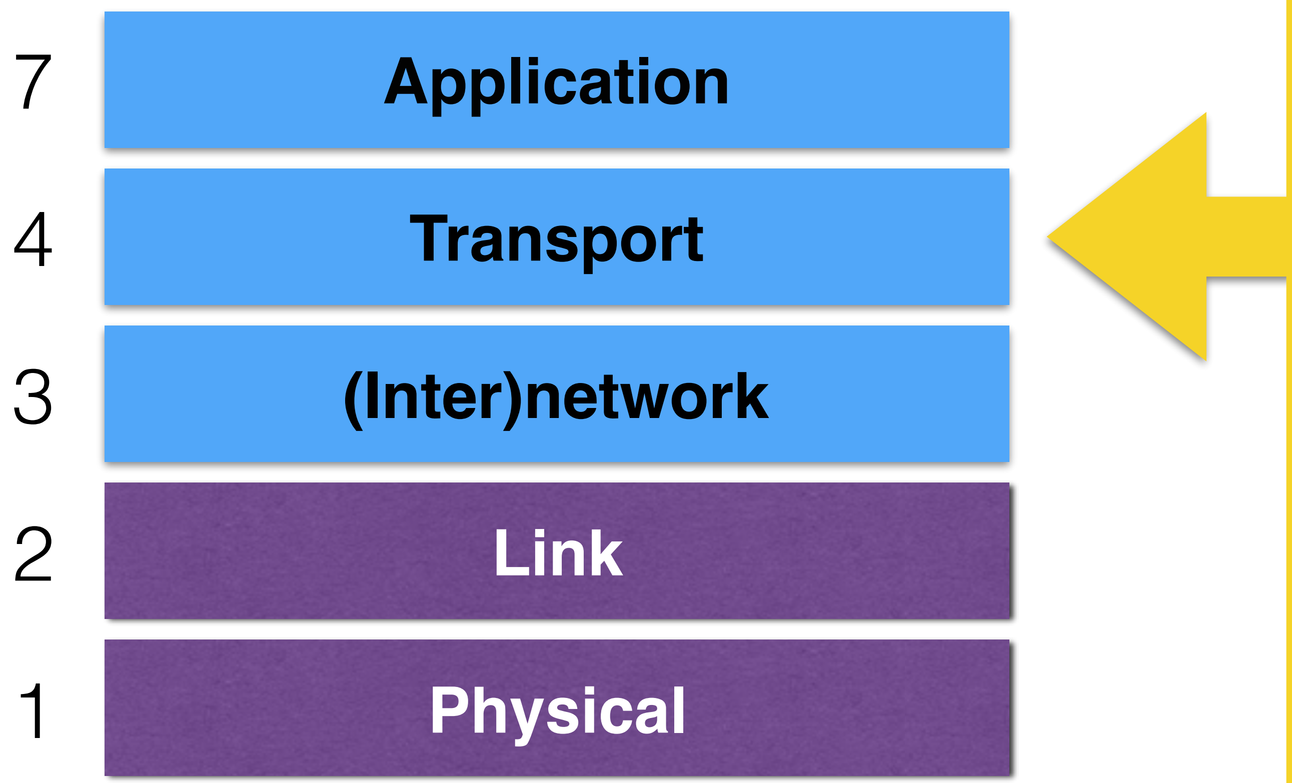
Transport layer attacks

Slides from

- Dave Levin 414-spring2016

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

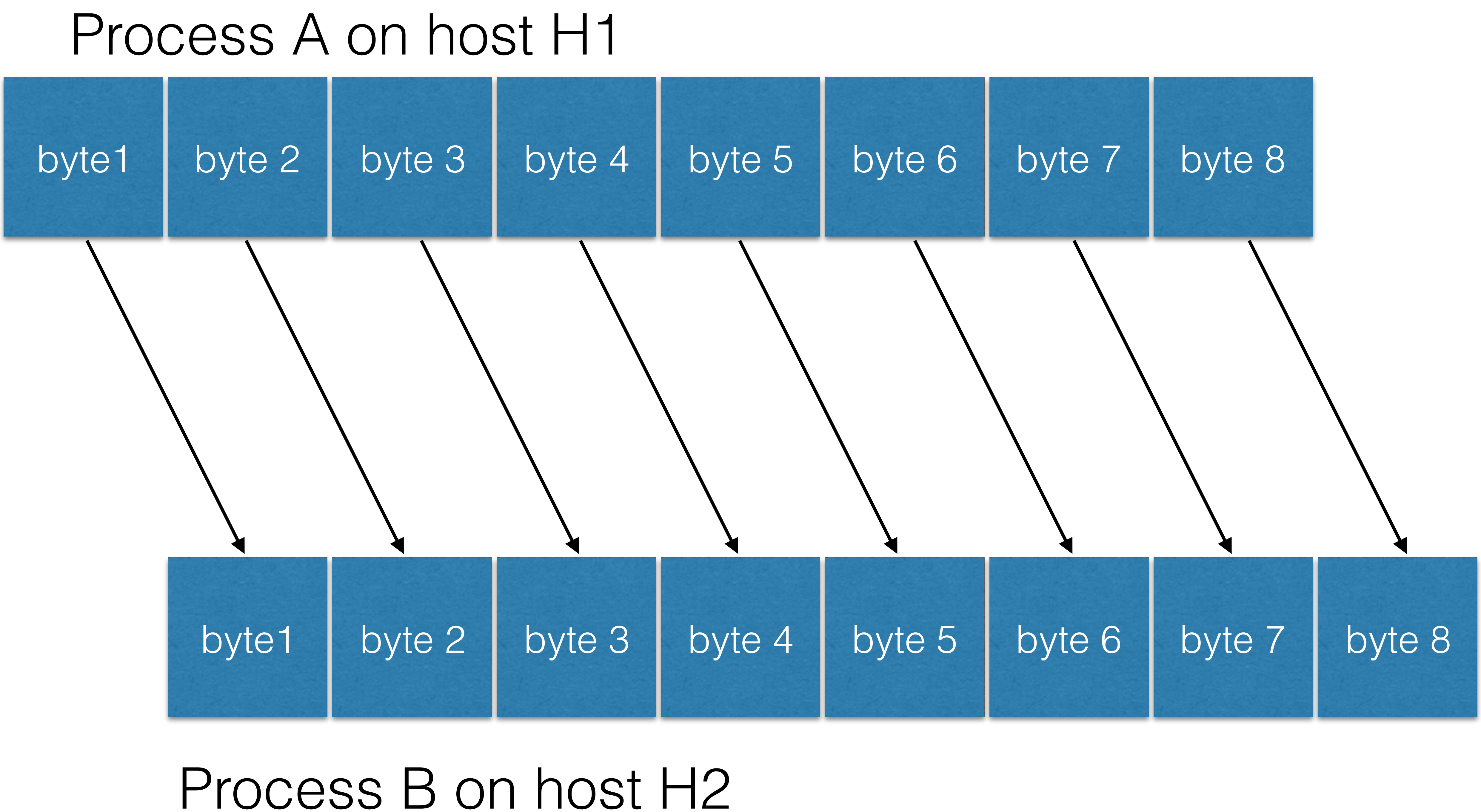
- Given best-effort deliver, the goal is to ensure *reliability*
  - All packets are delivered to applications
  - … in order
  - … unmodified (with reasonably high probability)
- Must robustly detect and retransmit lost data

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)

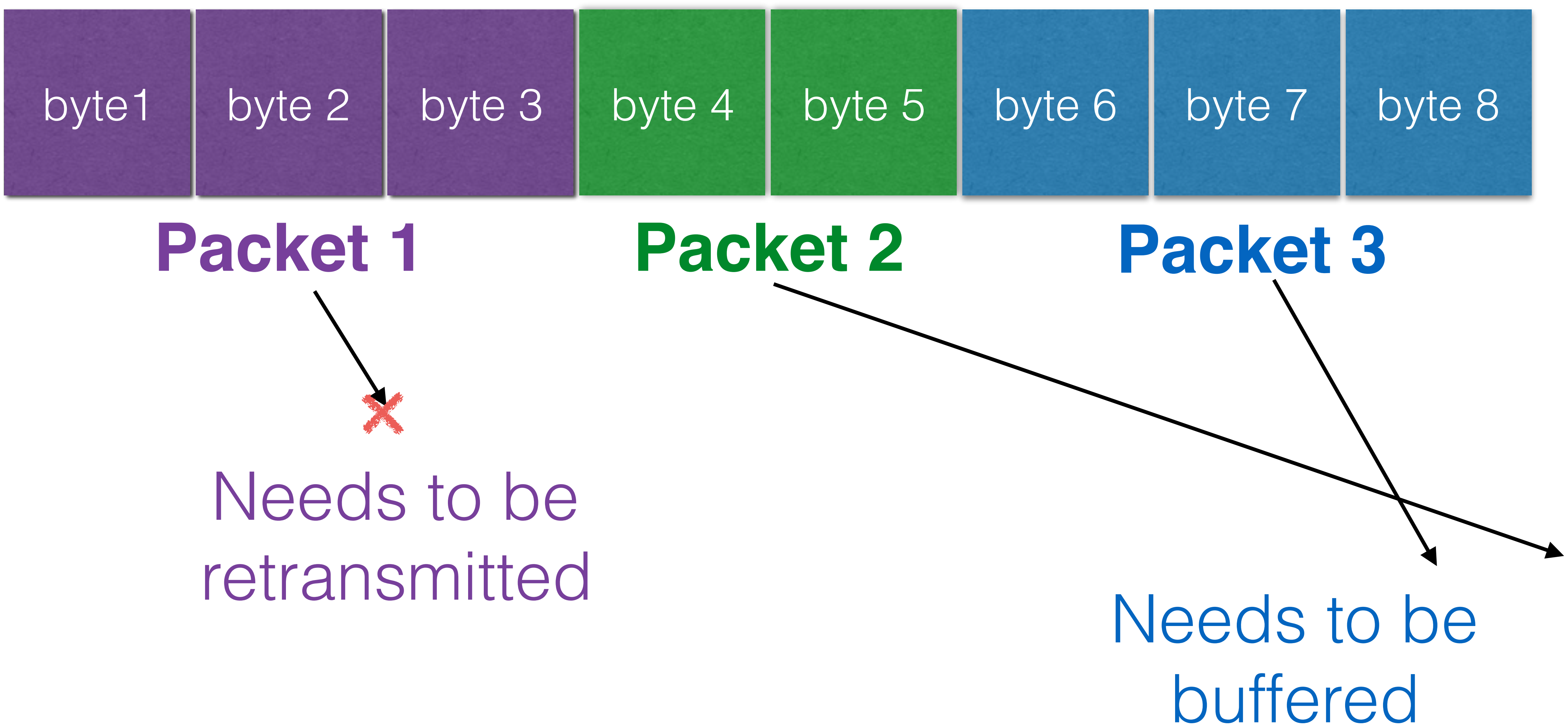
TCP bytestream service

Abstraction: Each *byte* reliably delivered in order



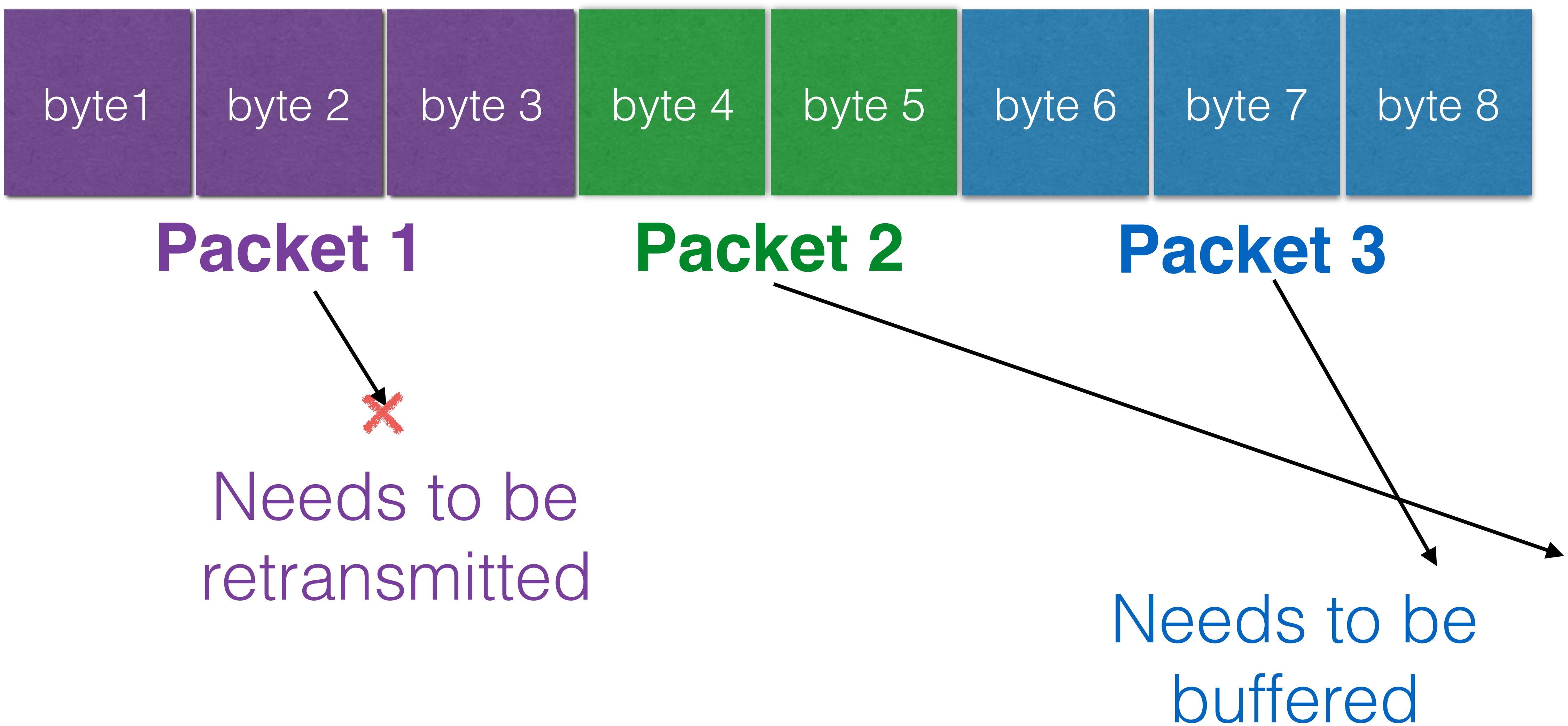
TCP bytestream service

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



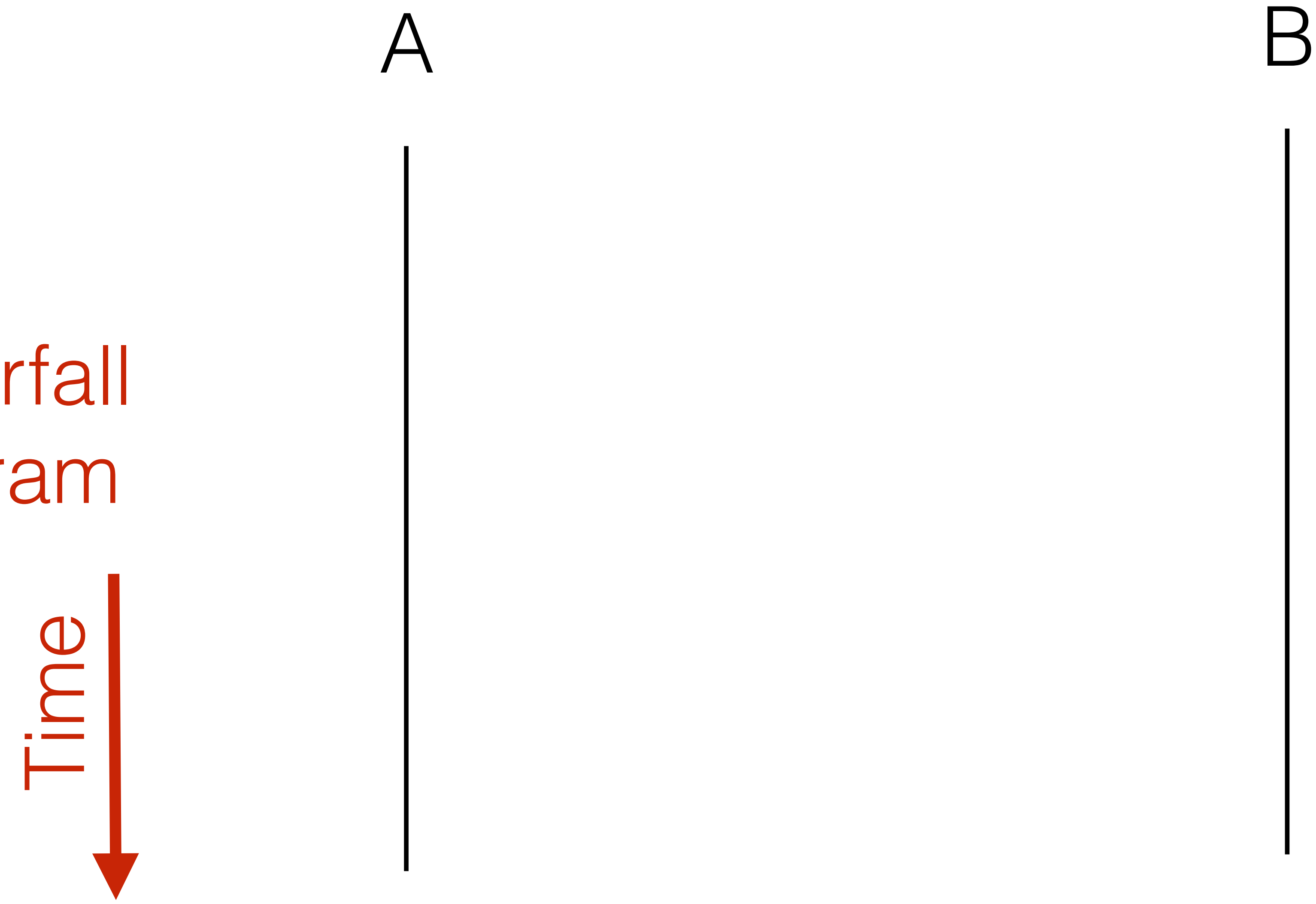
TCP bytestream service

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



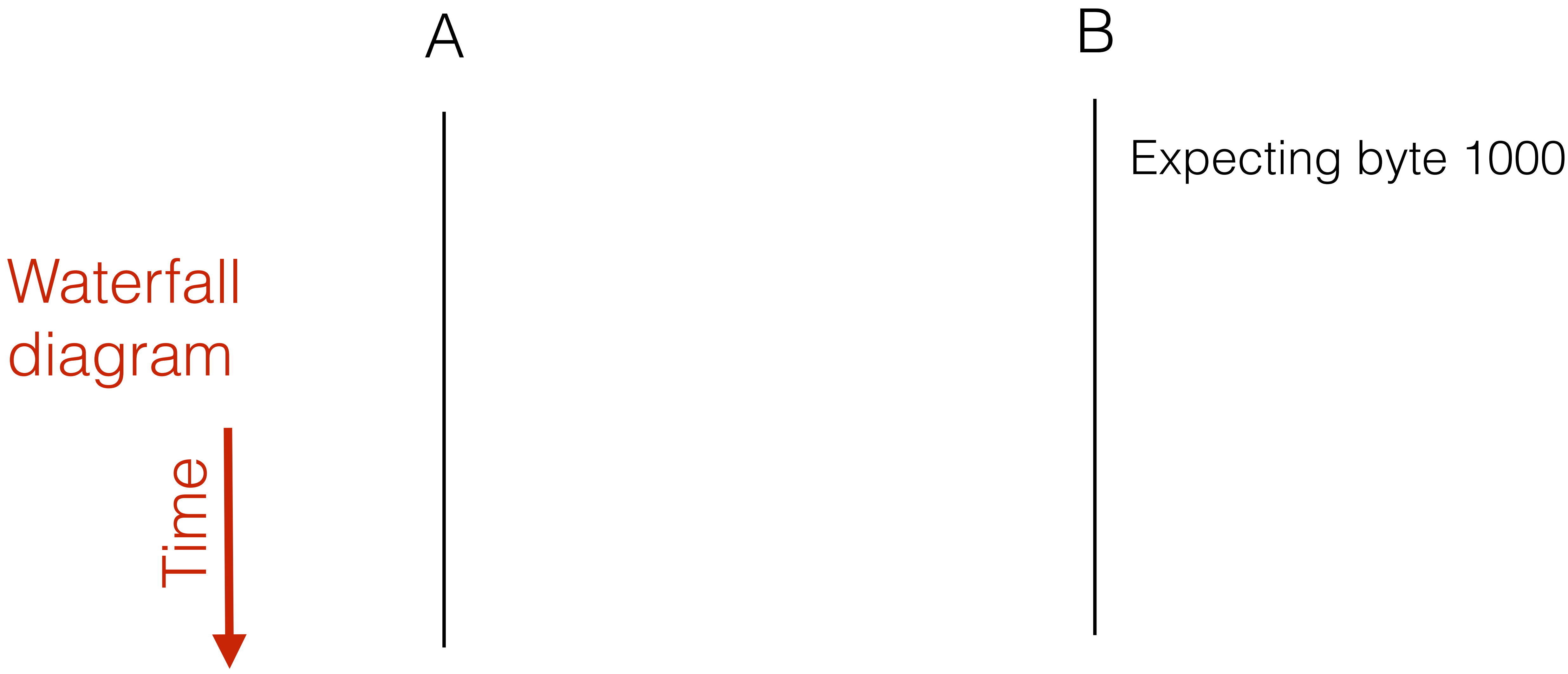
TCP's first job: achieve the abstraction while hiding the reality from the application

How does TCP achieve reliability?

