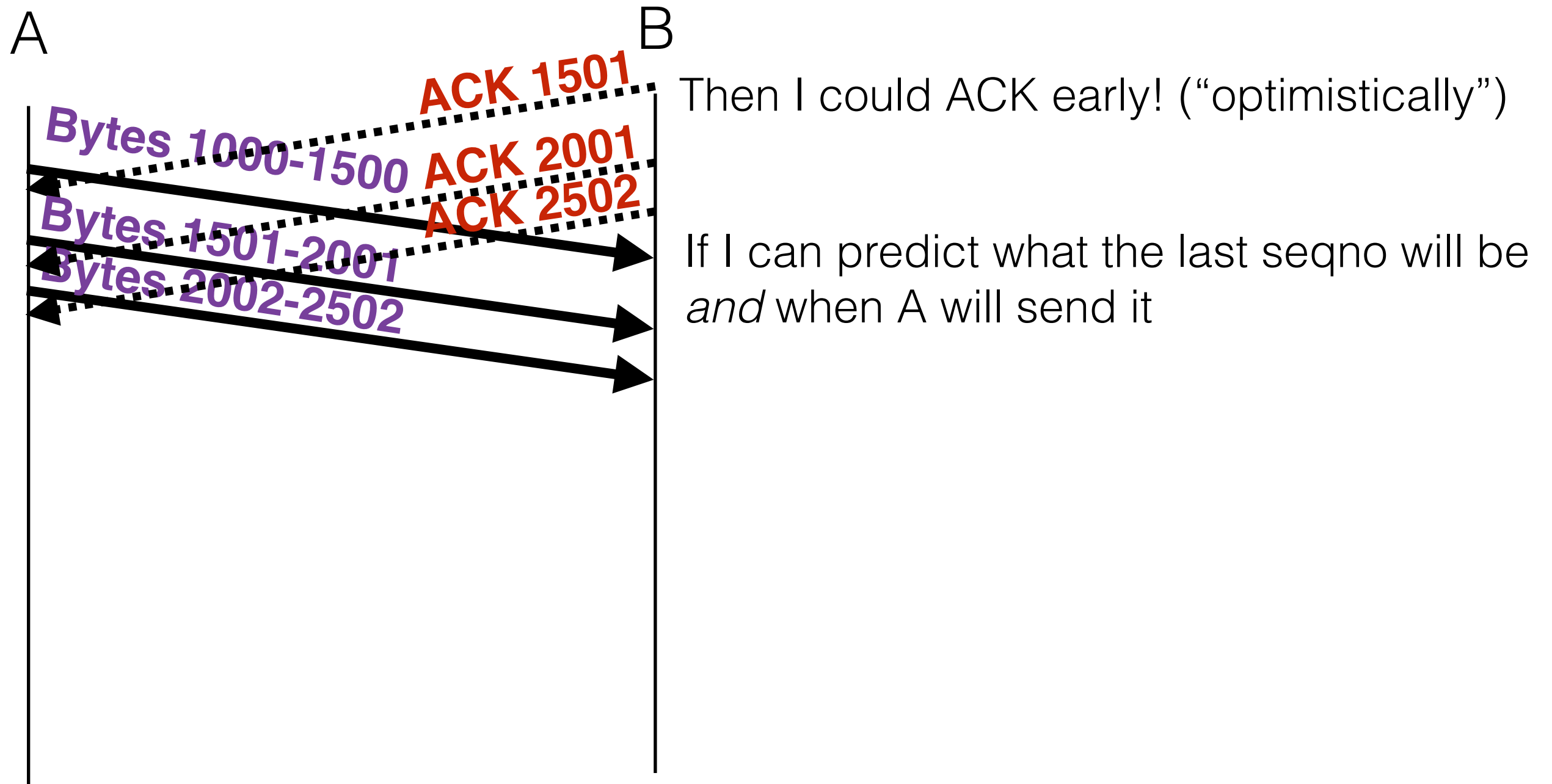
Opt-ack attack



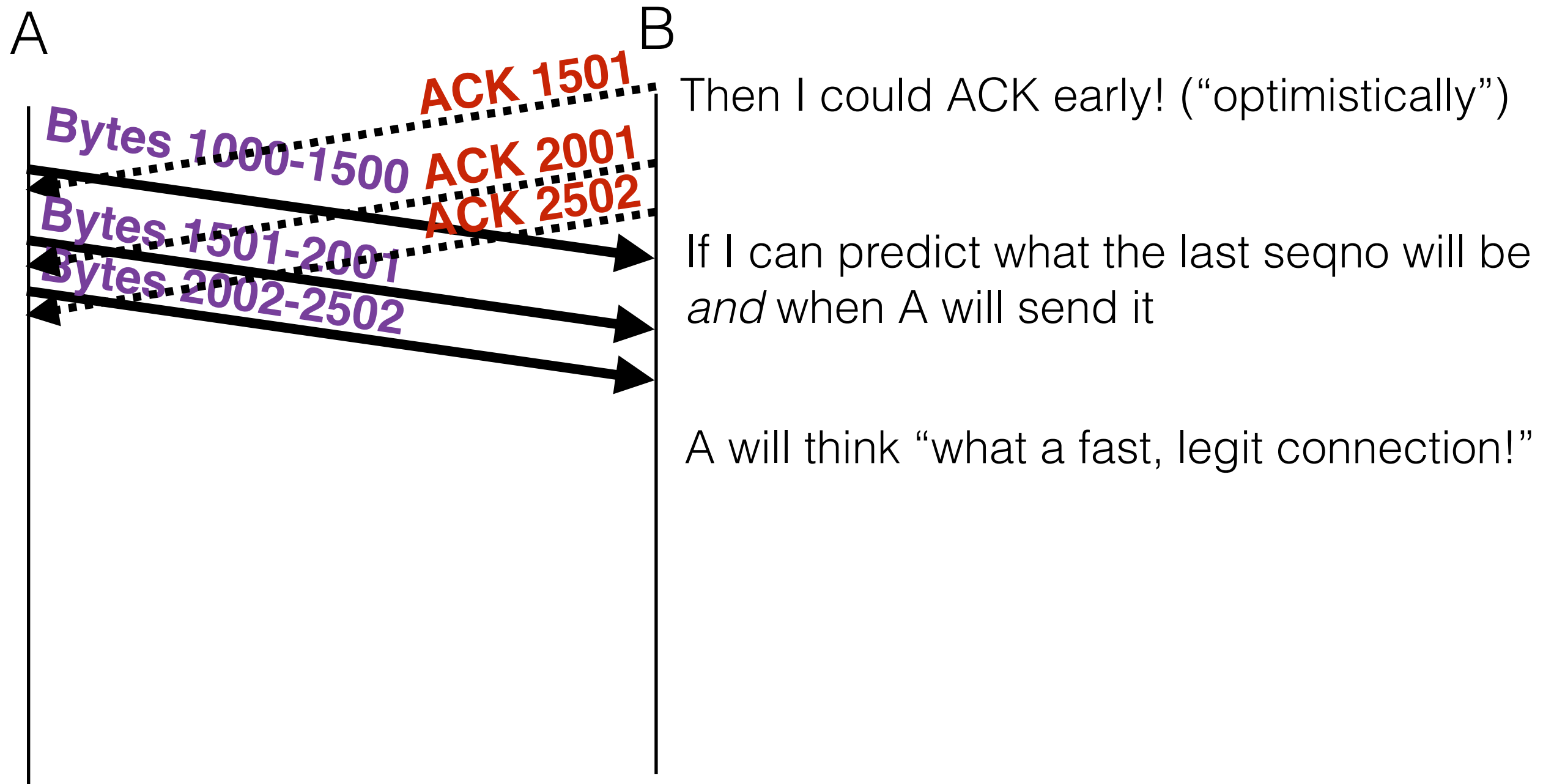
Opt-ack attack