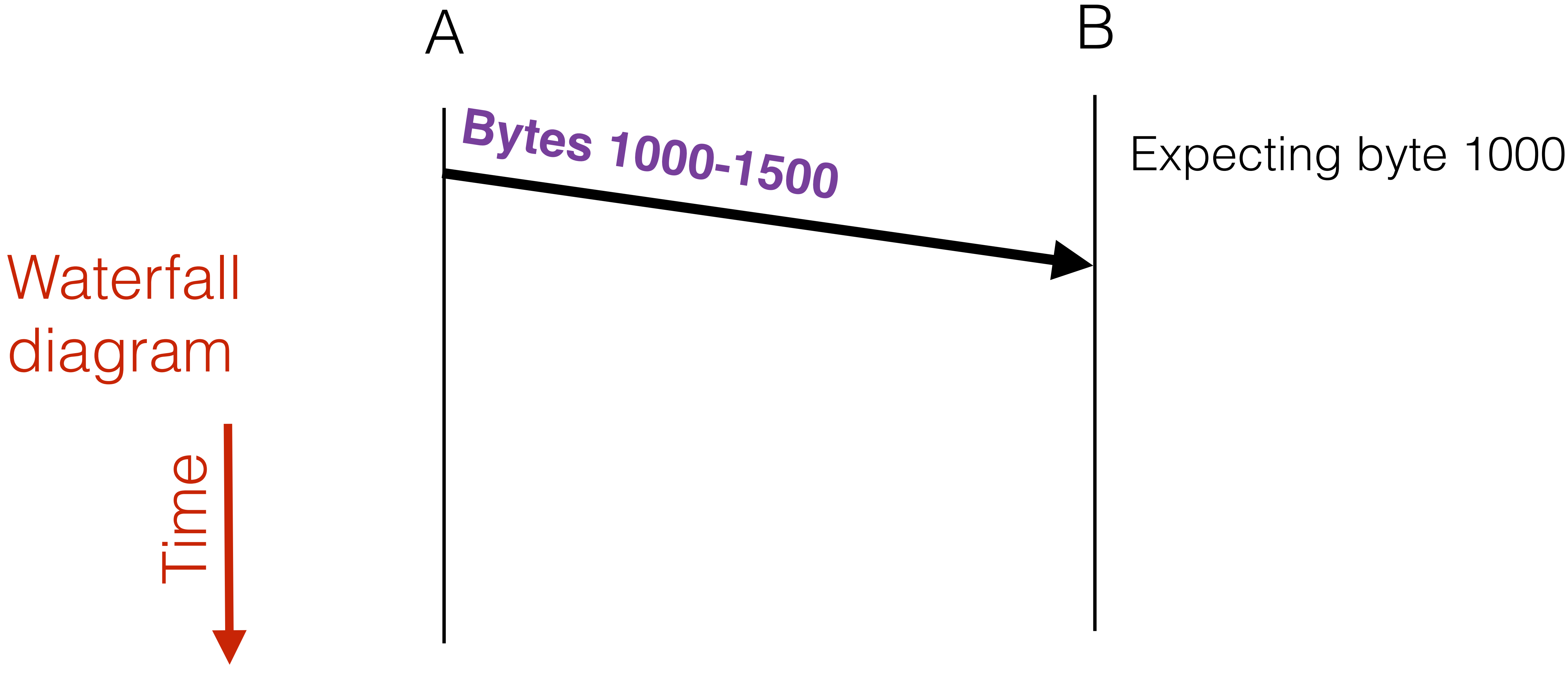


Waterfall diagram

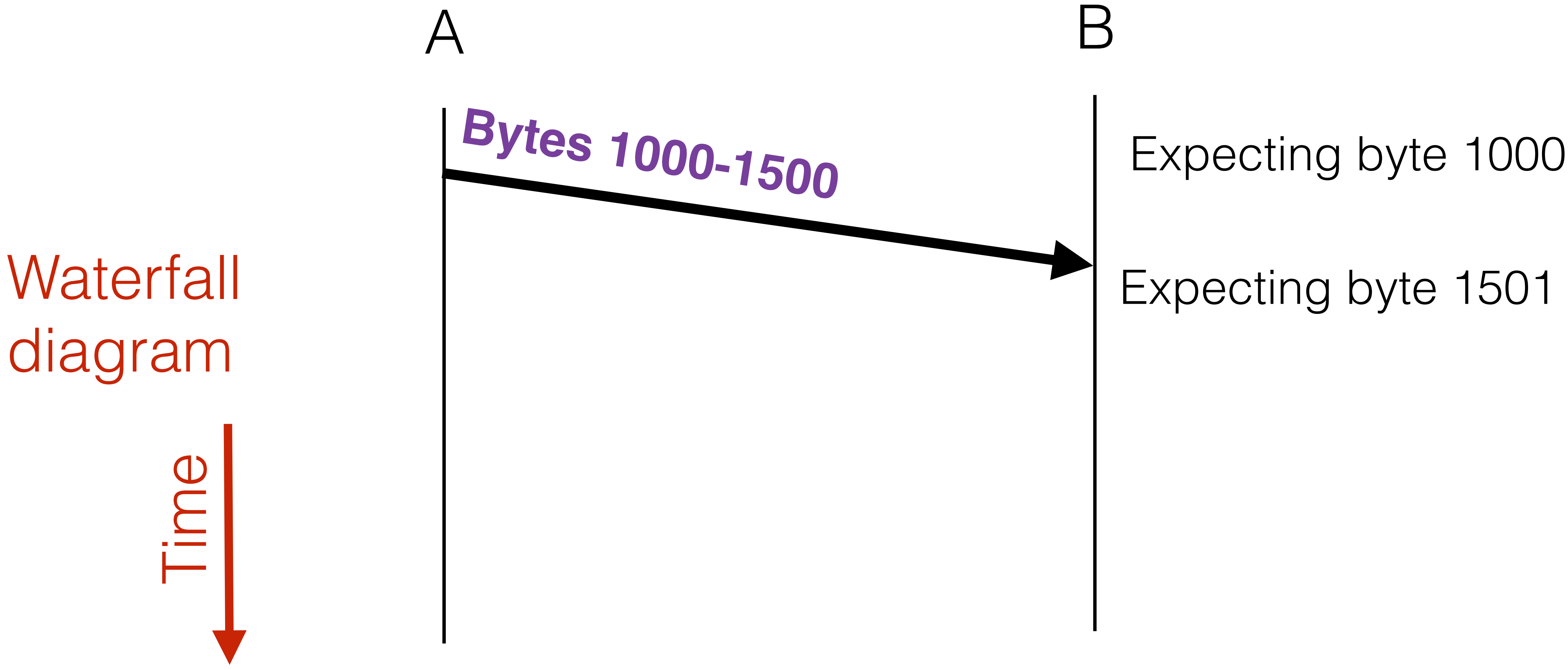
How does TCP achieve reliability?



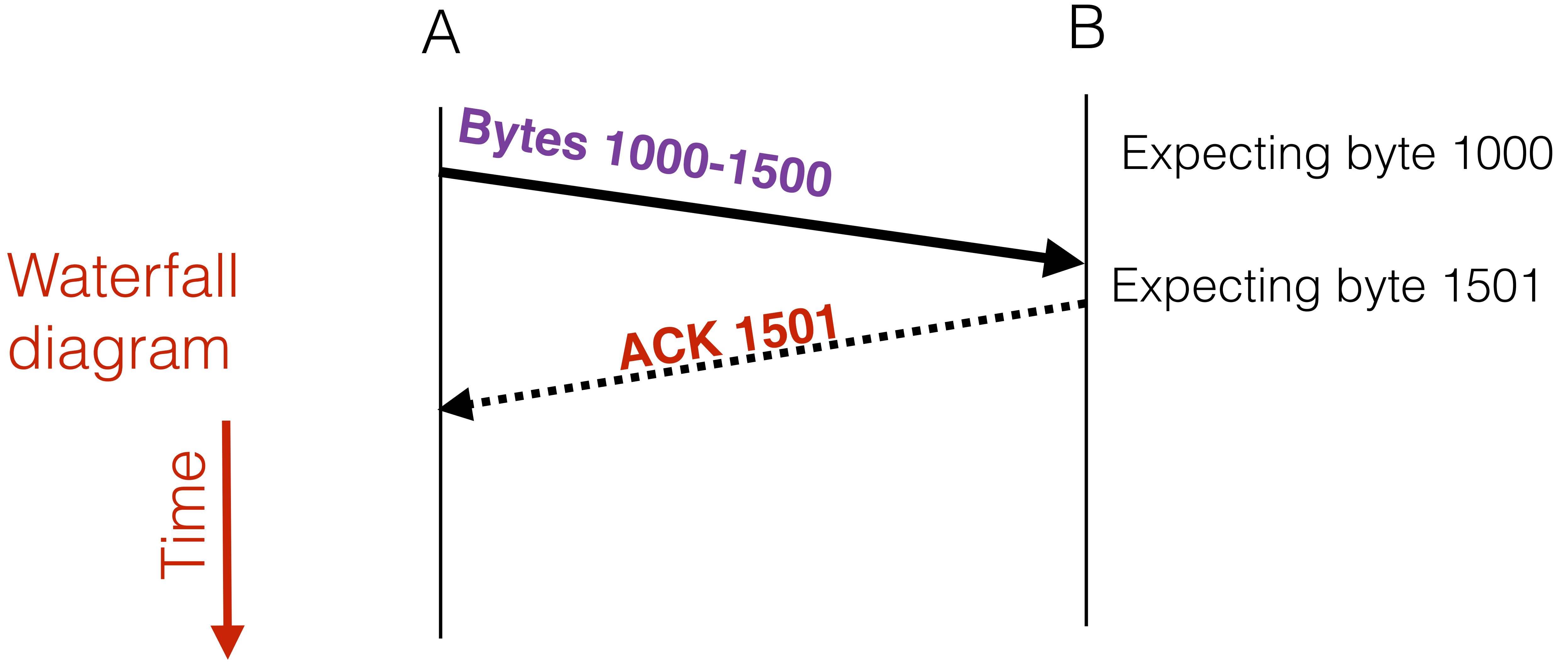
How does TCP achieve reliability?



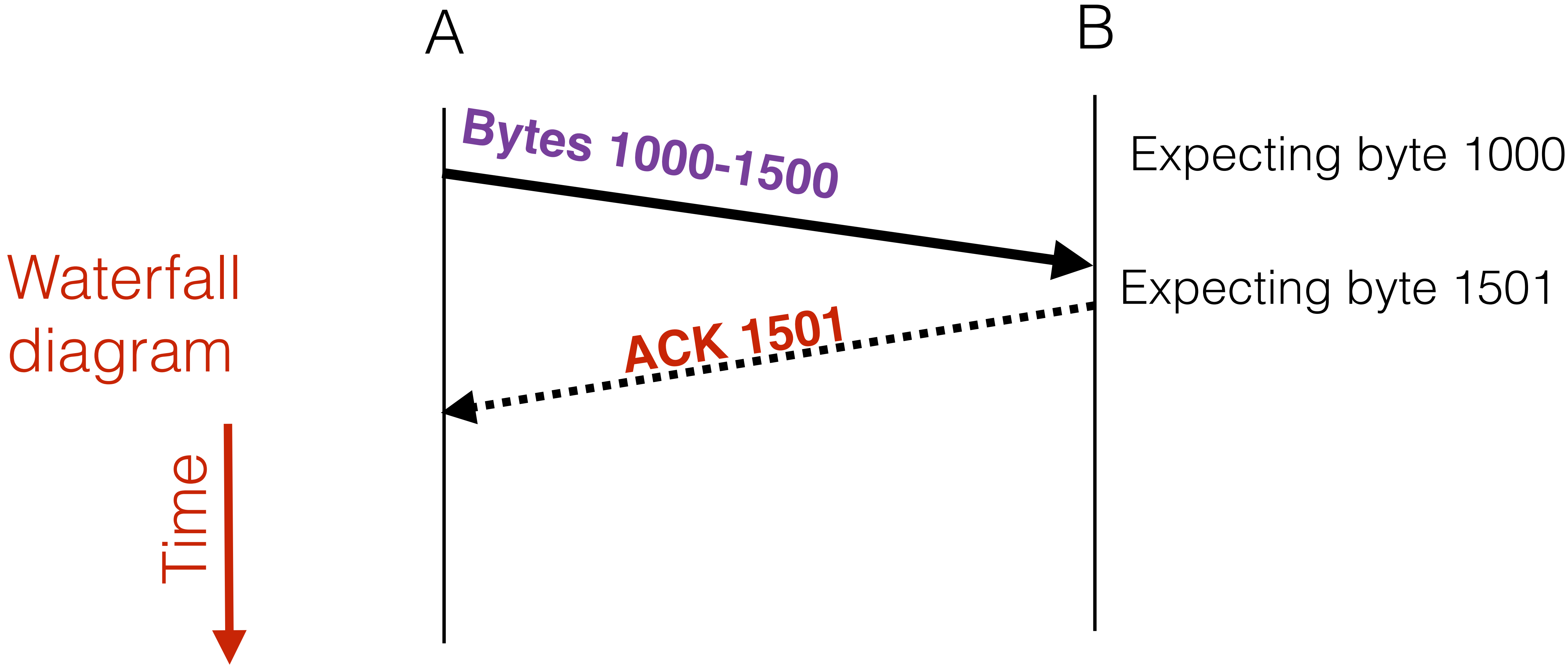
How does TCP achieve reliability?



How does TCP achieve reliability?

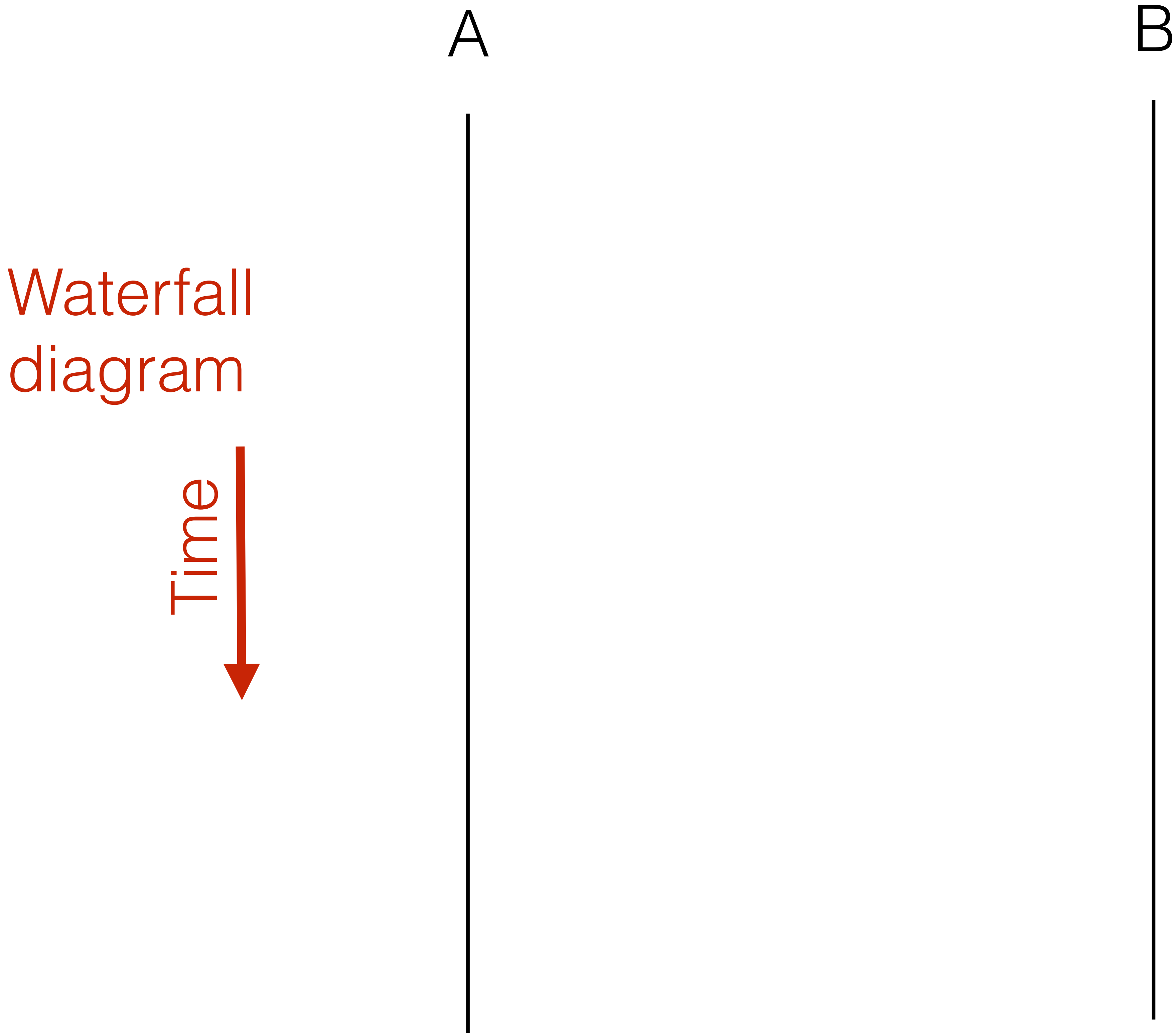


How does TCP achieve reliability?

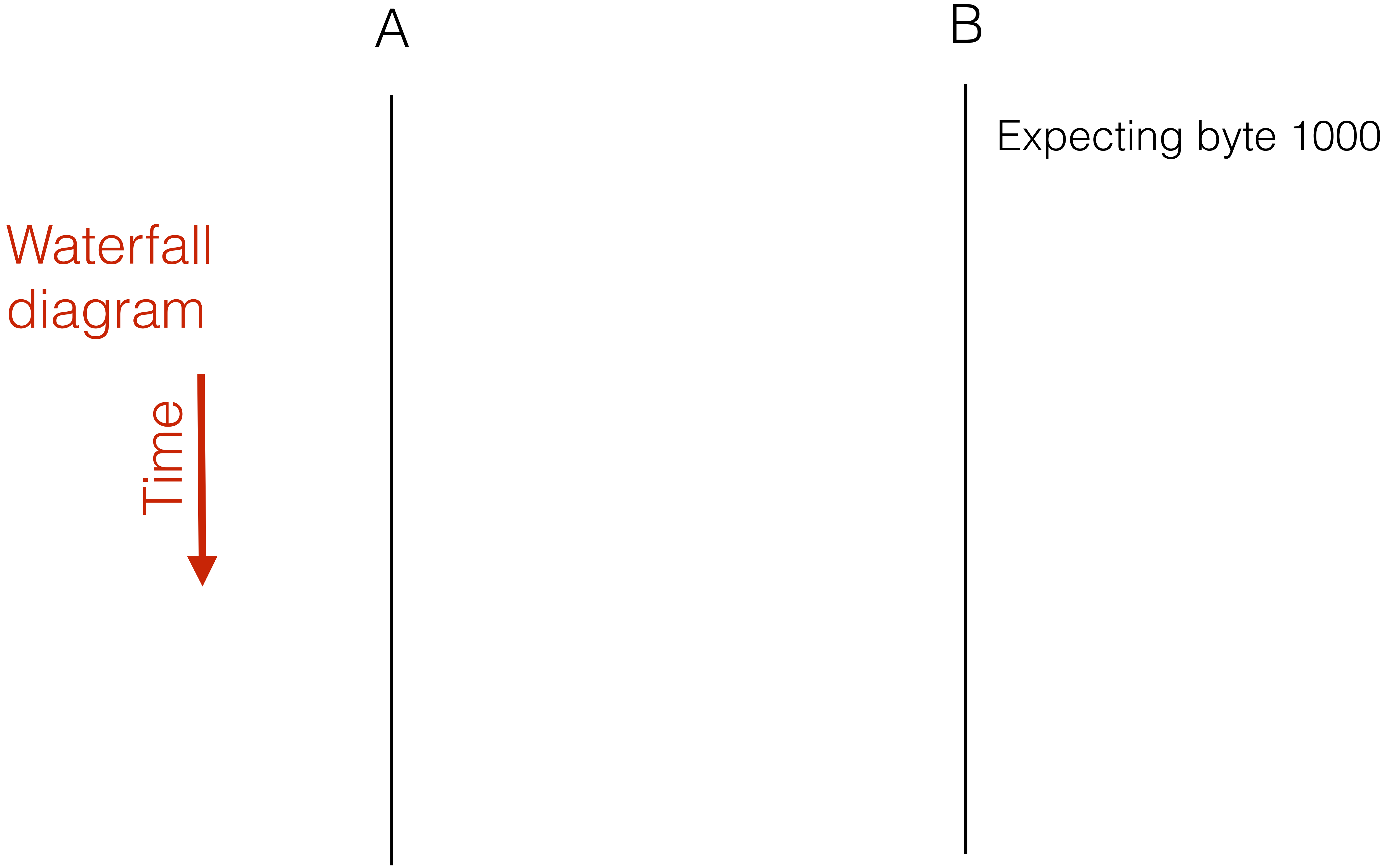


Reliability through acknowledgments
to determine whether something was received.

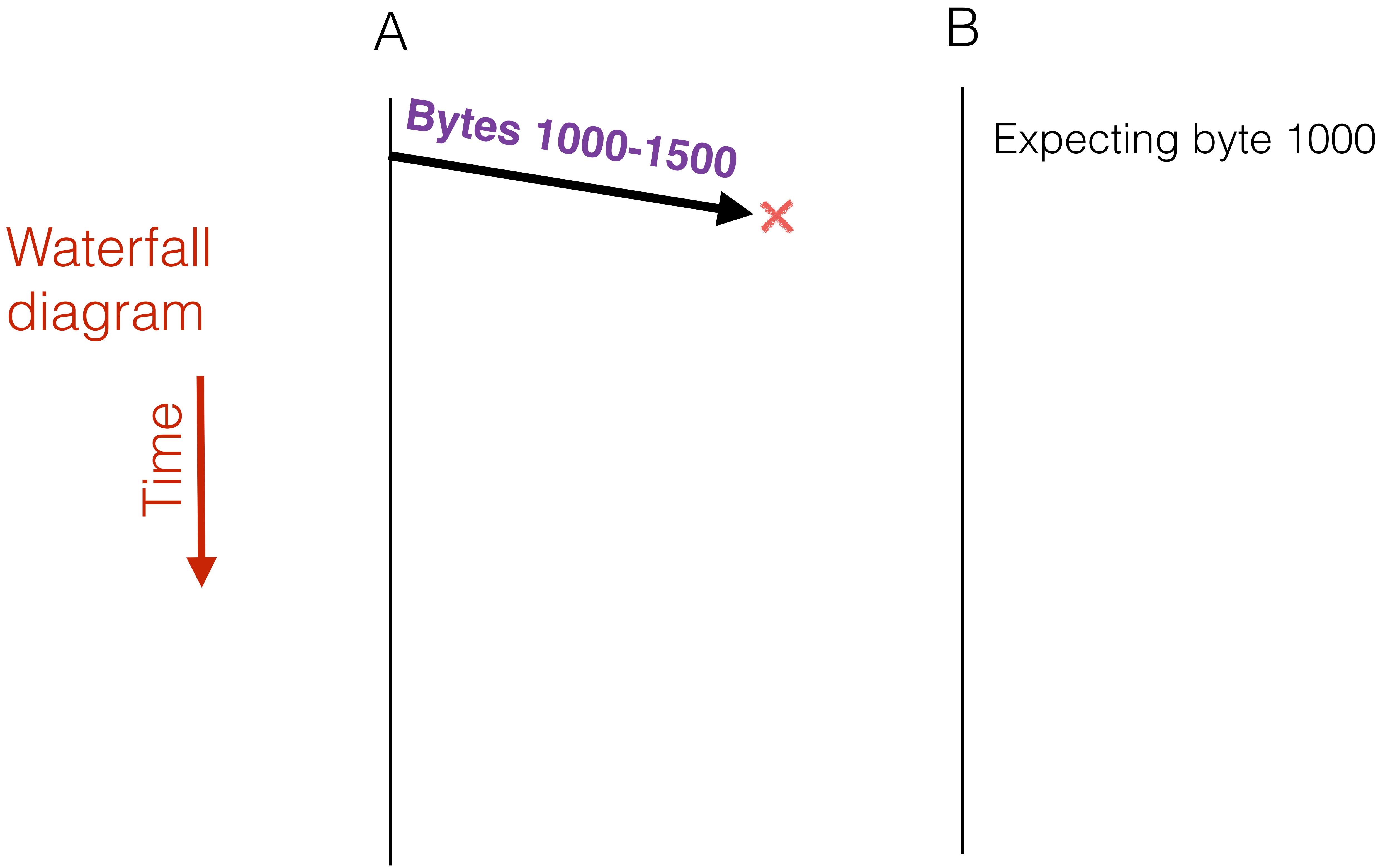
How does TCP achieve reliability?



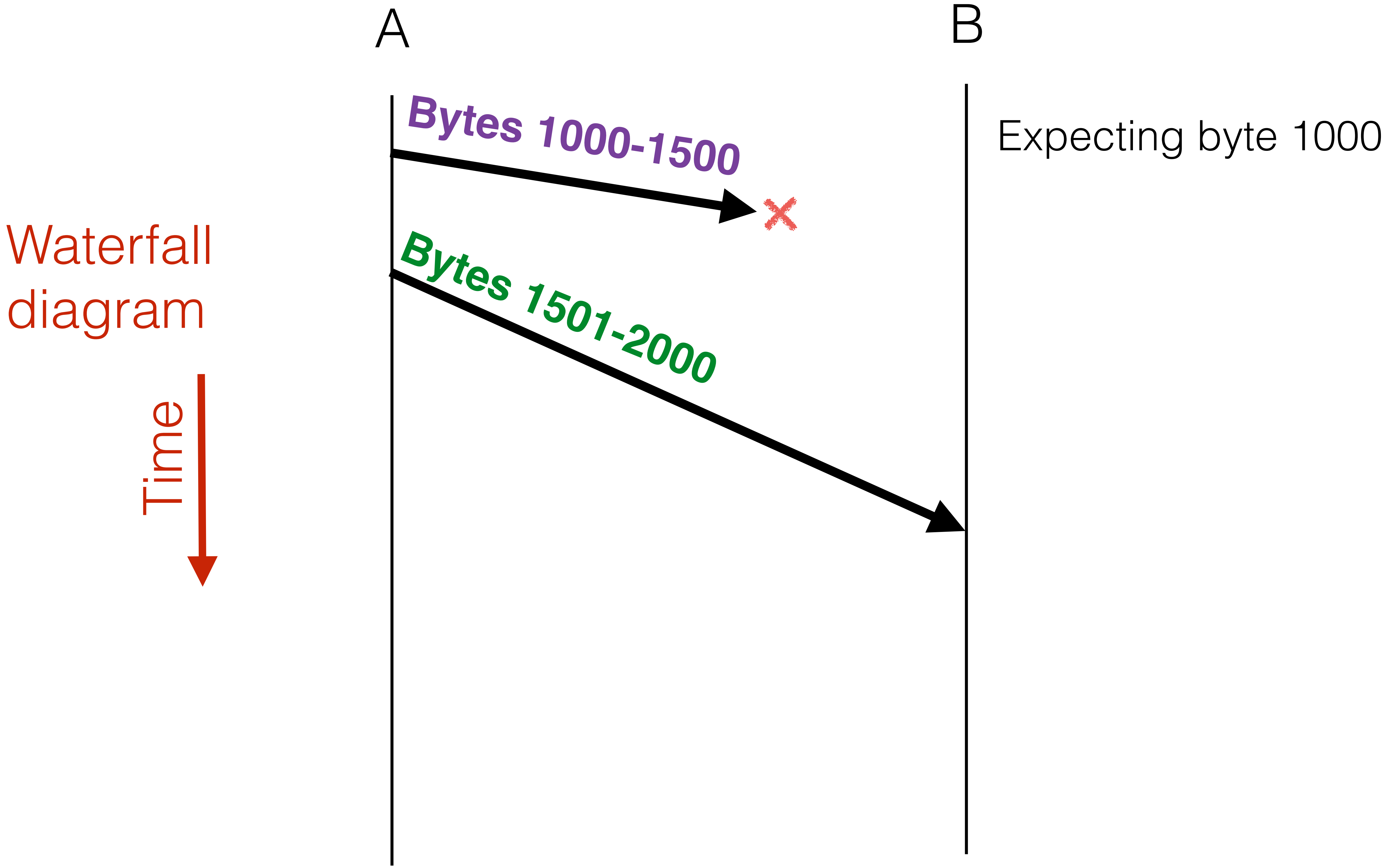
How does TCP achieve reliability?



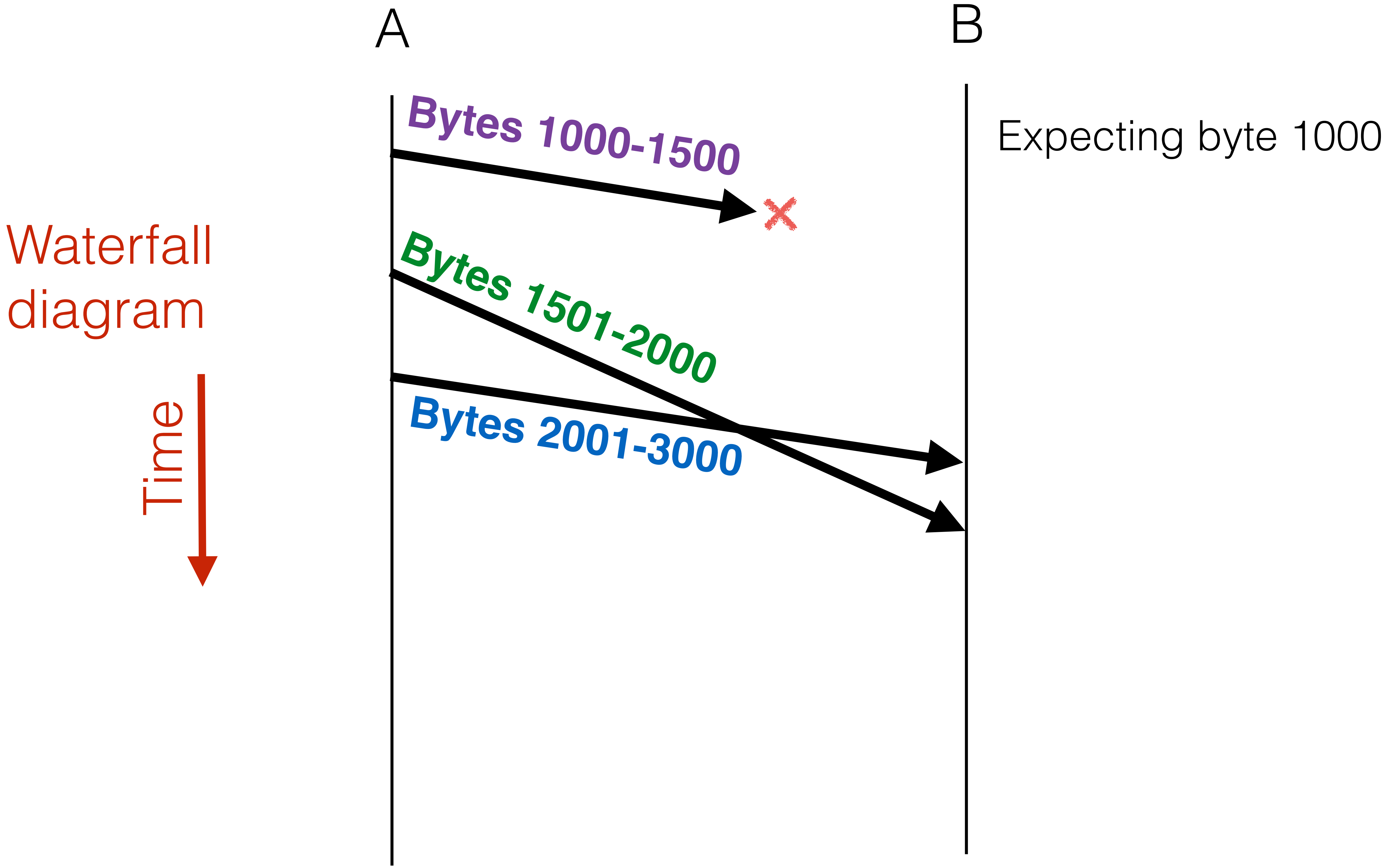
How does TCP achieve reliability?



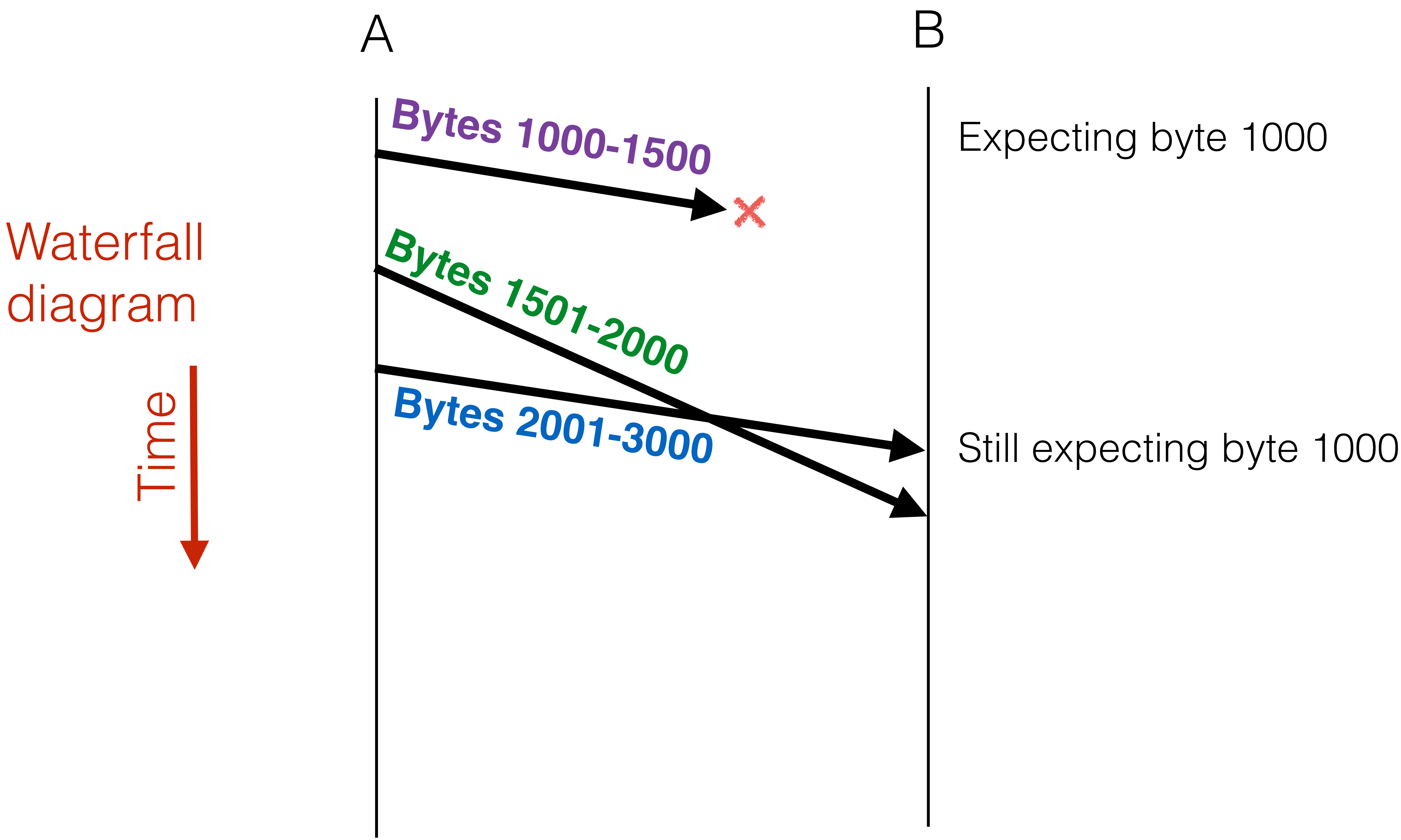
How does TCP achieve reliability?



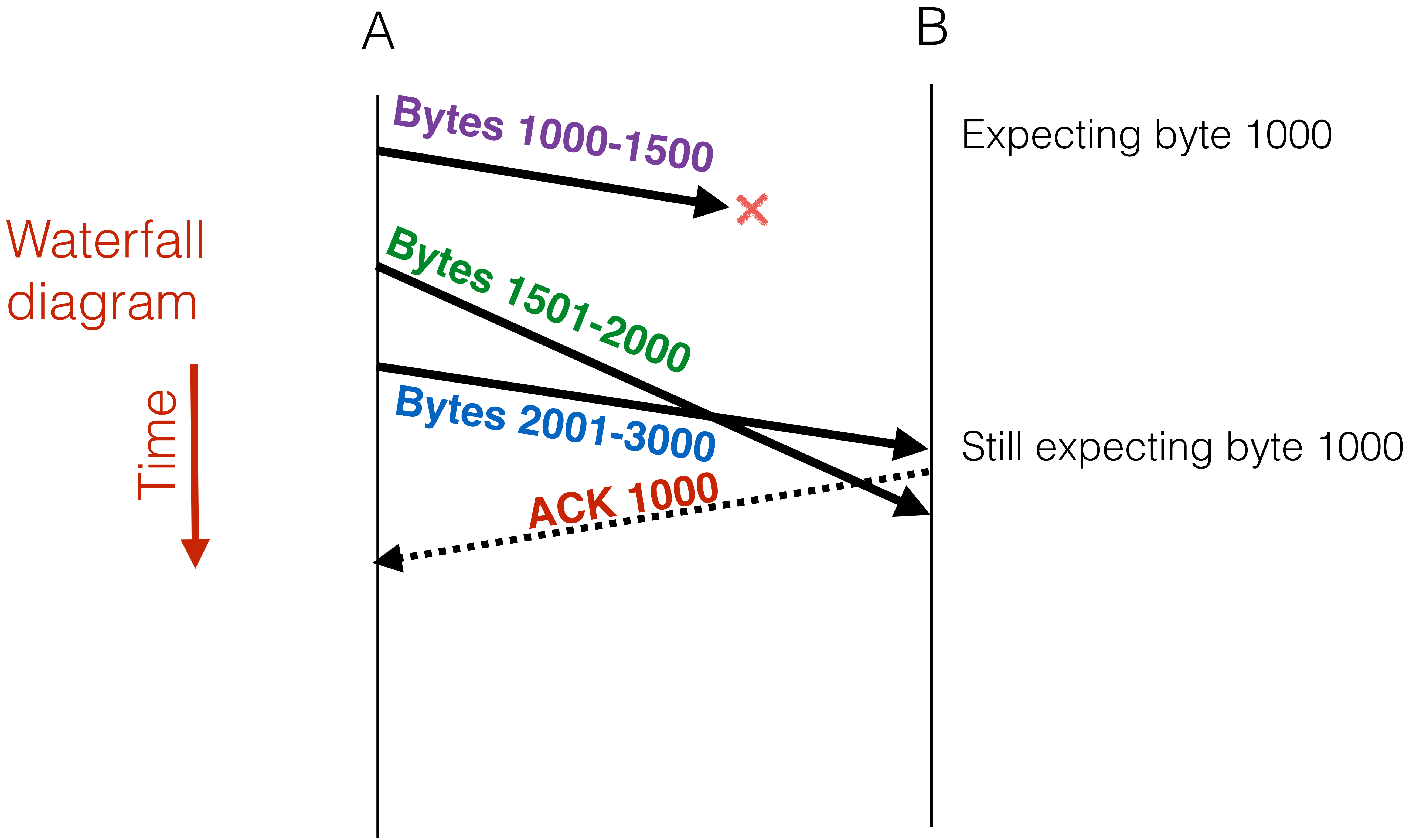
How does TCP achieve reliability?



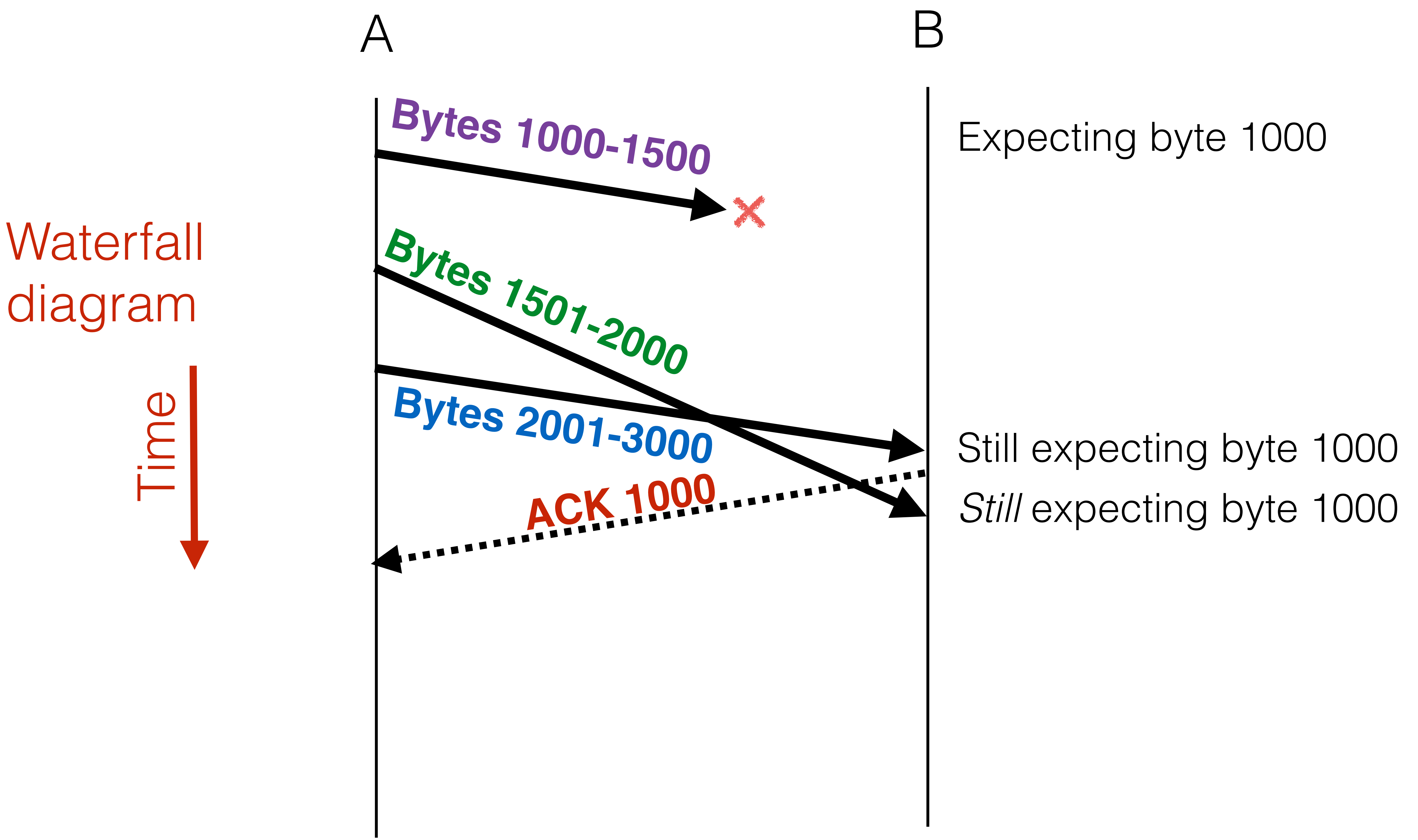
How does TCP achieve reliability?



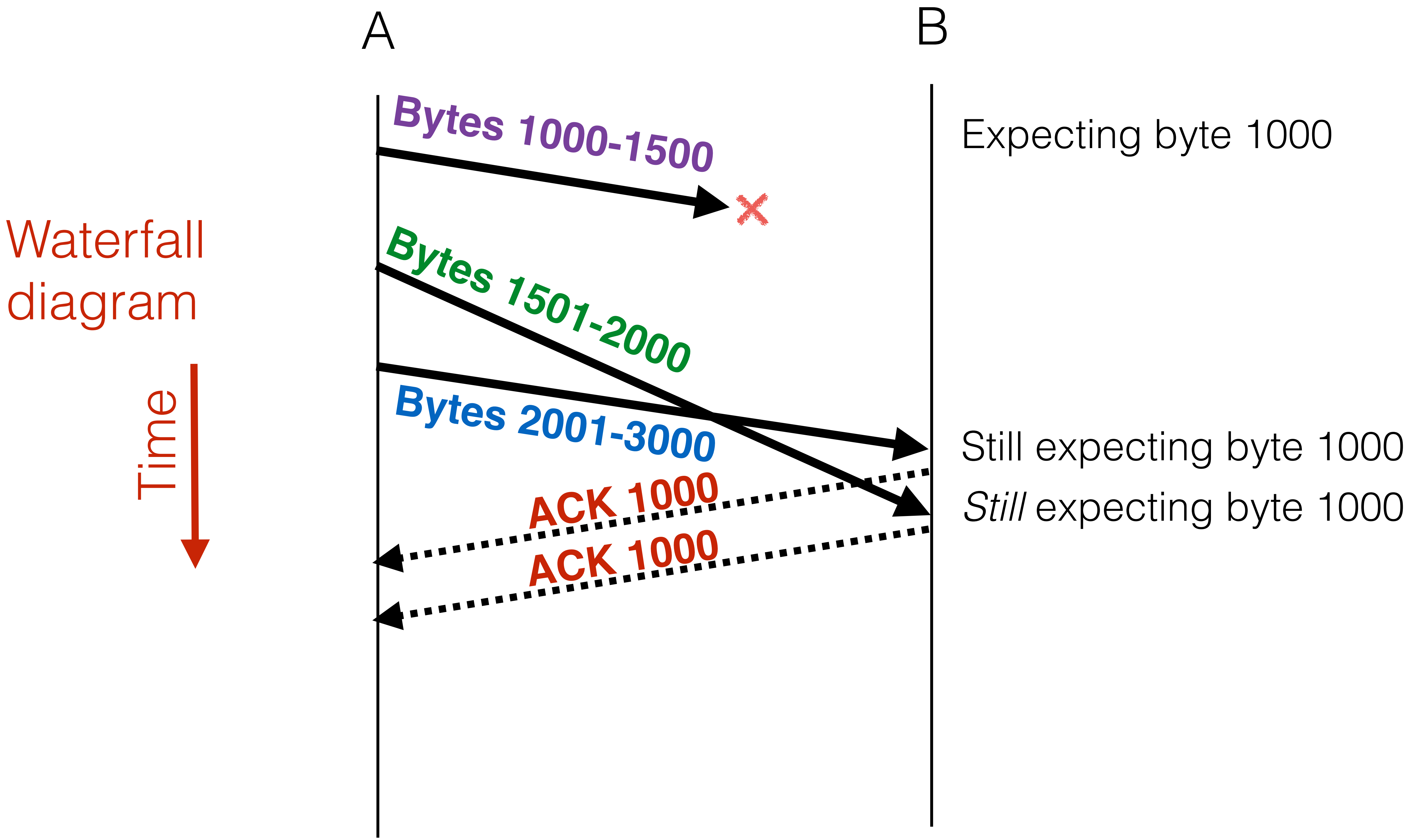
How does TCP achieve reliability?