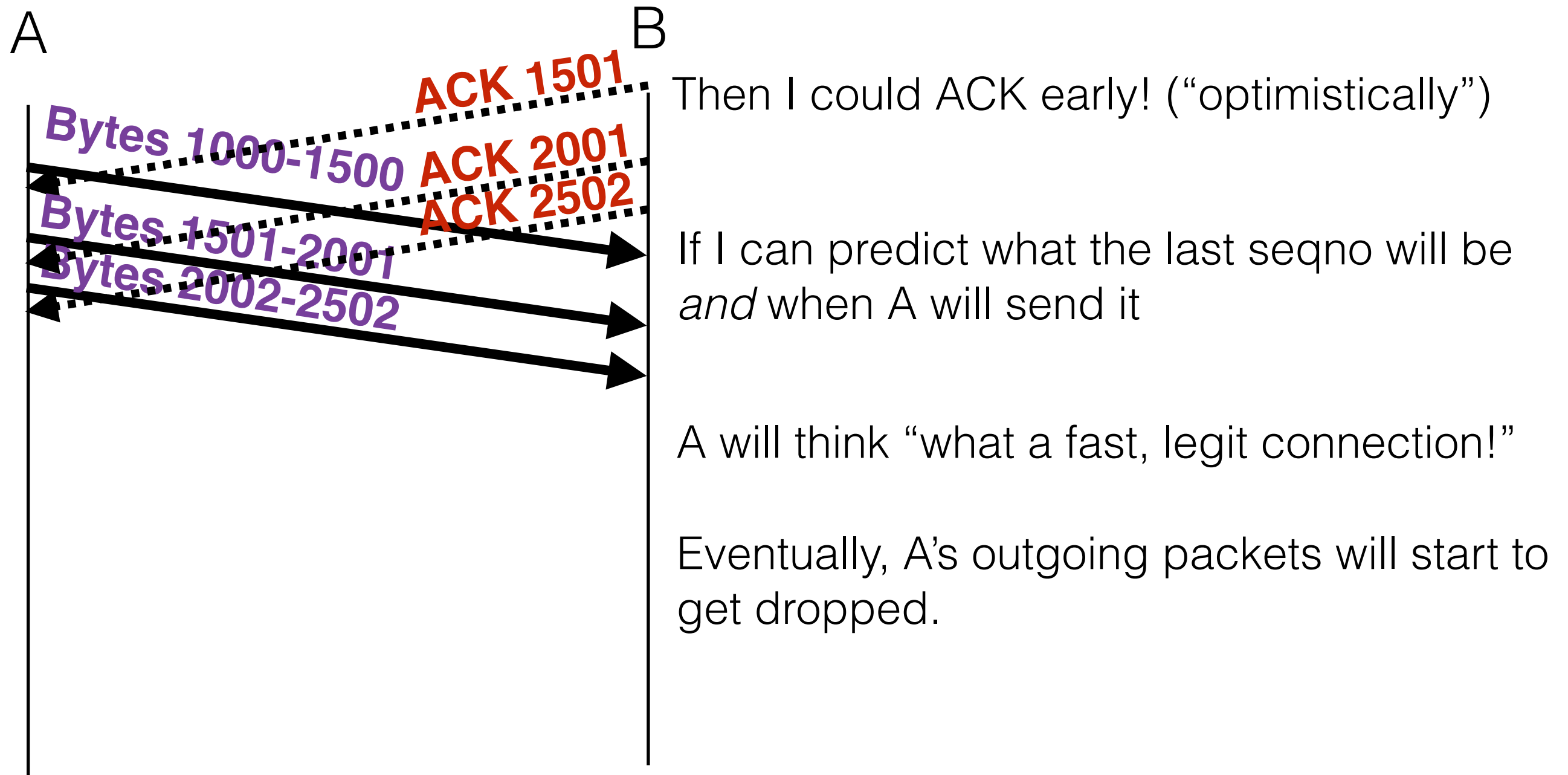
Opt-ack attack



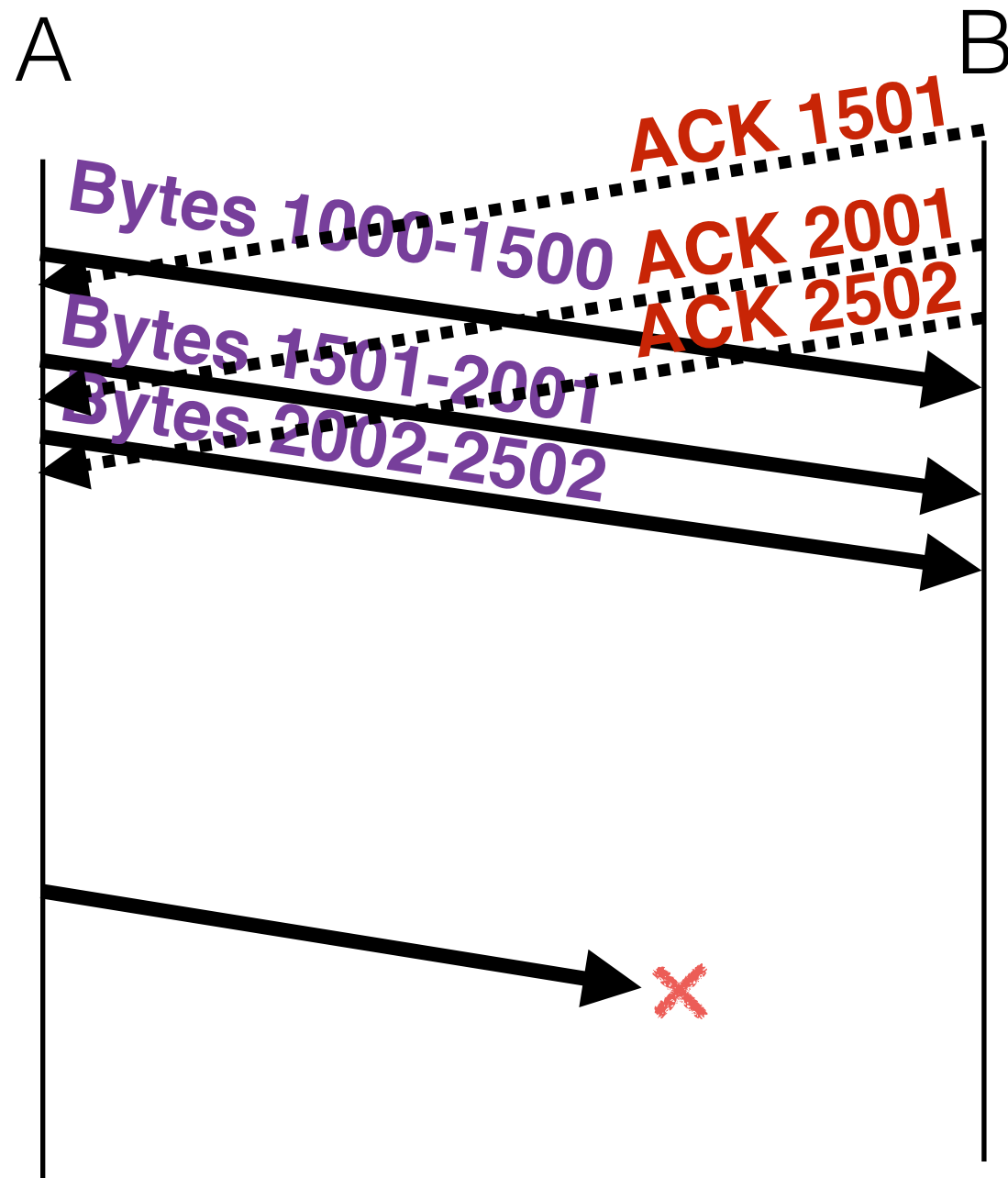
Opt-ack attack



Opt-ack attack



Opt-ack attack