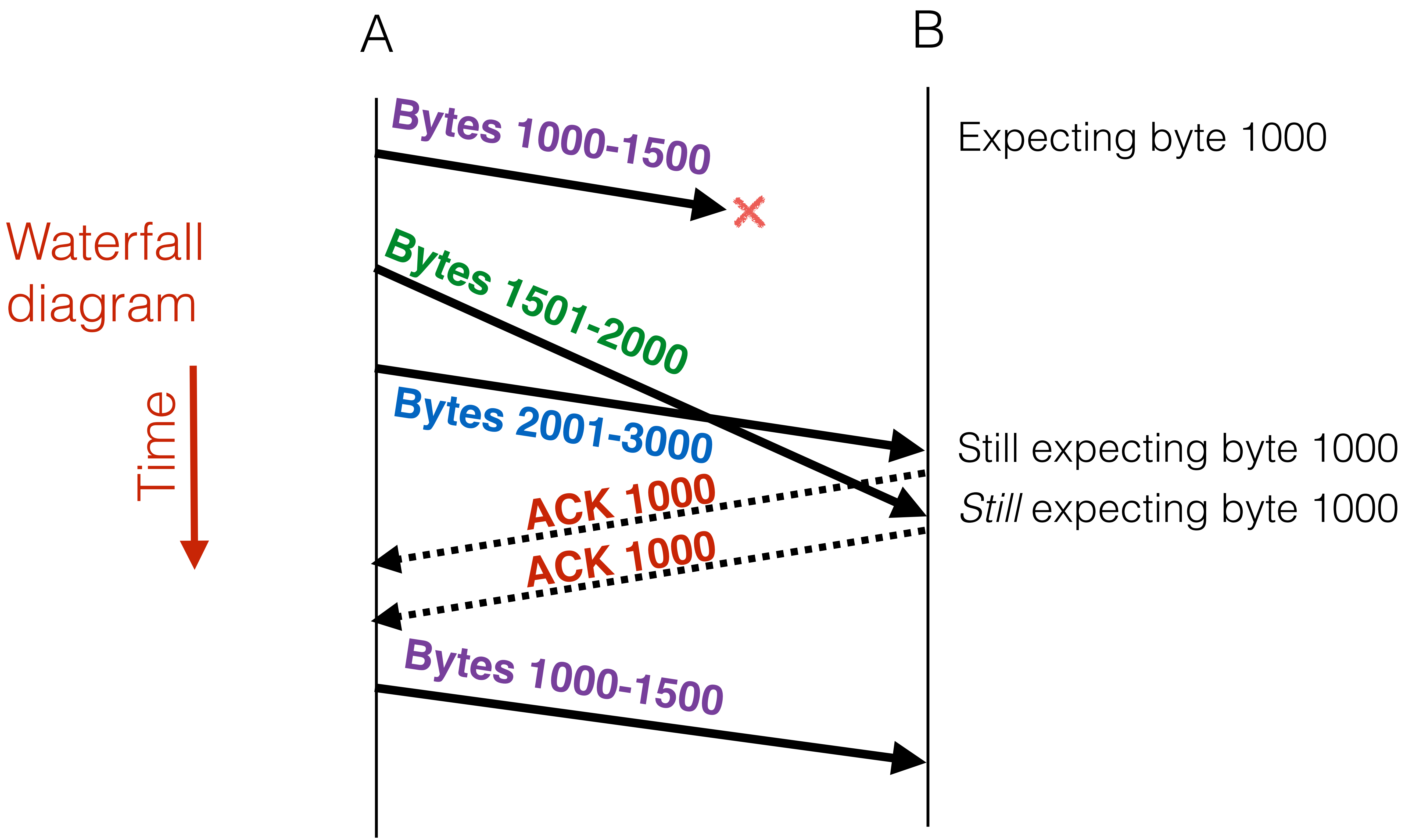
How does TCP achieve reliability?



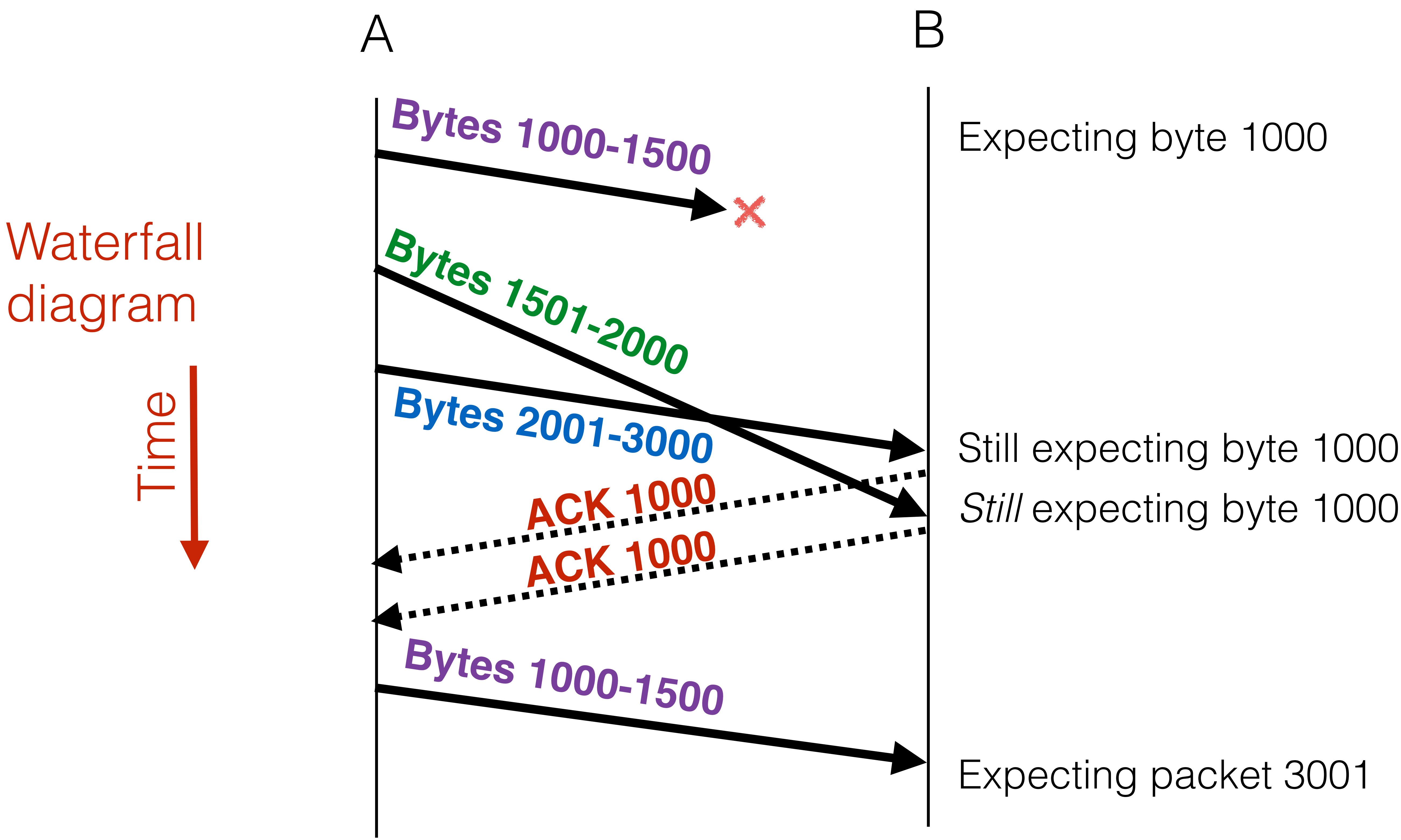
How does TCP achieve reliability?



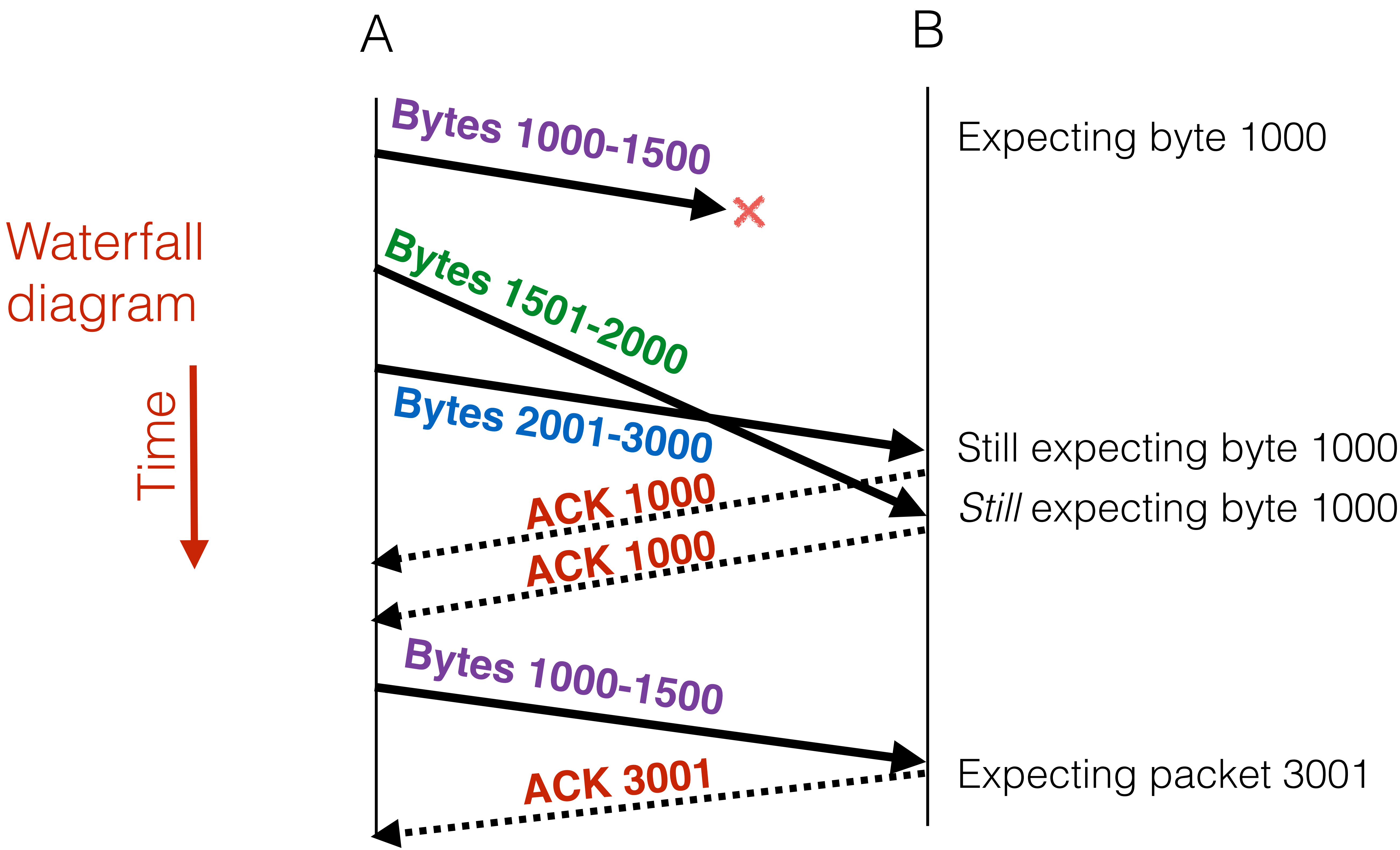
How does TCP achieve reliability?



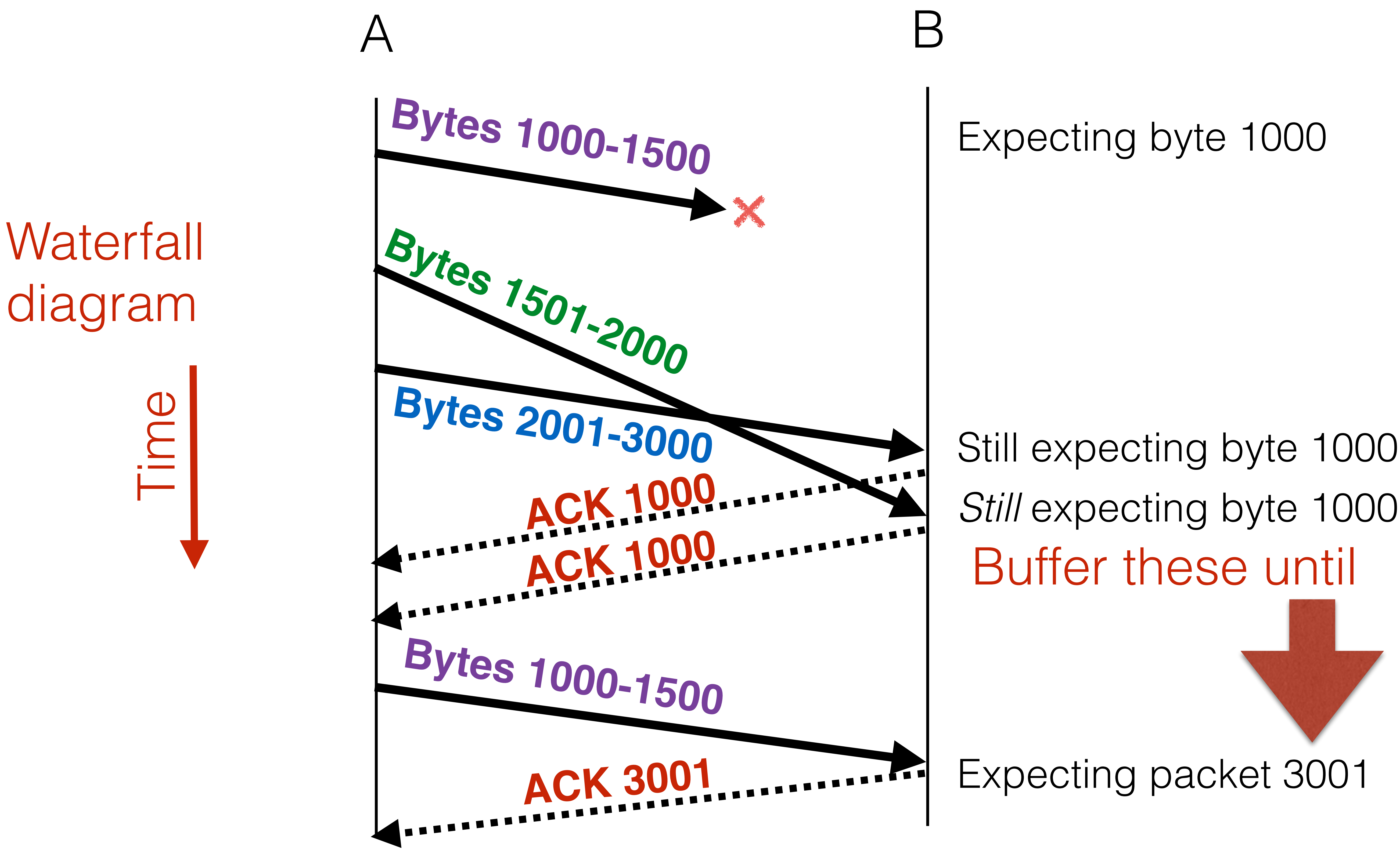
How does TCP achieve reliability?



How does TCP achieve reliability?



How does TCP achieve reliability?

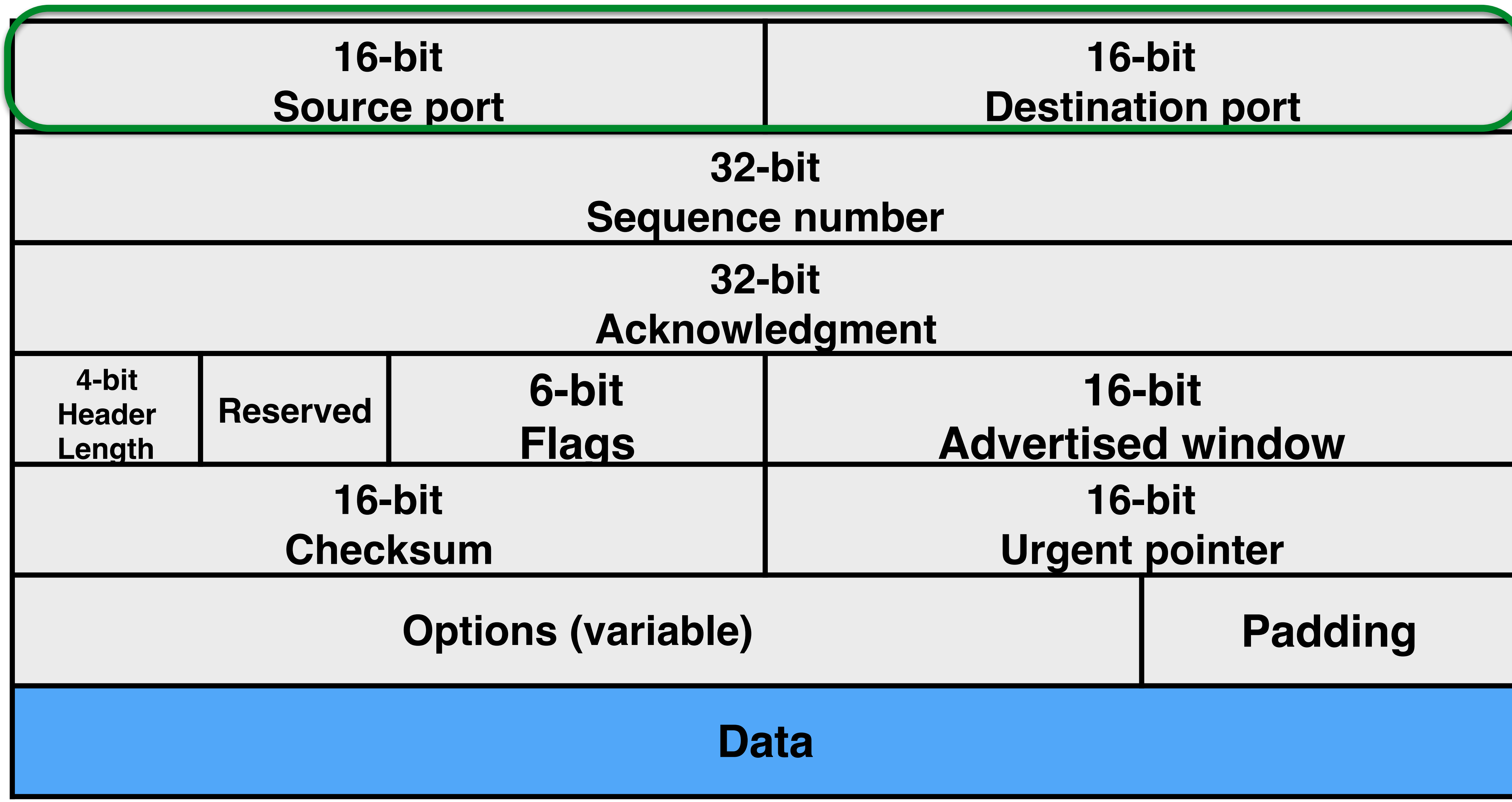


TCP congestion control

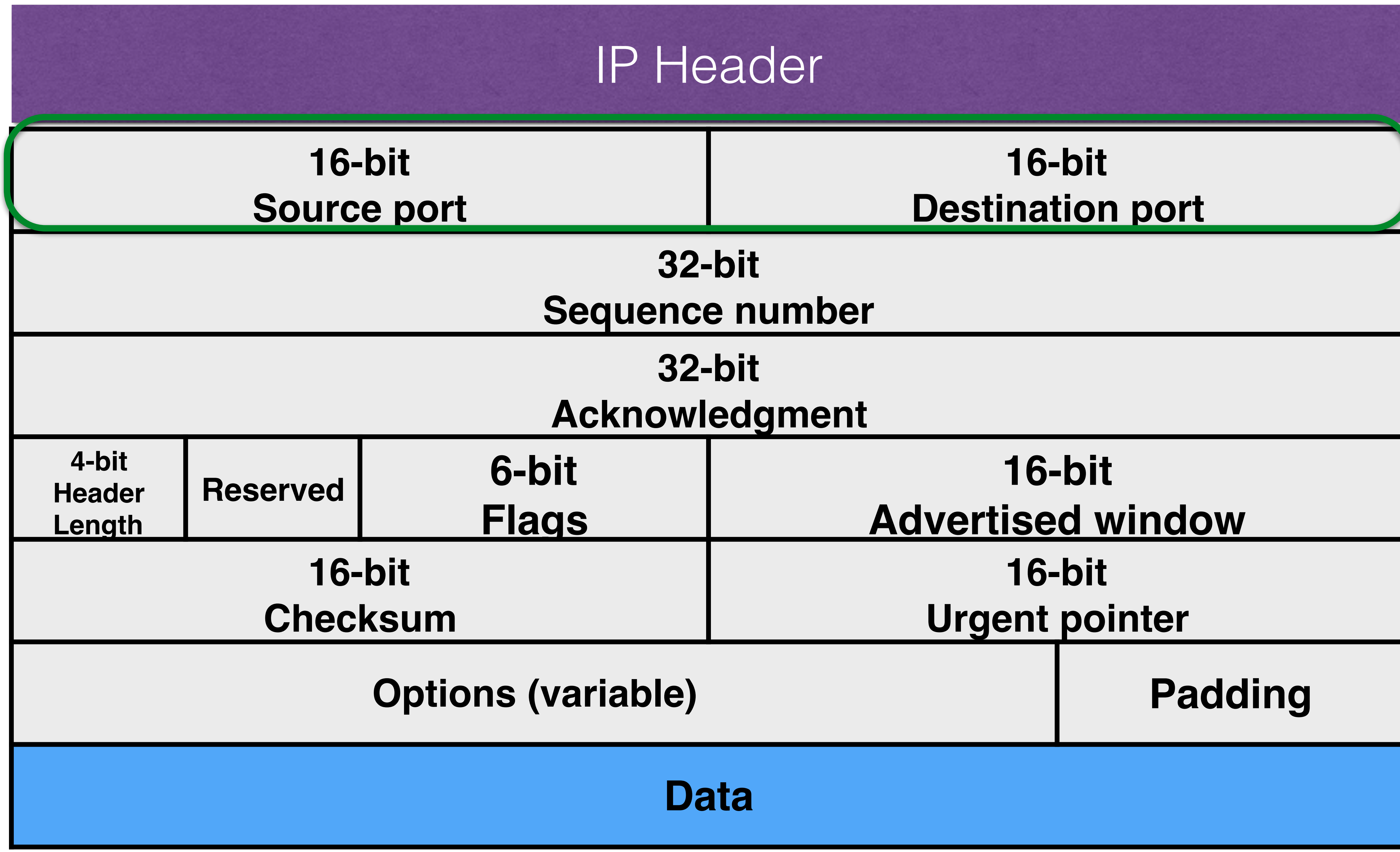
TCP's second job: don't break the network!