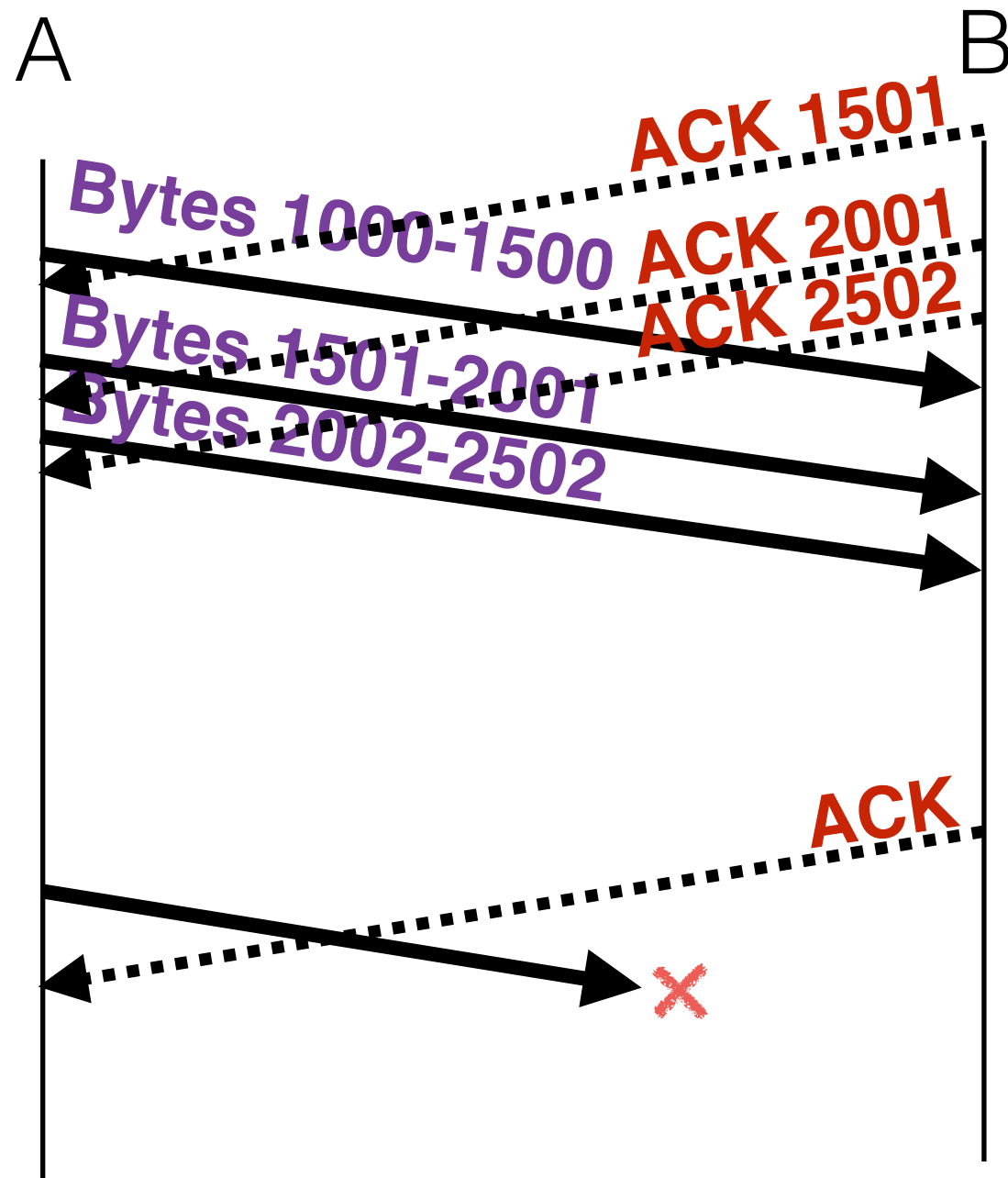
Then I could ACK early! (“optimistically”)

If I can predict what the last seqno will be
and when A will send it

A will think “what a fast, legit connection!”

Eventually, A’s outgoing packets will start to
get dropped.