- 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


16-bit
Source port
16-bit
Destination port
32-bit
Sequence number
32-bit
Acknowledgment
4-bit
Header
Length
Reserved
6-bit
Flags
16-bit
Advertised window
16-bit
Checksum
16-bit
Urgent pointer
Options (variable)
Padding
Data

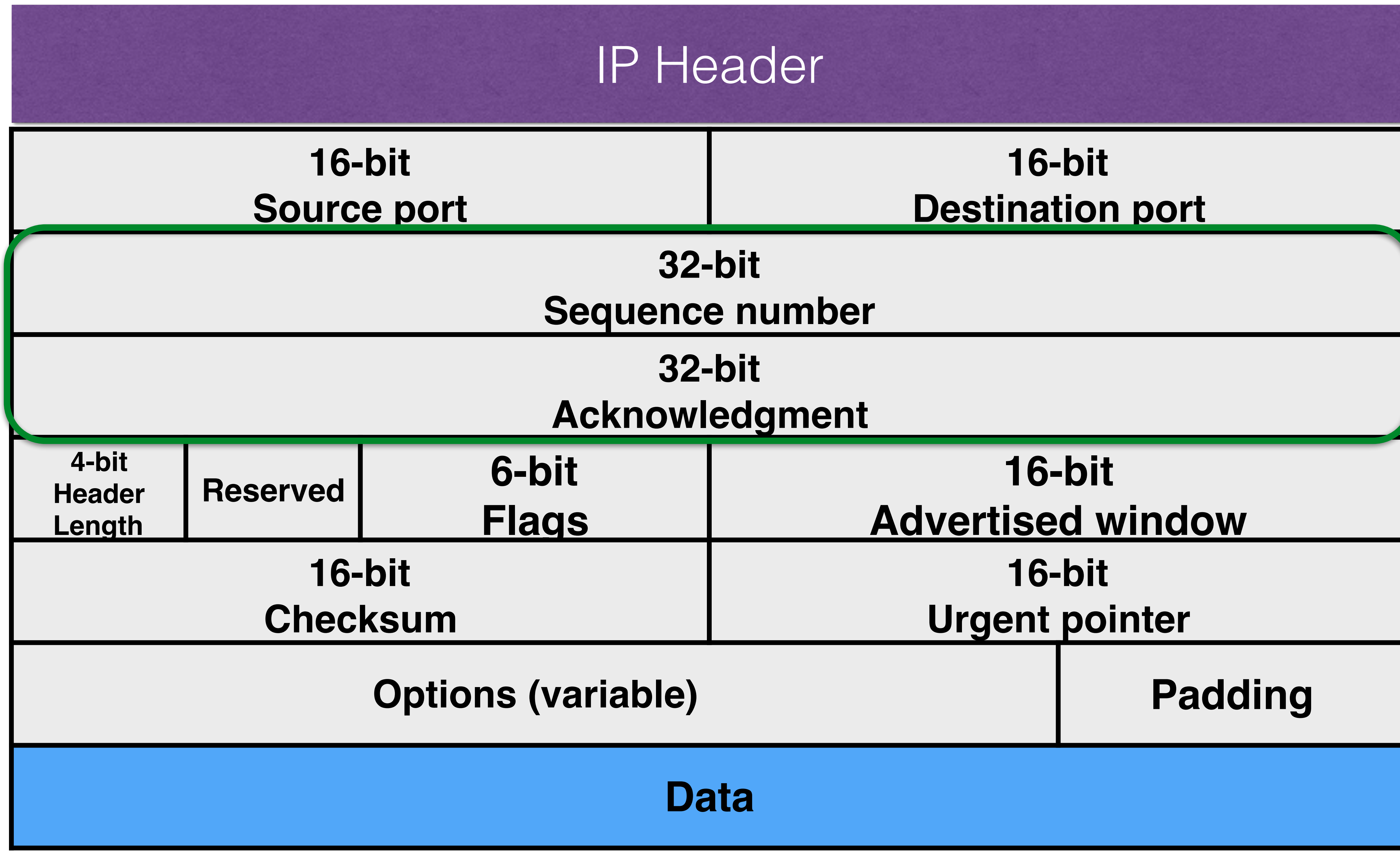
TCP header


IP Header
16-bit
Source port
16-bit
Destination port
32-bit
Sequence number
32-bit
Acknowledgment
4-bit
Header
Length
Reserved
6-bit
Flags
16-bit
Advertised window
16-bit
Checksum
16-bit
Urgent pointer
Options (variable)
Padding
Data

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

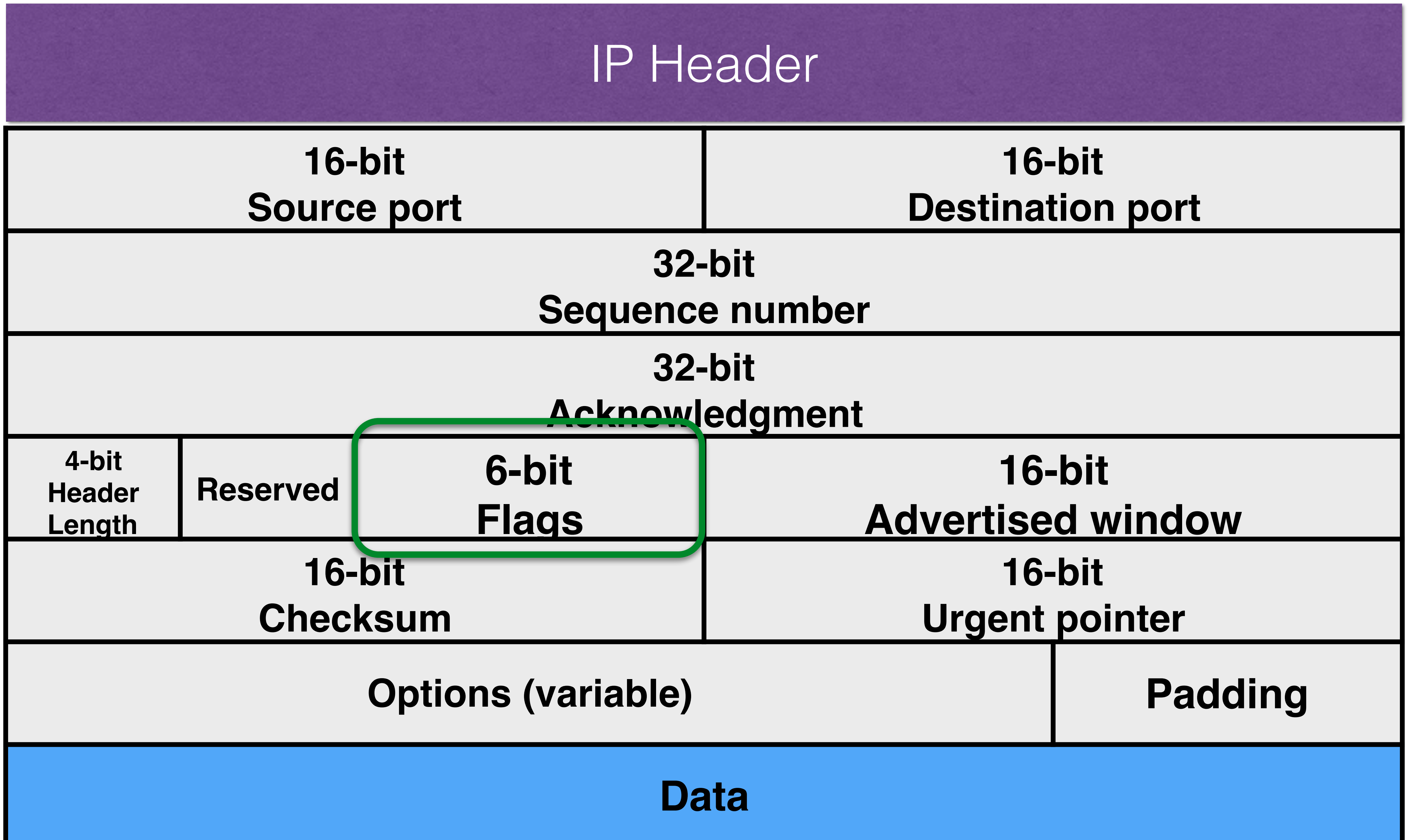
TCP header


IP Header
16-bit
Source port
16-bit
Destination port
32-bit
Sequence number
32-bit
Acknowledgment
4-bit
Header
Length
Reserved
6-bit
Flags
16-bit
Advertised window
16-bit
Checksum
16-bit
Urgent pointer
Options (variable)
Padding
Data

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


IP Header
16-bit
Source port
16-bit
Destination port
32-bit
Sequence number
32-bit
Acknowledgment
4-bit
Header
Length
Reserved
6-bit
Flags
16-bit
Advertised window
16-bit
Checksum
16-bit
Urgent pointer
Options (variable)
Padding
Data