Opt-ack attack



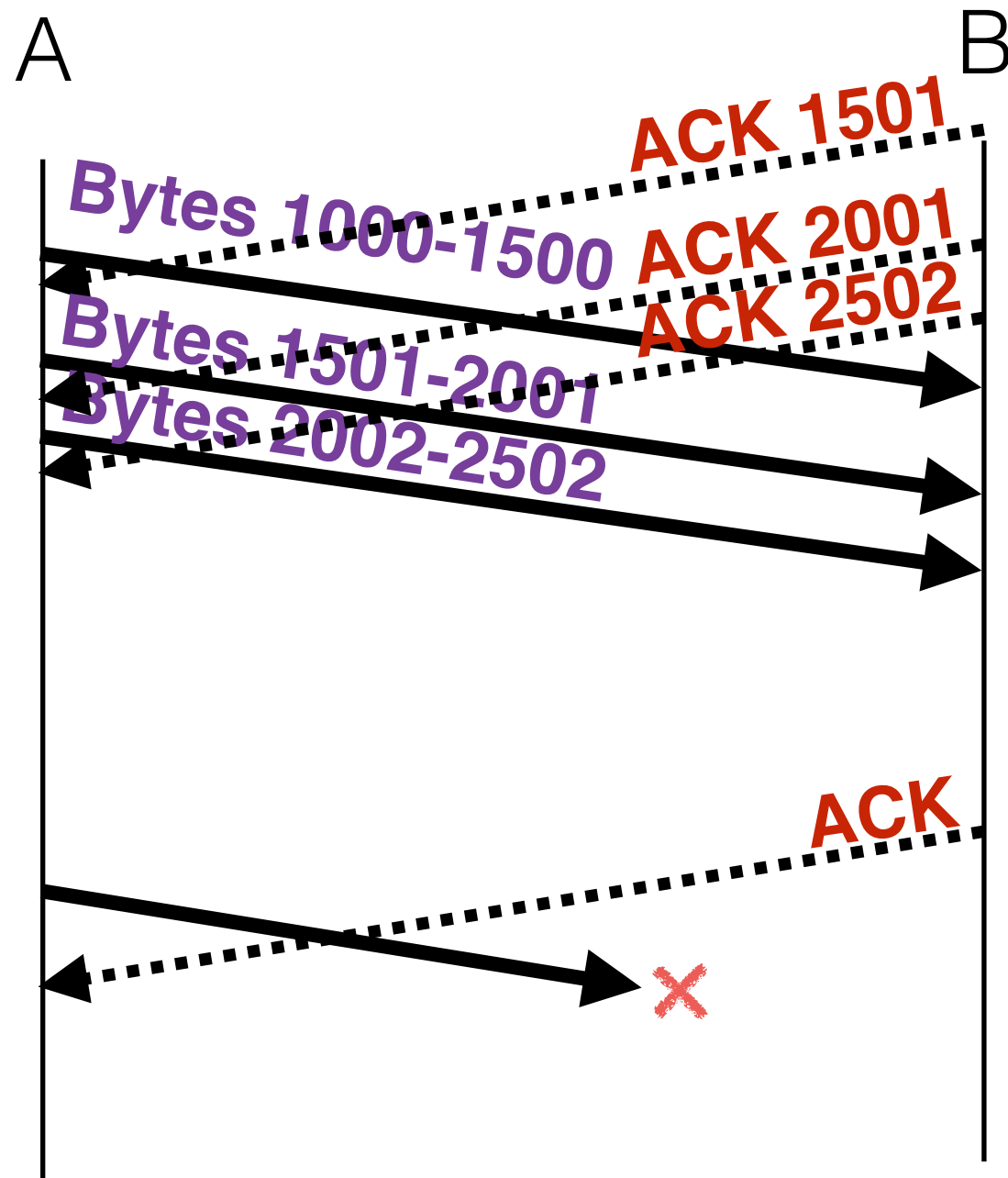
Then I could ACK early! (“optimistically”)

If I can predict what the last seqno will be
and when A will send it

A will think “what a fast, legit connection!”