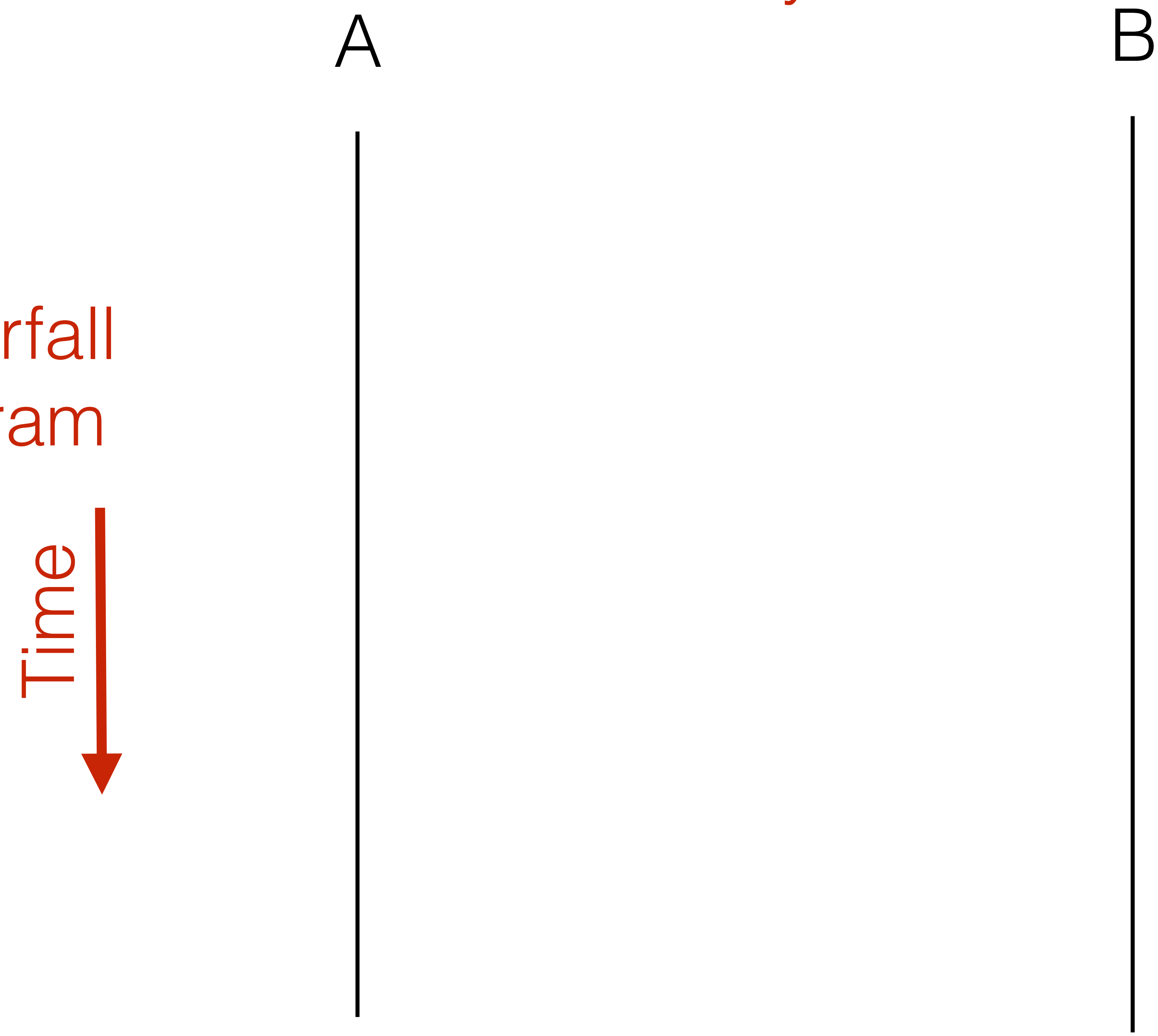
TCP flags

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

Setting up a connection

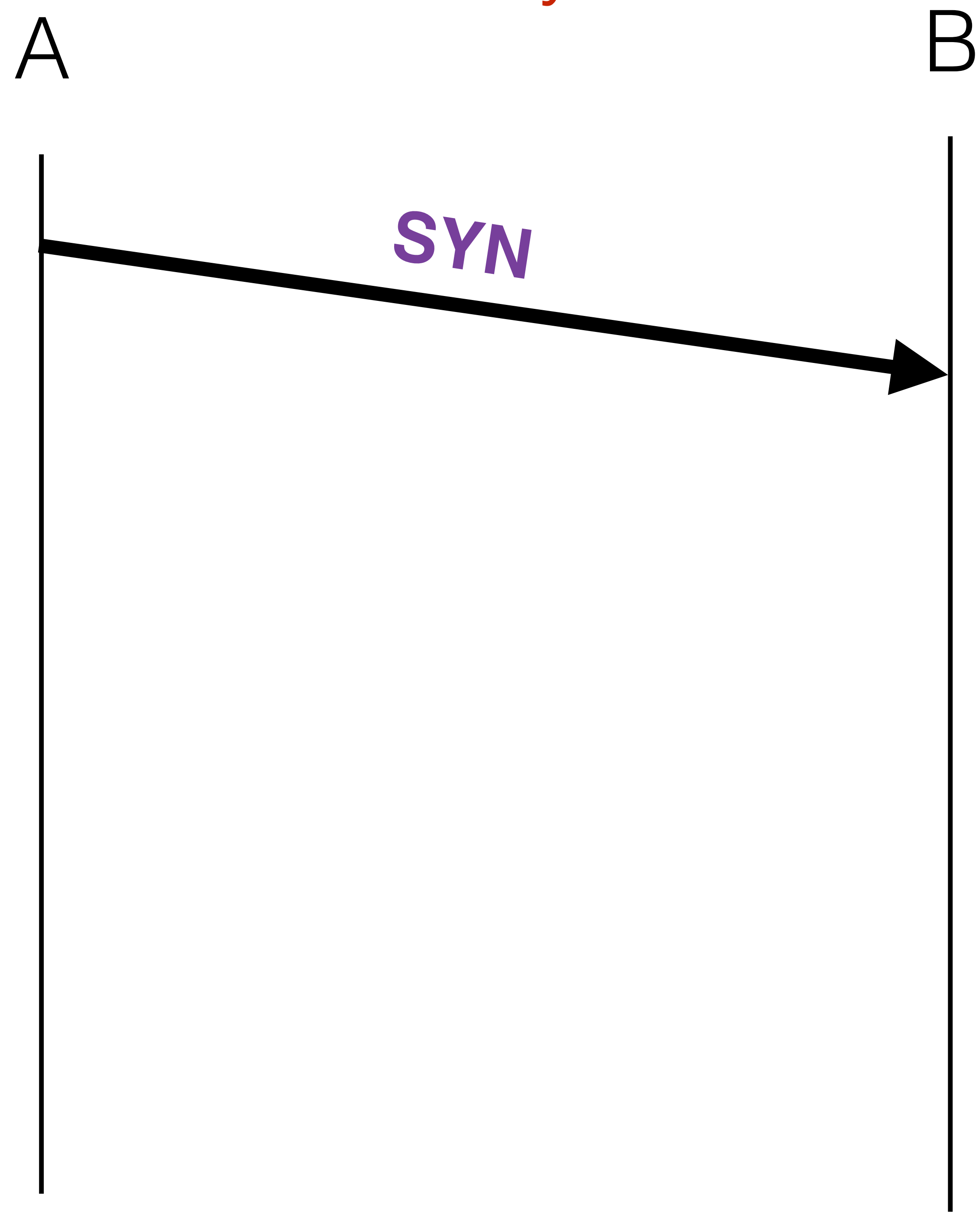
Three-way handshake

Waterfall diagram



Setting up a connection

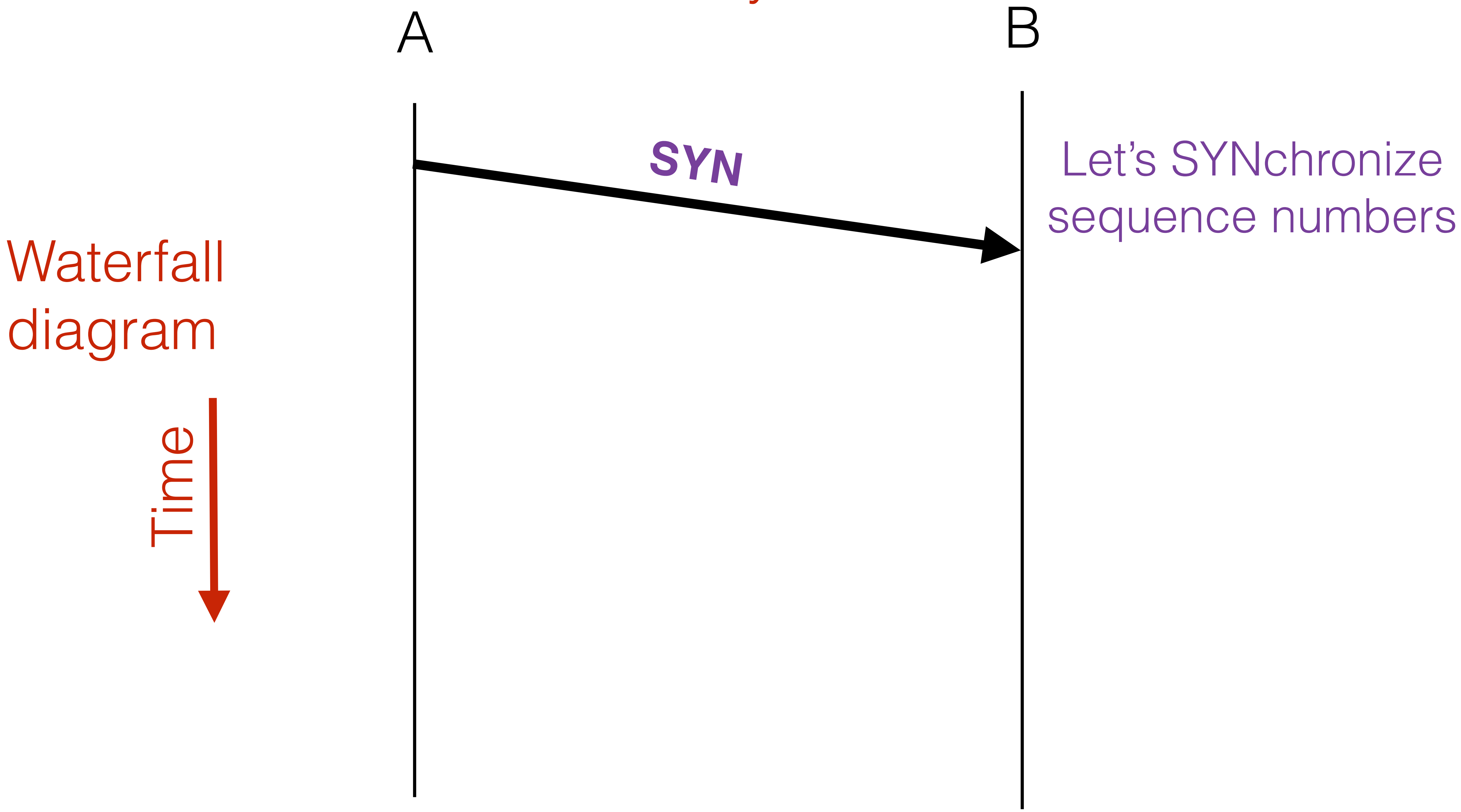
Three-way handshake



Waterfall diagram

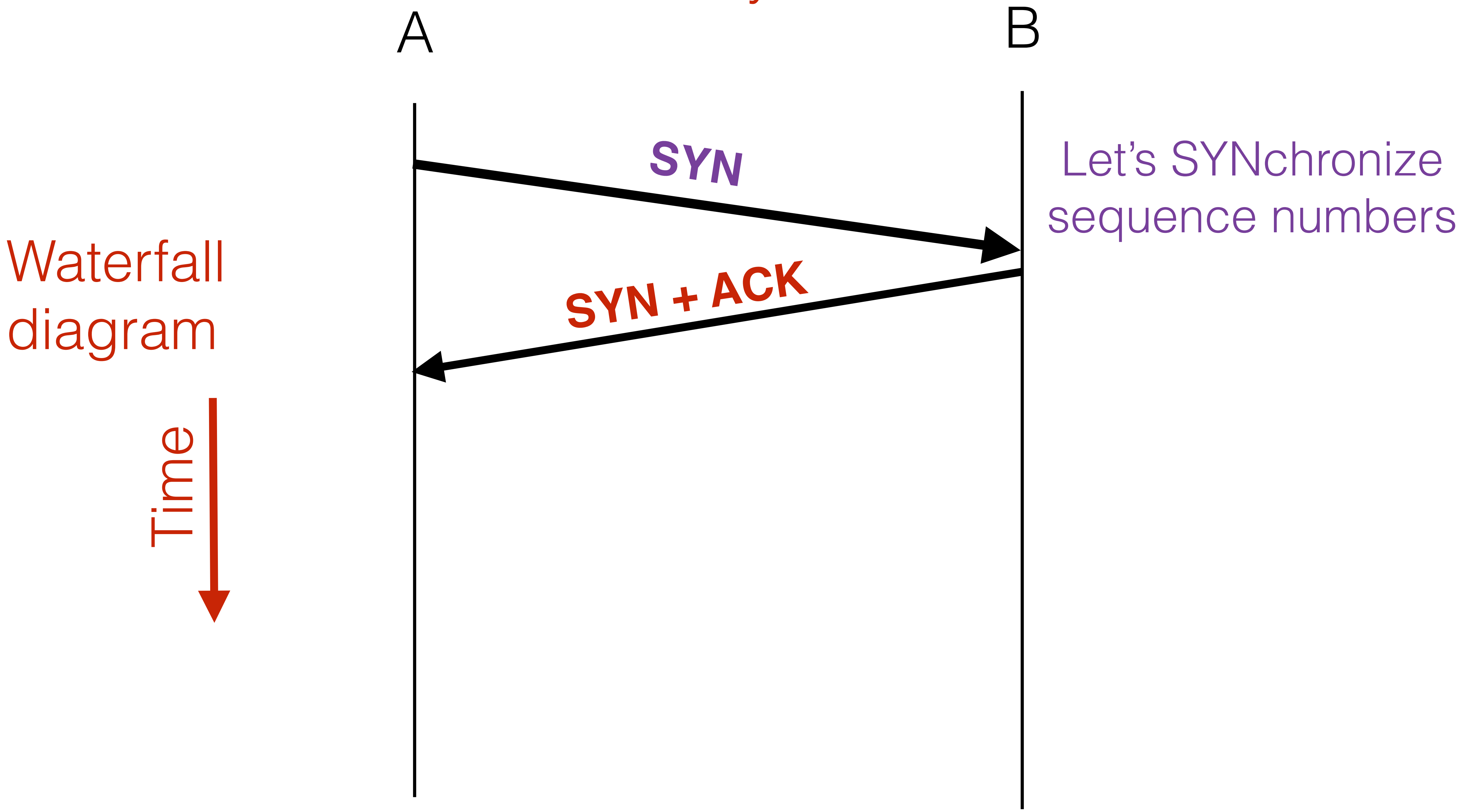
Setting up a connection

Three-way handshake

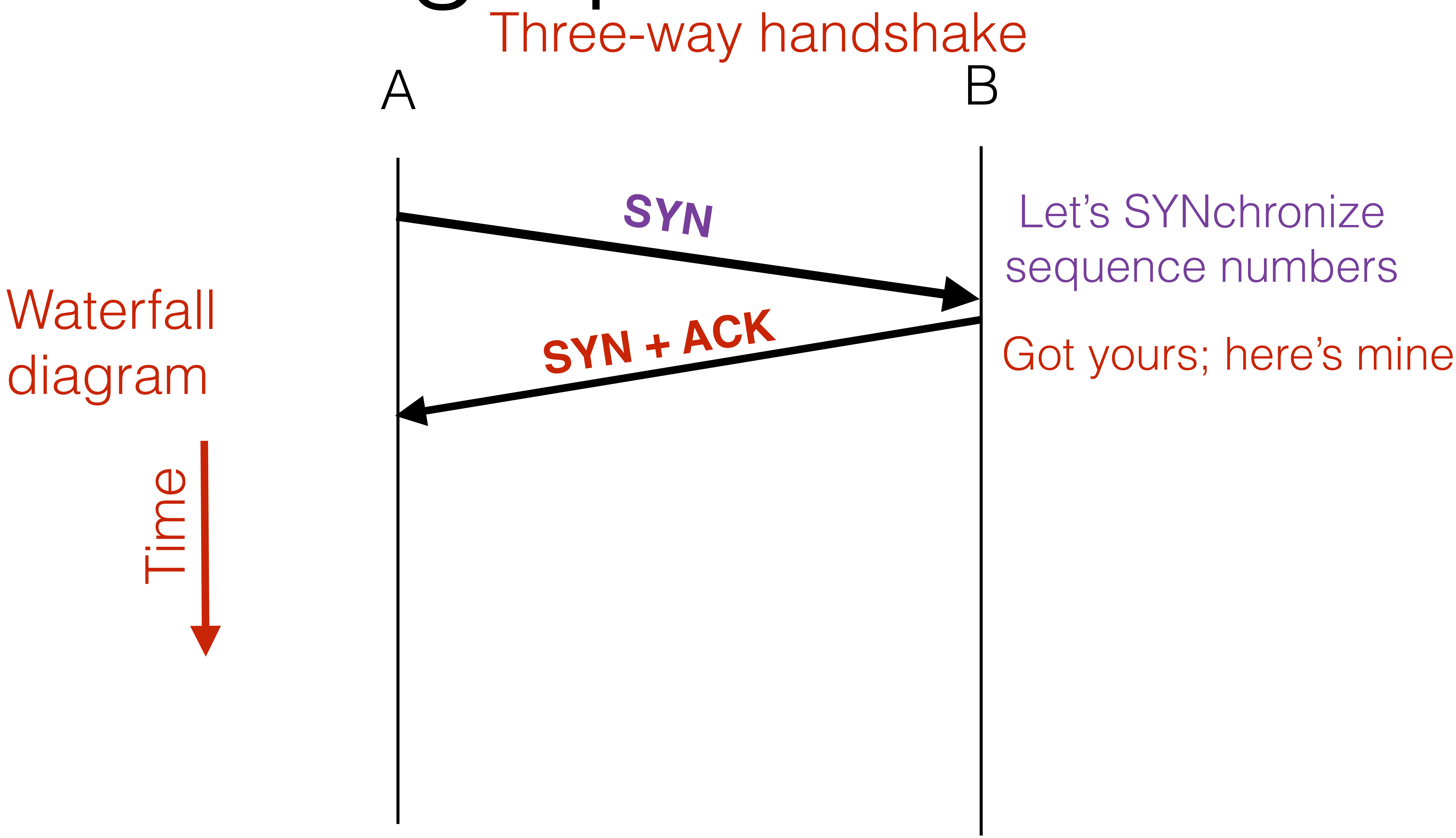


Setting up a connection

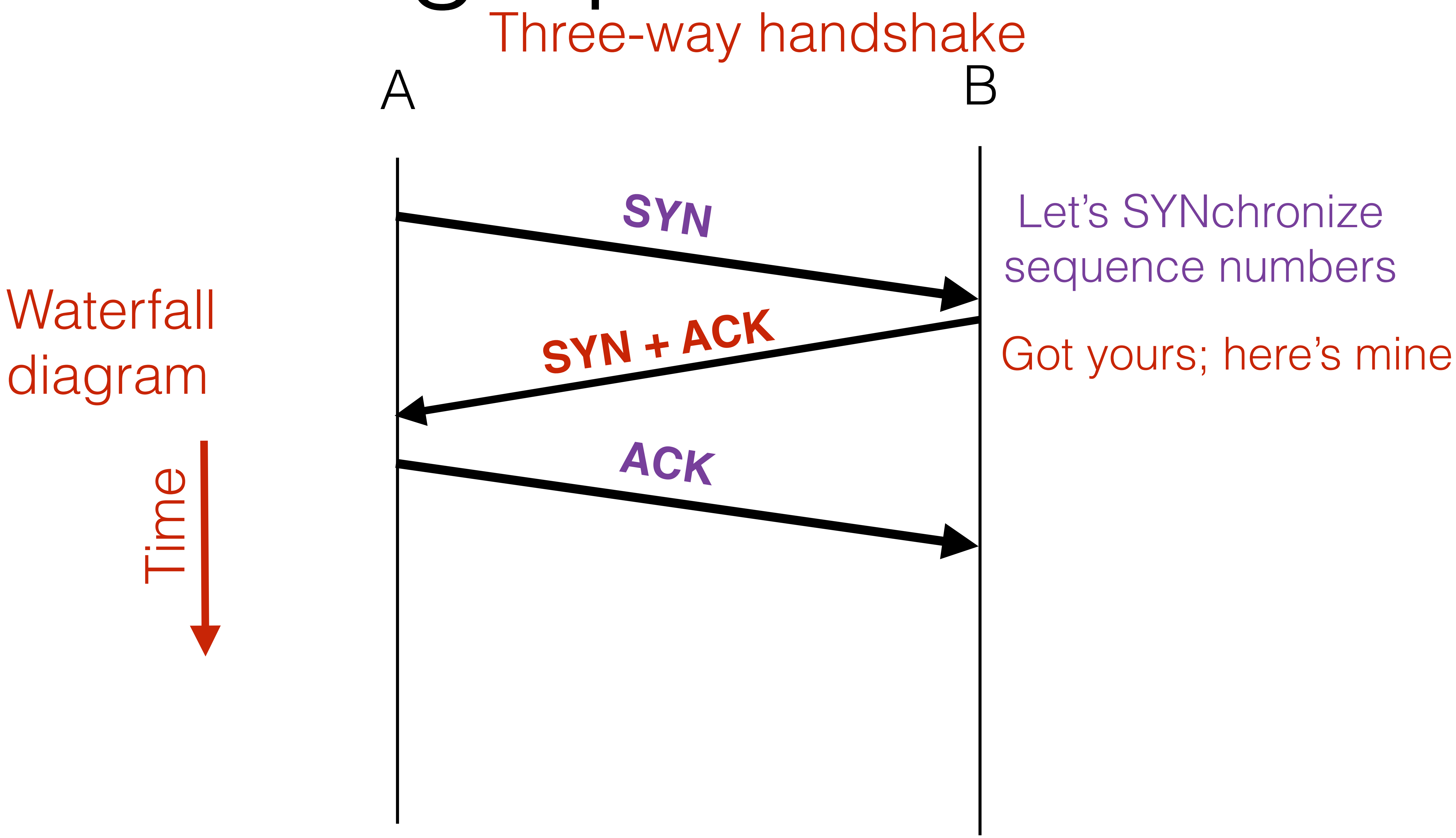
Three-way handshake



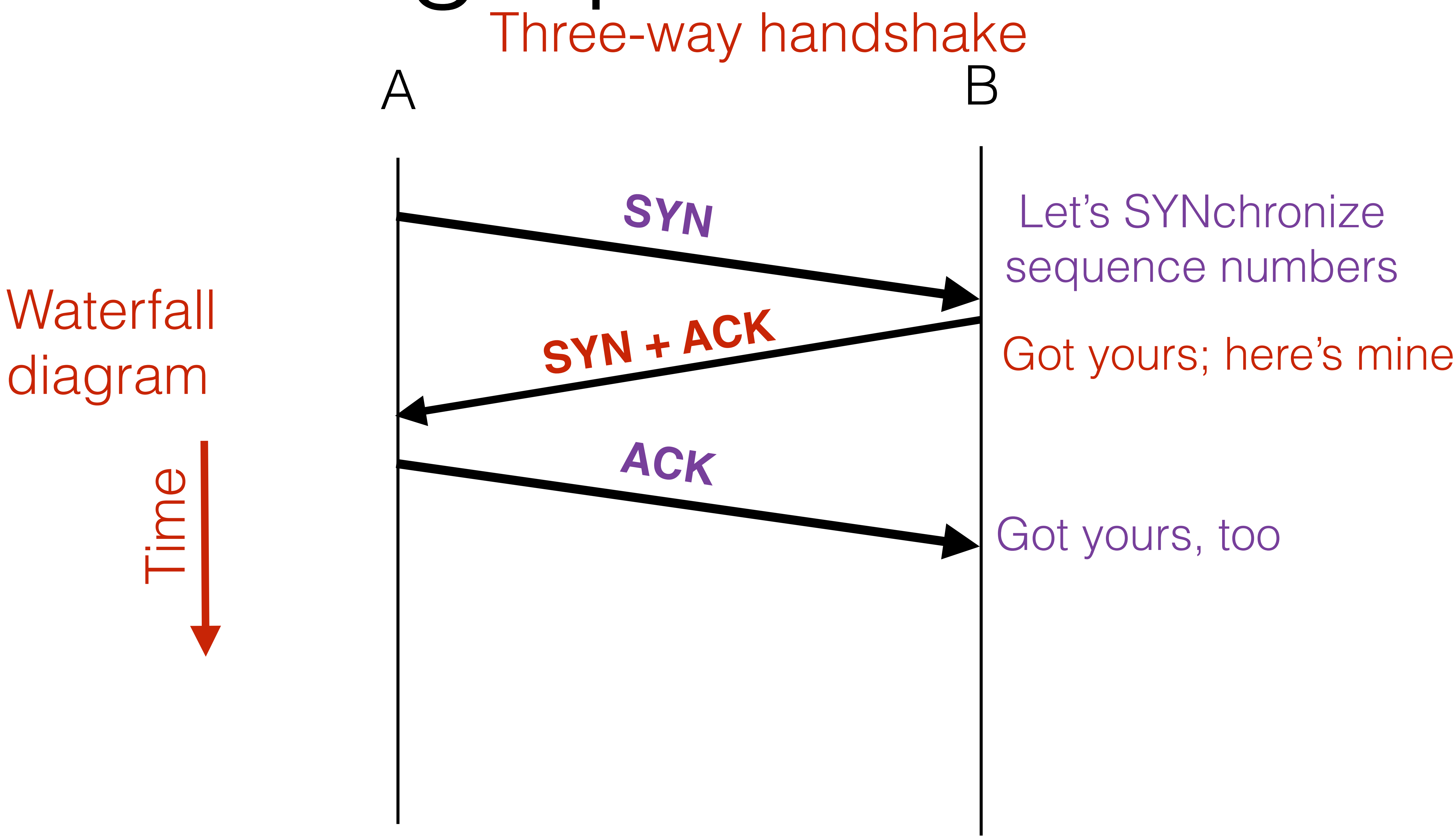
Setting up a connection



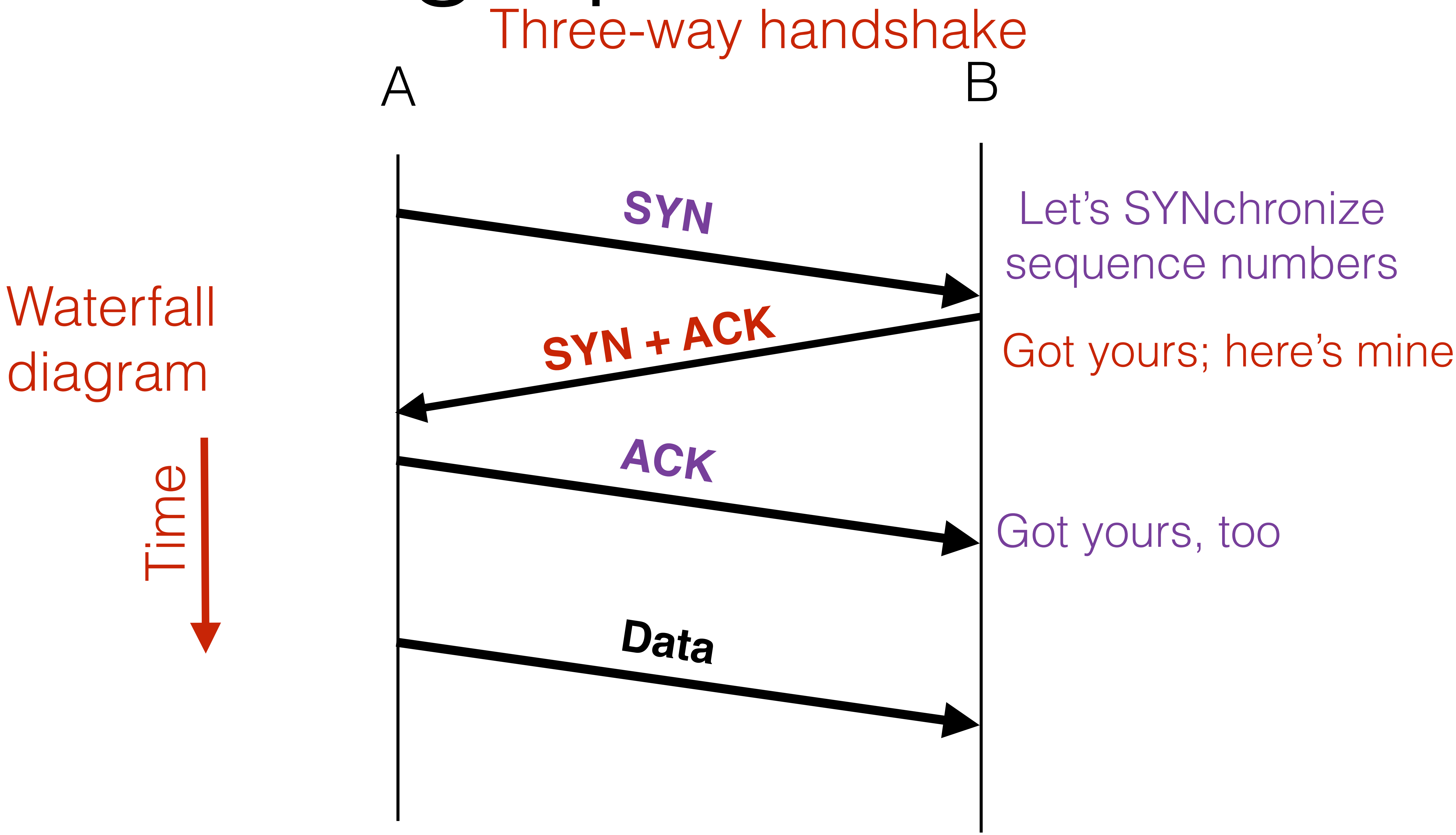
Setting up a connection



Setting up a connection

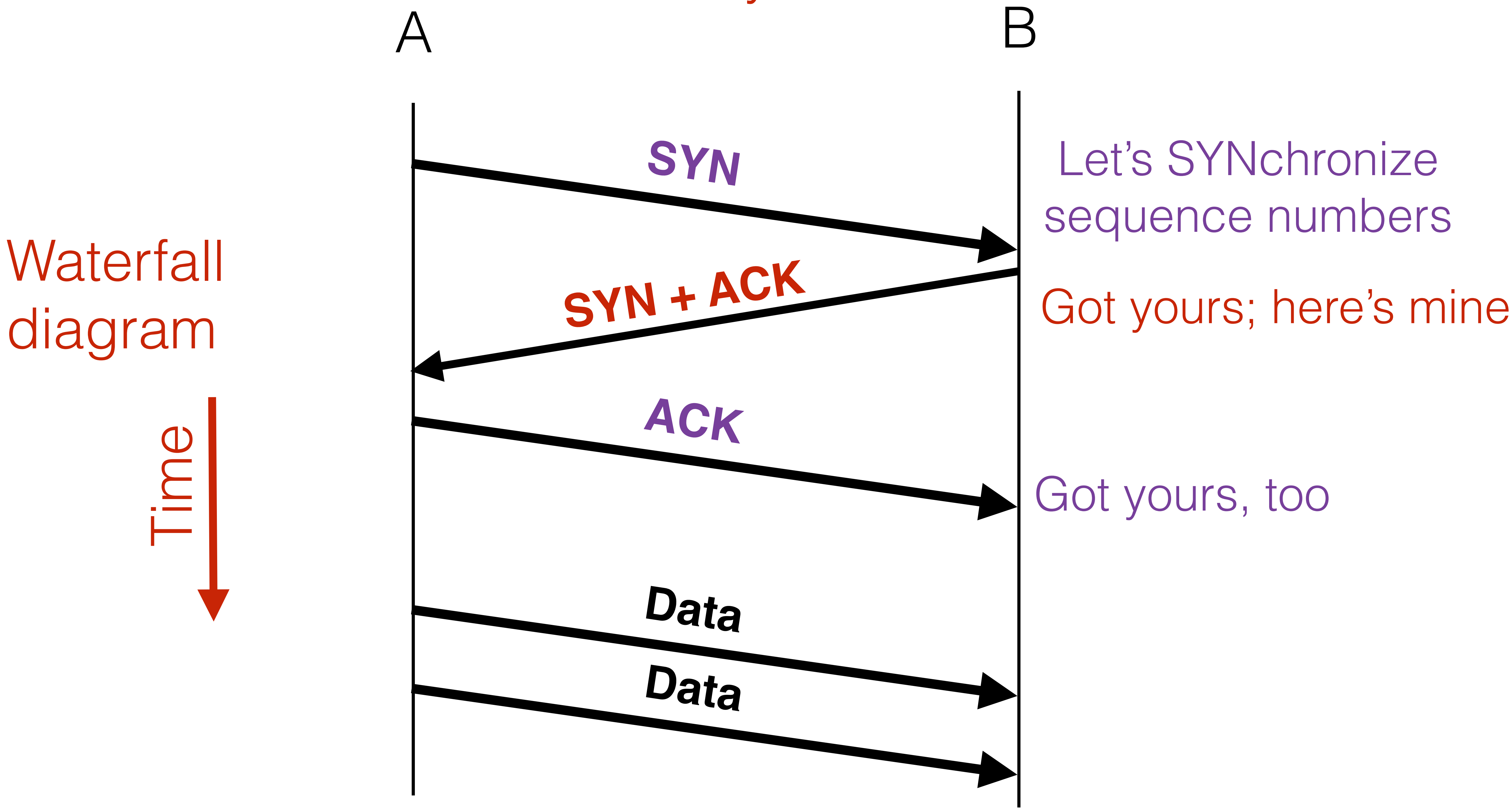


Setting up a connection