Eventually, A’s outgoing packets will start to
get dropped.

Opt-ack attack



Then I could ACK early! (“optimistically”)

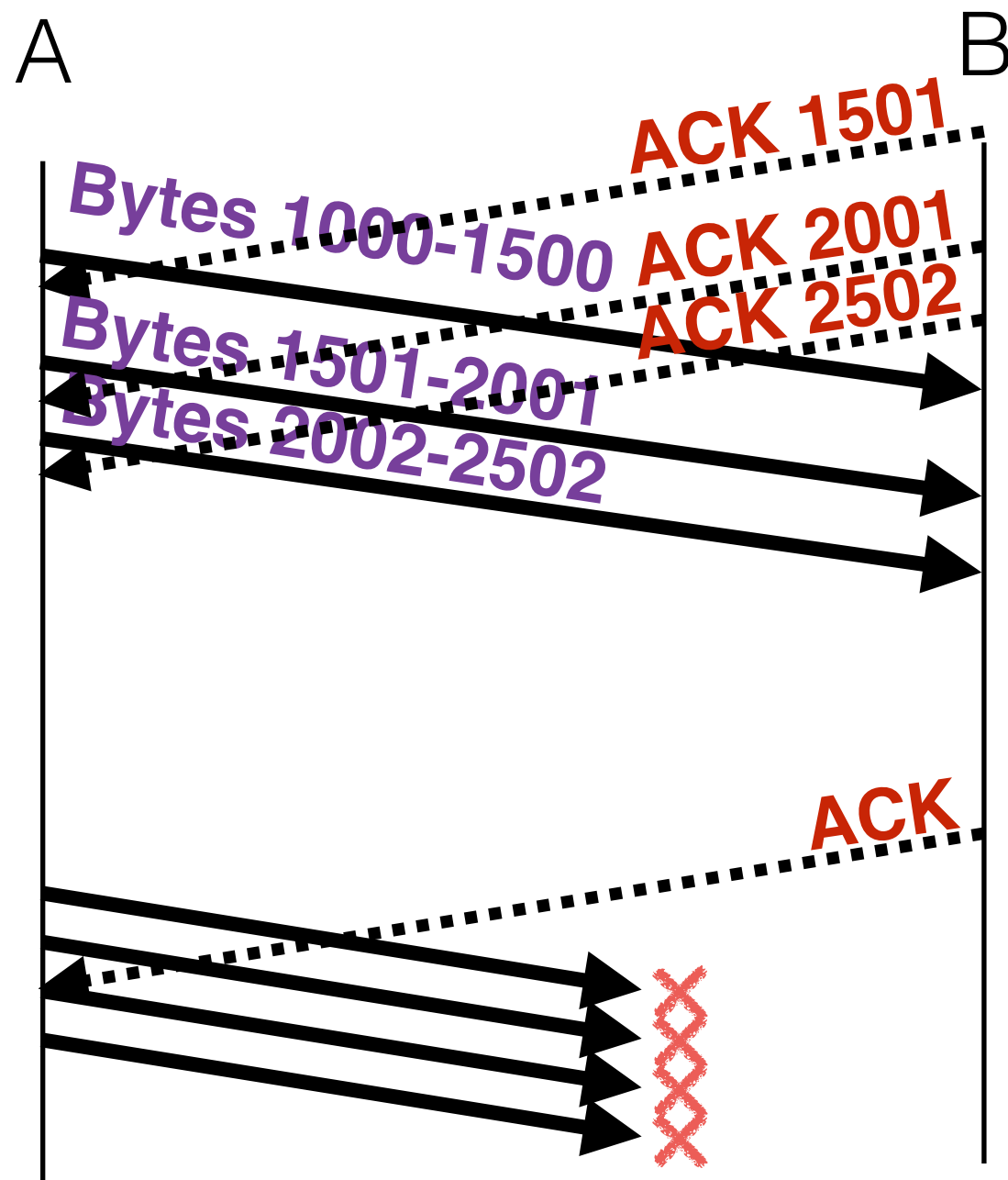
If I can predict what the last seqno will be
and when A will send it

A will think “what a fast, legit connection!”

Eventually, A’s outgoing packets will start to
get dropped.

But so long as I keep ACKing correctly, it
doesn’t matter.

Opt-ack attack



Then I could ACK early! (“optimistically”)

If I can predict what the last seqno will be
and when A will send it

A will think “what a fast, legit connection!”

Eventually, A’s outgoing packets will start to
get dropped.

But so long as I keep ACKing correctly, it
doesn’t matter.

Amplification

- The big deal with this attack is its *Amplification Factor*
 - Attacker sends x bytes of data, causing the victim to send many more bytes of data in response
 - Recent examples: NTP, DNSSEC
- Amplified in TCP due to cumulative ACKs
 - “ACK x ” says “I’ve seen all bytes up to but not including x ”

Opt-ack's amplification factor

- Max bytes sent by victim per ACK:
- Max ACKs attacker can send per second:

Opt-ack's amplification factor

- Max bytes sent by victim per ACK:

$$\begin{array}{c} \text{Packets sent per ACK} \\ \hline \frac{\text{Max window size}}{\text{MSS}} \end{array} \times \begin{array}{c} \text{Bytes per packet} \\ (14 + 40 + \text{MSS}) \end{array}$$

Ethernet TCP/IP Payload

- Max ACKs attacker can send per second:

Opt-ack's amplification factor

- Max bytes sent by victim per ACK:

$$\begin{array}{c} \text{Packets sent per ACK} \\ \hline \frac{\text{Max window size}}{\text{MSS}} \end{array} \times \begin{array}{c} \text{Bytes per packet} \\ (14 + 40 + \text{MSS}) \end{array}$$

Ethernet TCP/IP Payload

- Max ACKs attacker can send per second:

$$\frac{\text{Attacker bandwidth (bytes/sec)}}{(14 + 40)}$$

Size of ACK packet

Opt-ack's amplification factor

- Boils down to max window size and MSS
 - Default max window size: 65,536
 - Default MSS: 536
- Default amp factor: $65536 * (1/536 + 1/54) \sim \mathbf{1336x}$
- Window scaling lets you increase this by a factor of 2^{14}
- Window scaling amp factor: $\sim 1336 * 2^{14} \sim \mathbf{22M}$
- Using minimum MSS of 88: $\sim \mathbf{32M}$

Opt-ack defenses

- Is there a way we could defend against opt-ack in a way that is still compatible with existing implementations of TCP?
- An important goal in networking is *incremental deployment*: ideally, we should be able to benefit from a system/modification when even a subset of hosts deploy it.