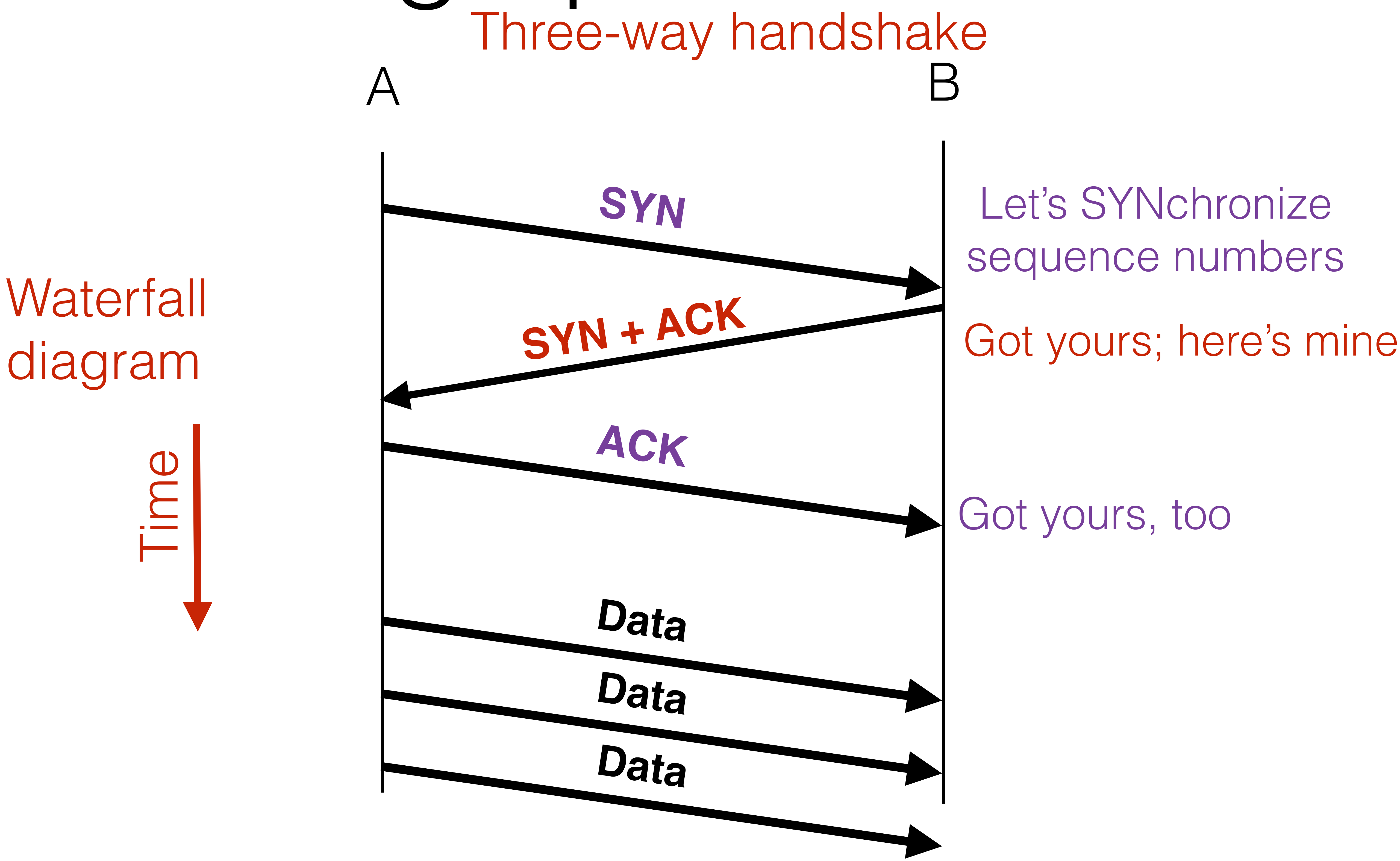


Setting up a connection

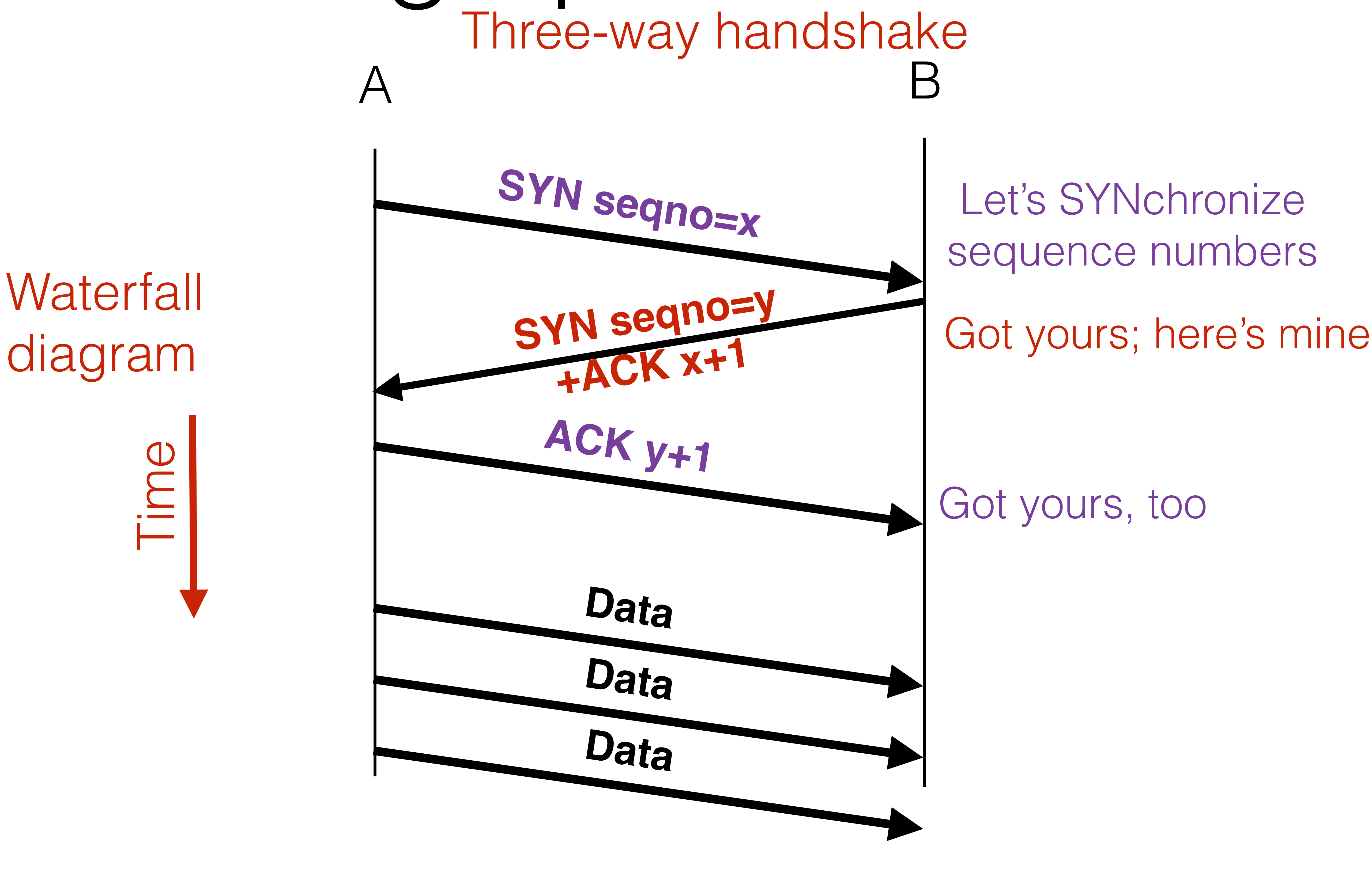
Three-way handshake



Setting up a connection



Setting up a connection



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

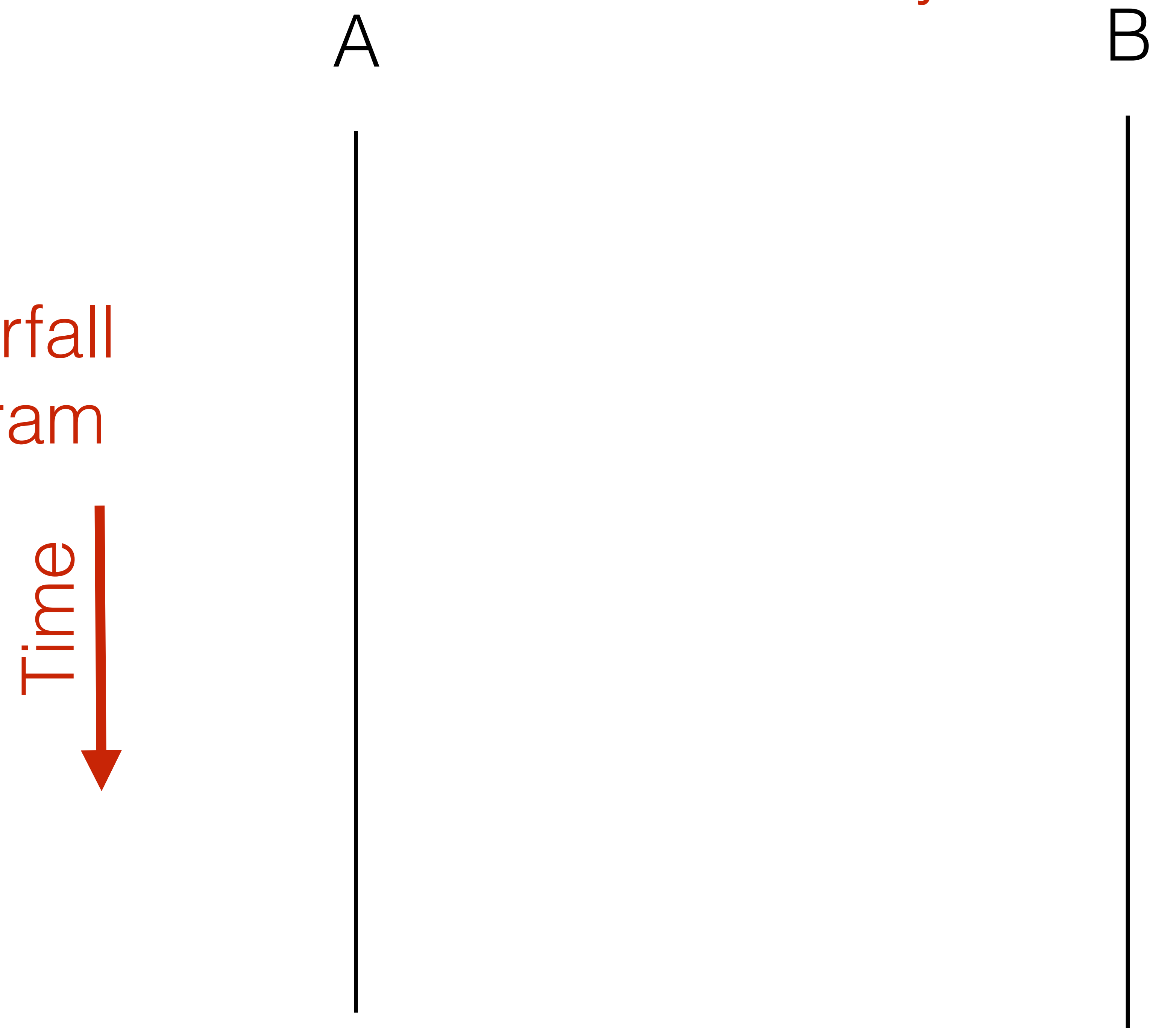
Attacks

- SYN flooding
- Injection attacks
- Opt-ack attack

SYN flooding

SYN flooding

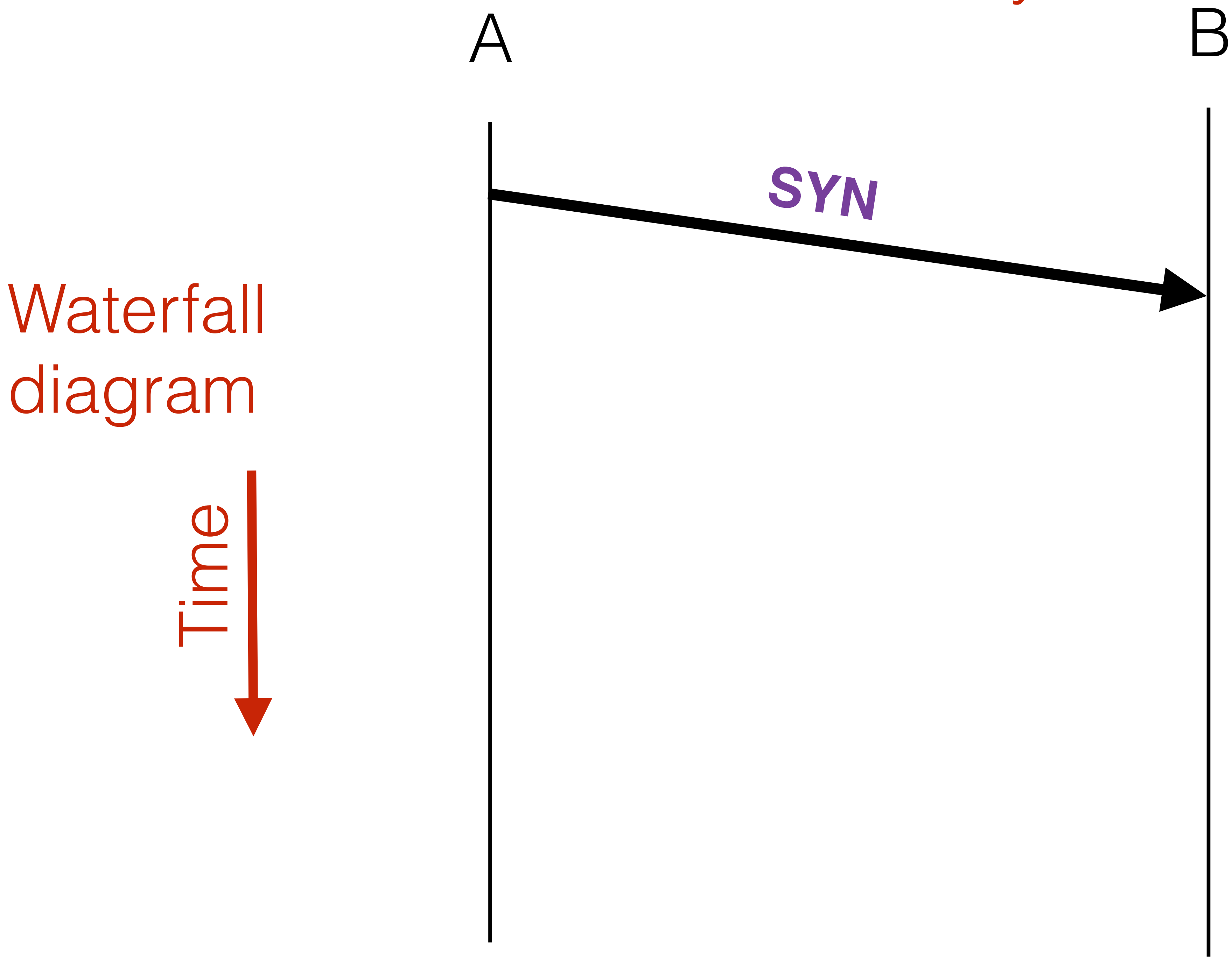
Recall the three-way handshake:



Waterfall diagram

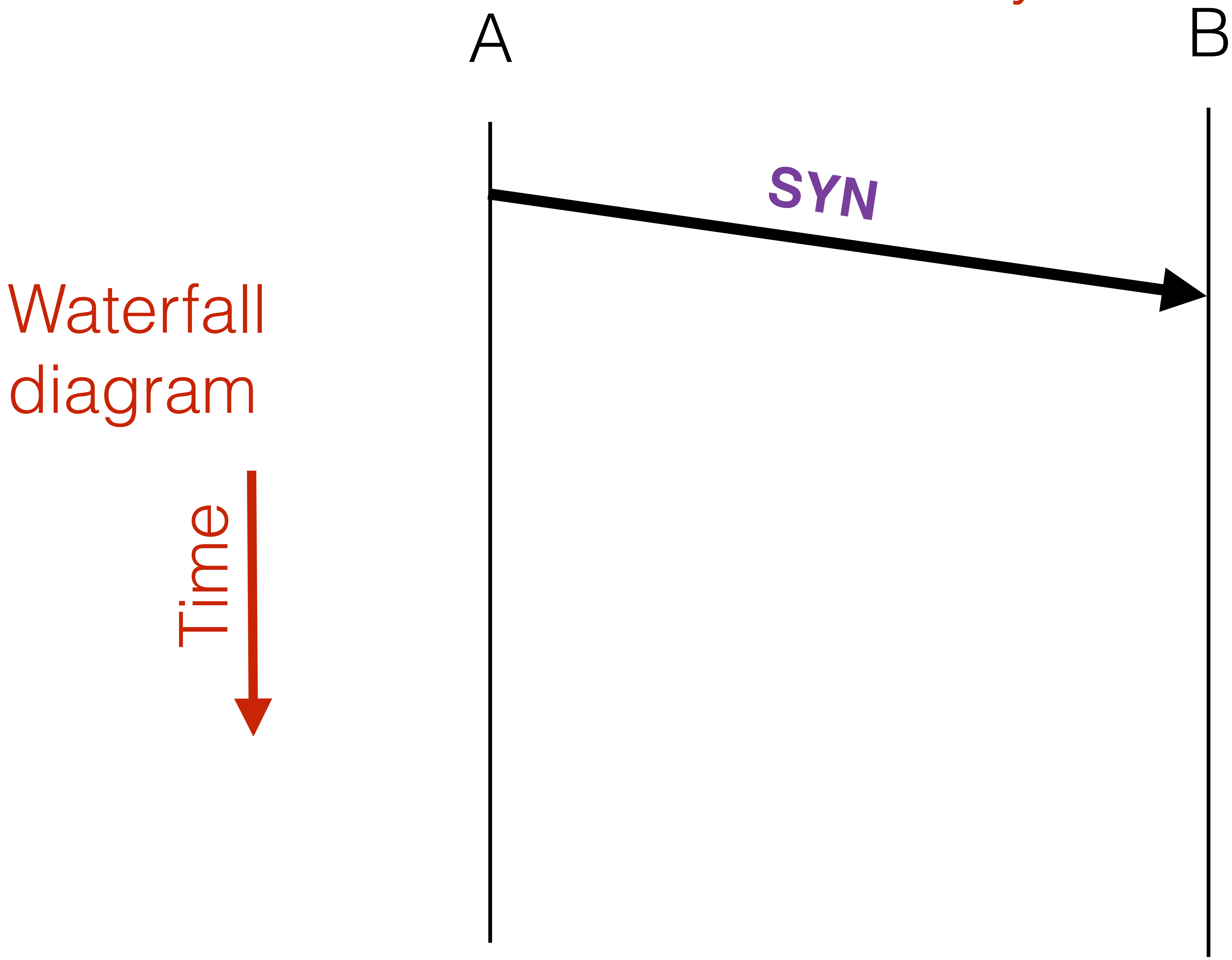
SYN flooding

Recall the three-way handshake:



SYN flooding

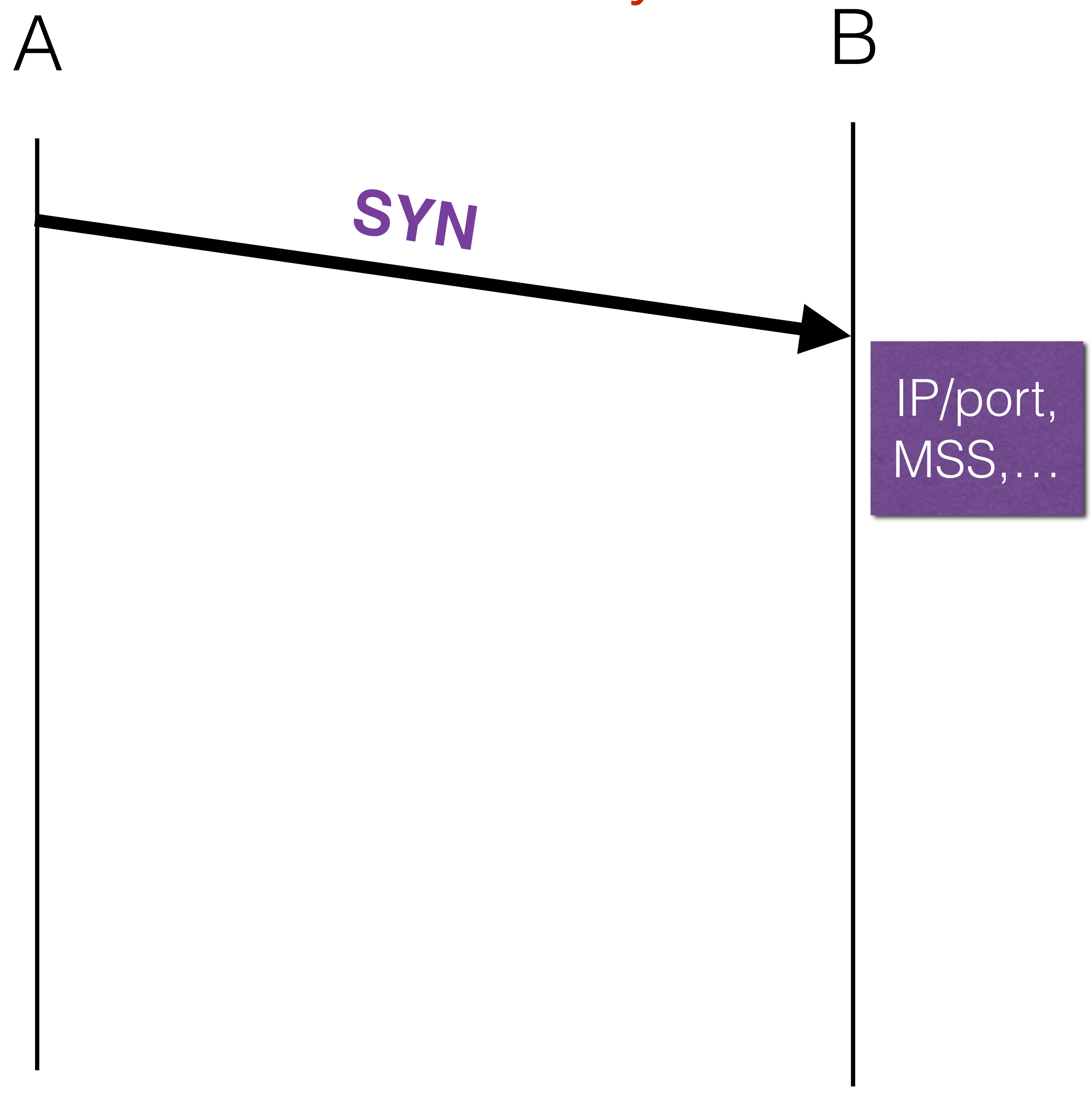
Recall the three-way handshake:



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

SYN flooding

Recall the three-way handshake:



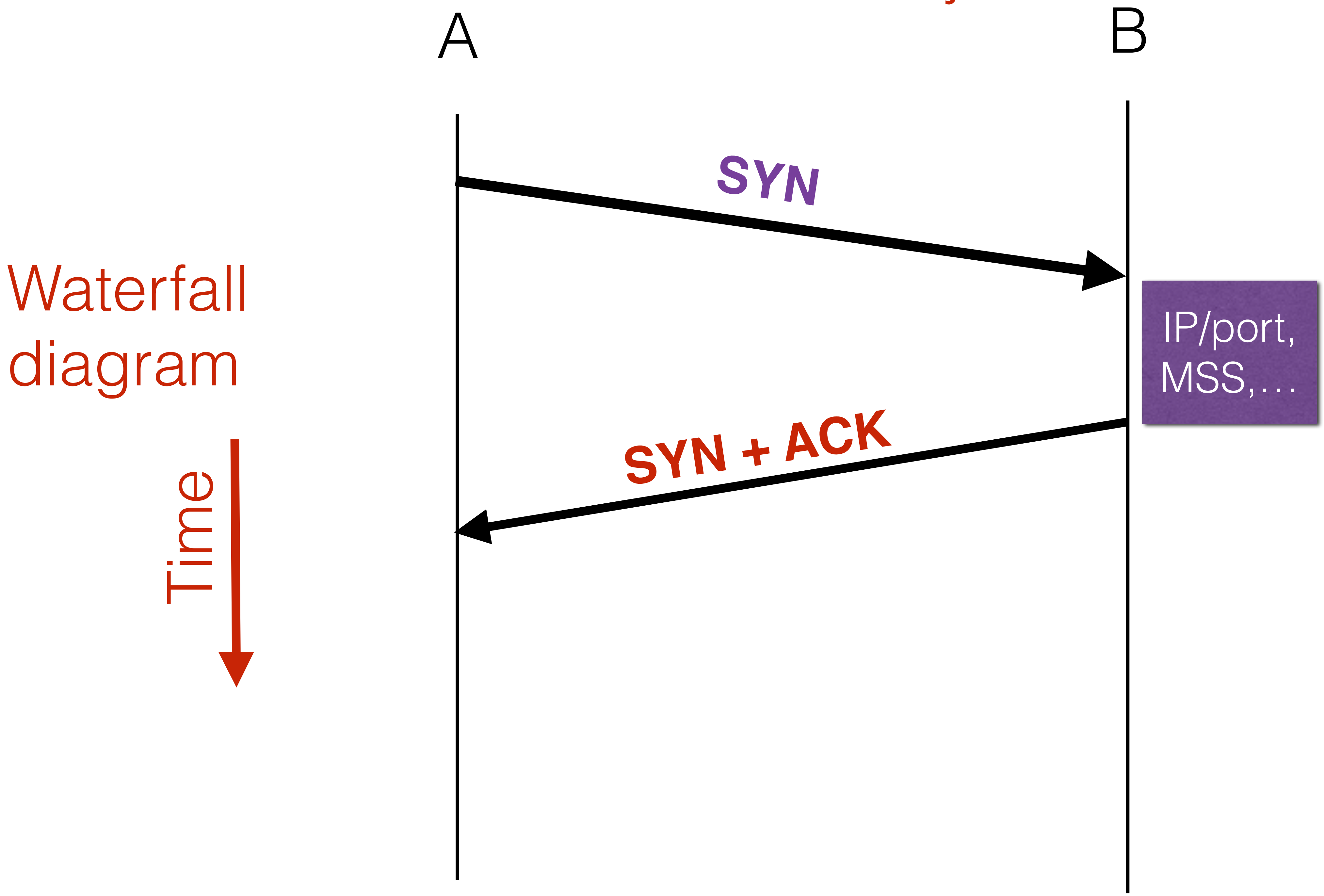
Waterfall diagram

Time

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

SYN flooding

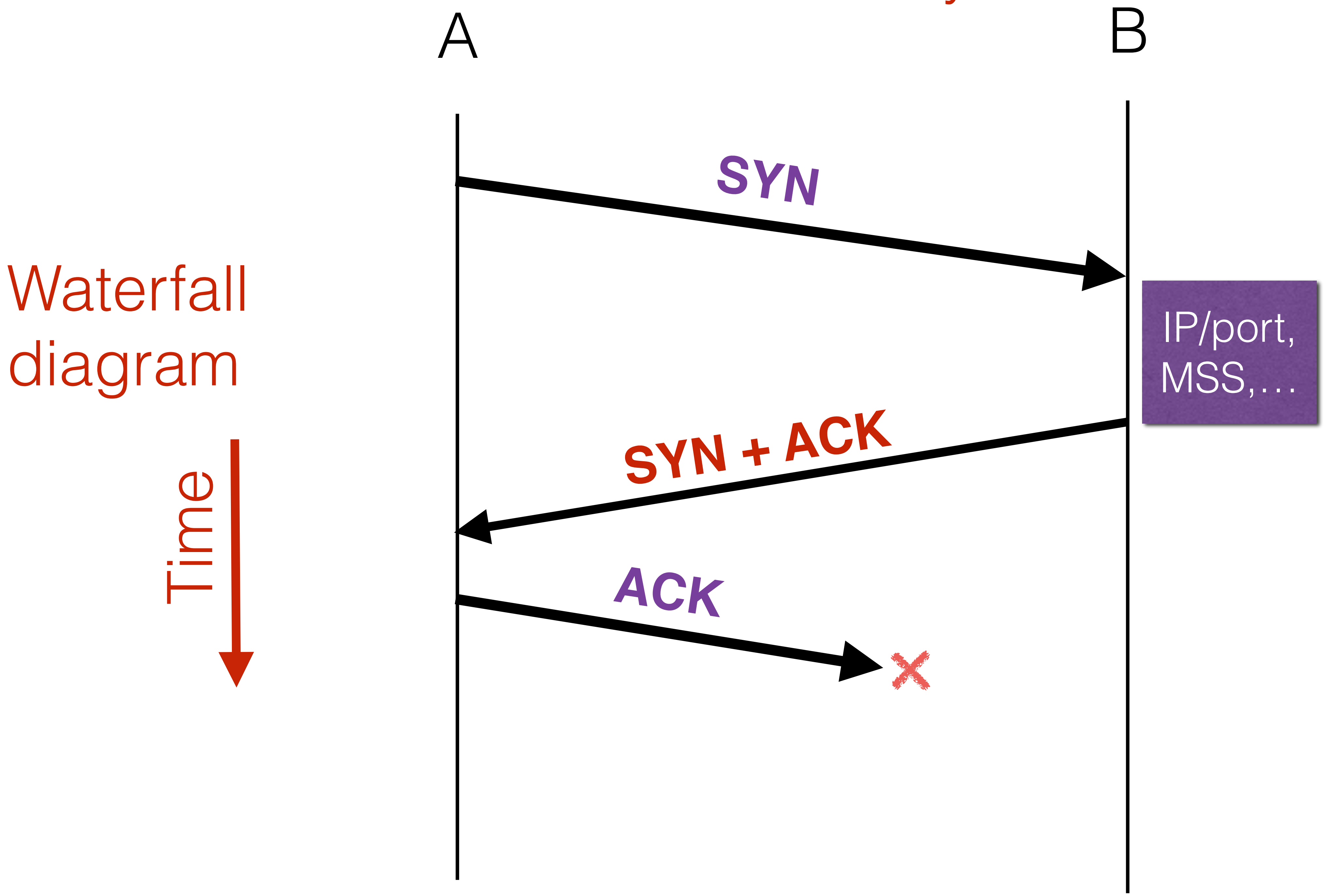
Recall the three-way handshake:



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

SYN flooding

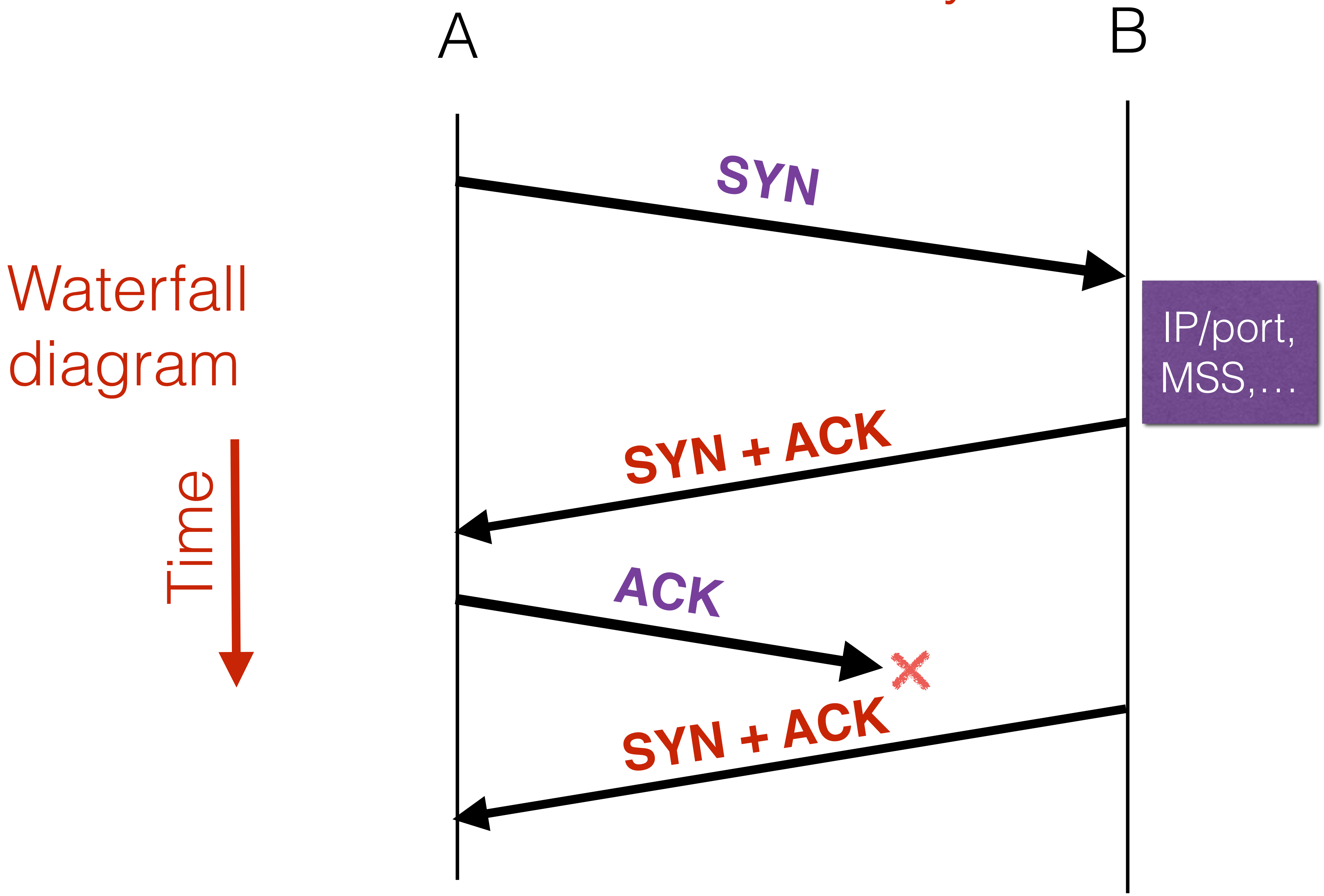
Recall the three-way handshake:



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

SYN flooding

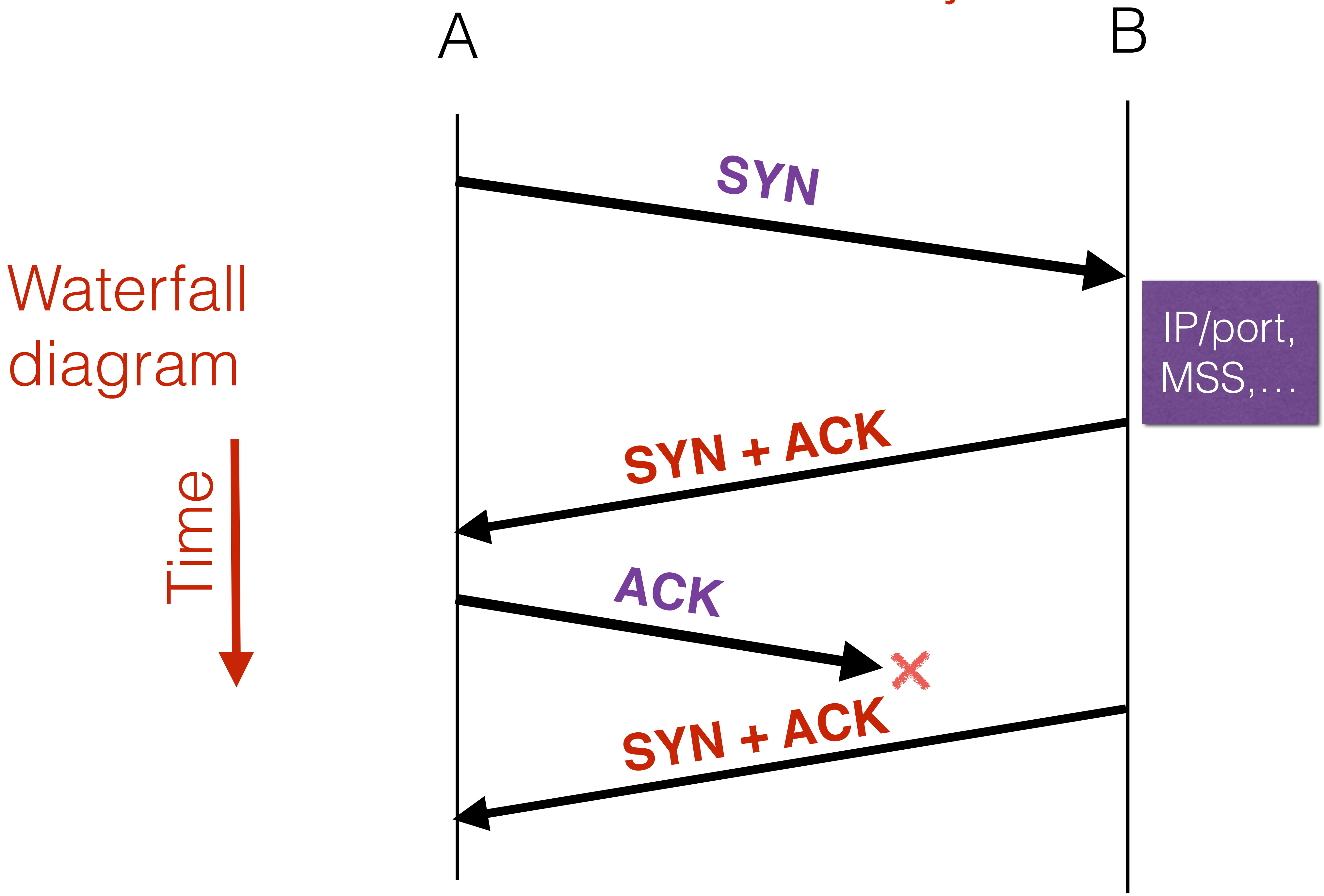
Recall the three-way handshake:



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

SYN flooding

Recall the three-way handshake:

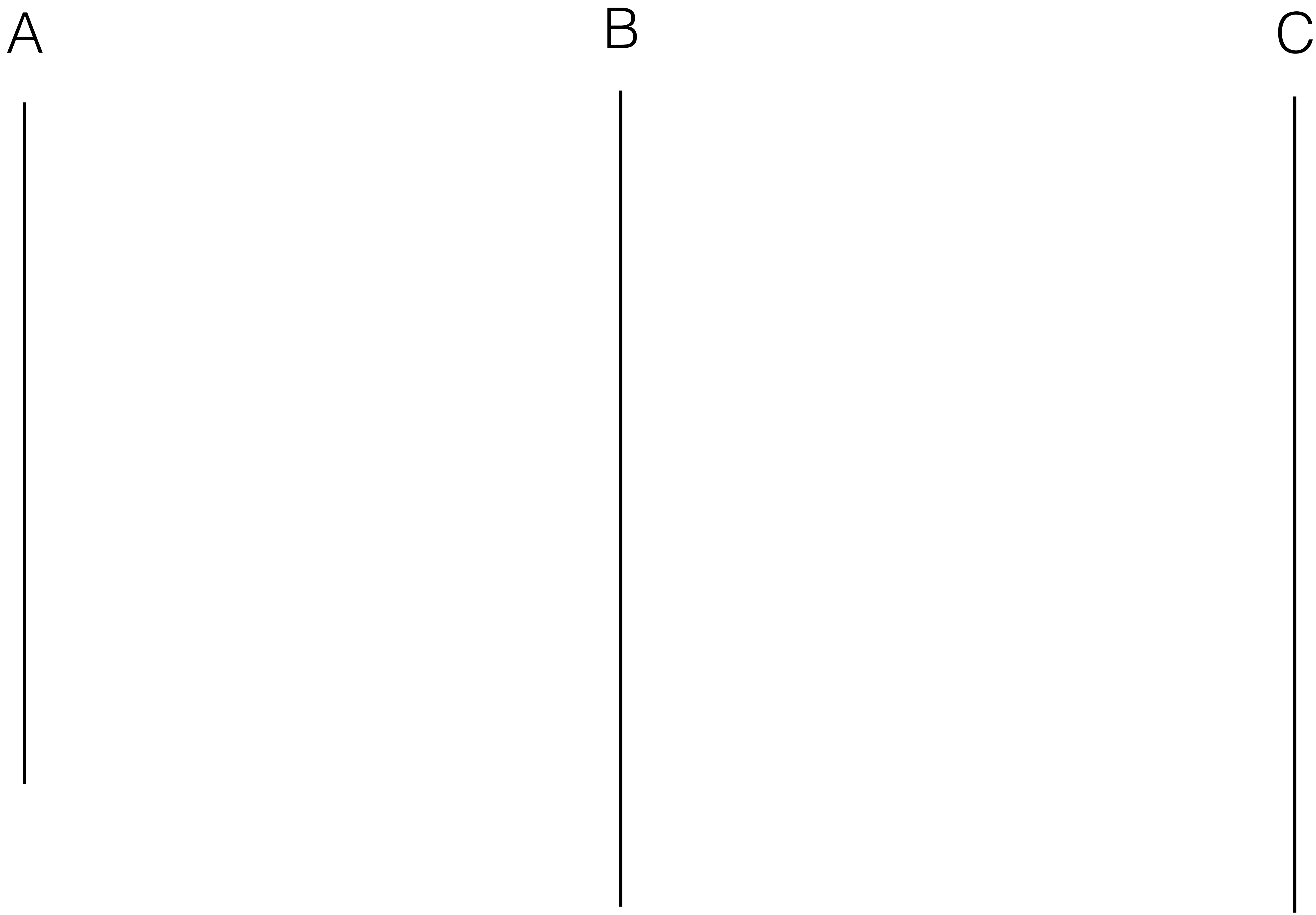


At this point, B allocates state for this new connection (incl. IP, port, maximum segment size)

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

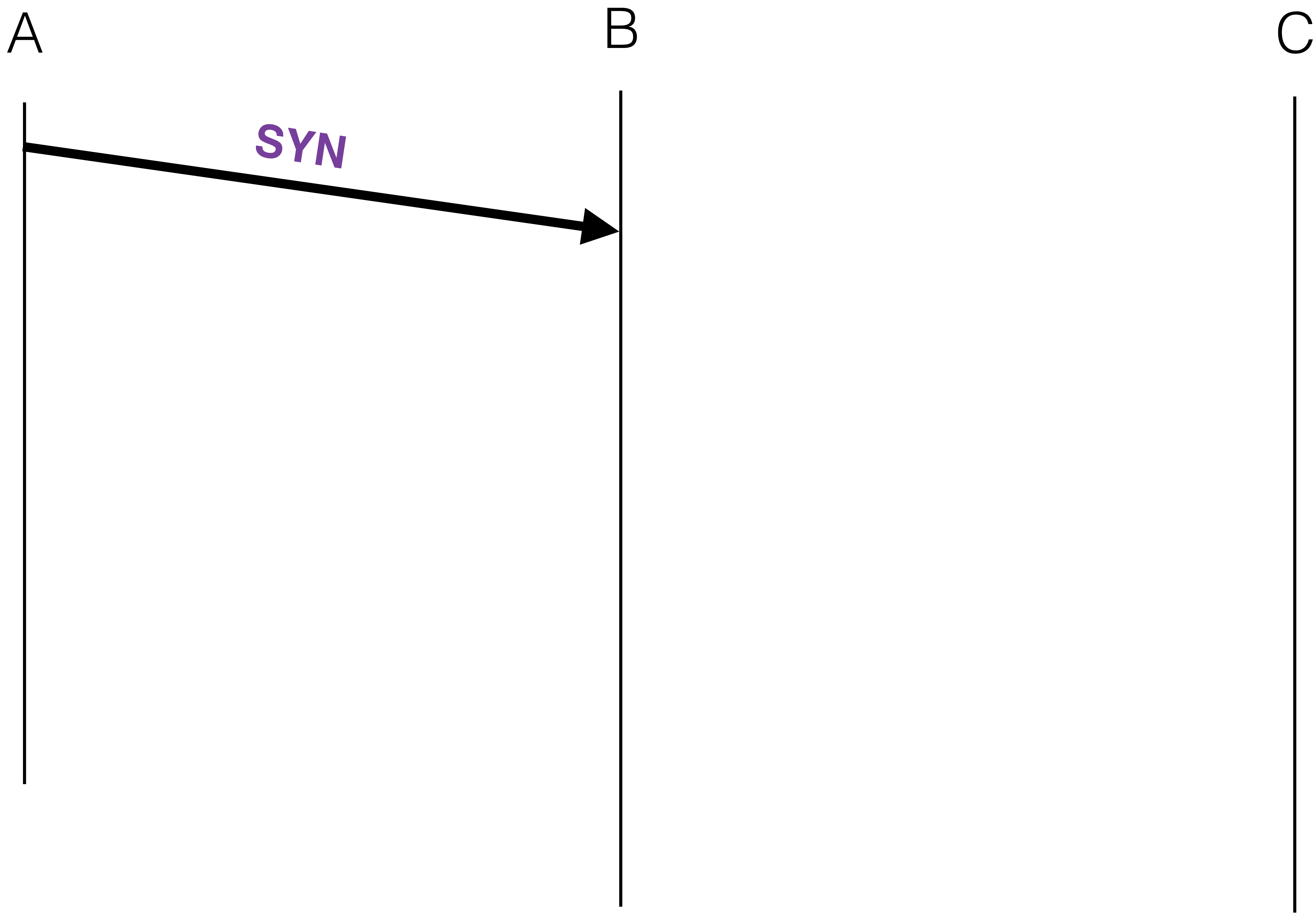
SYN flooding

The attack



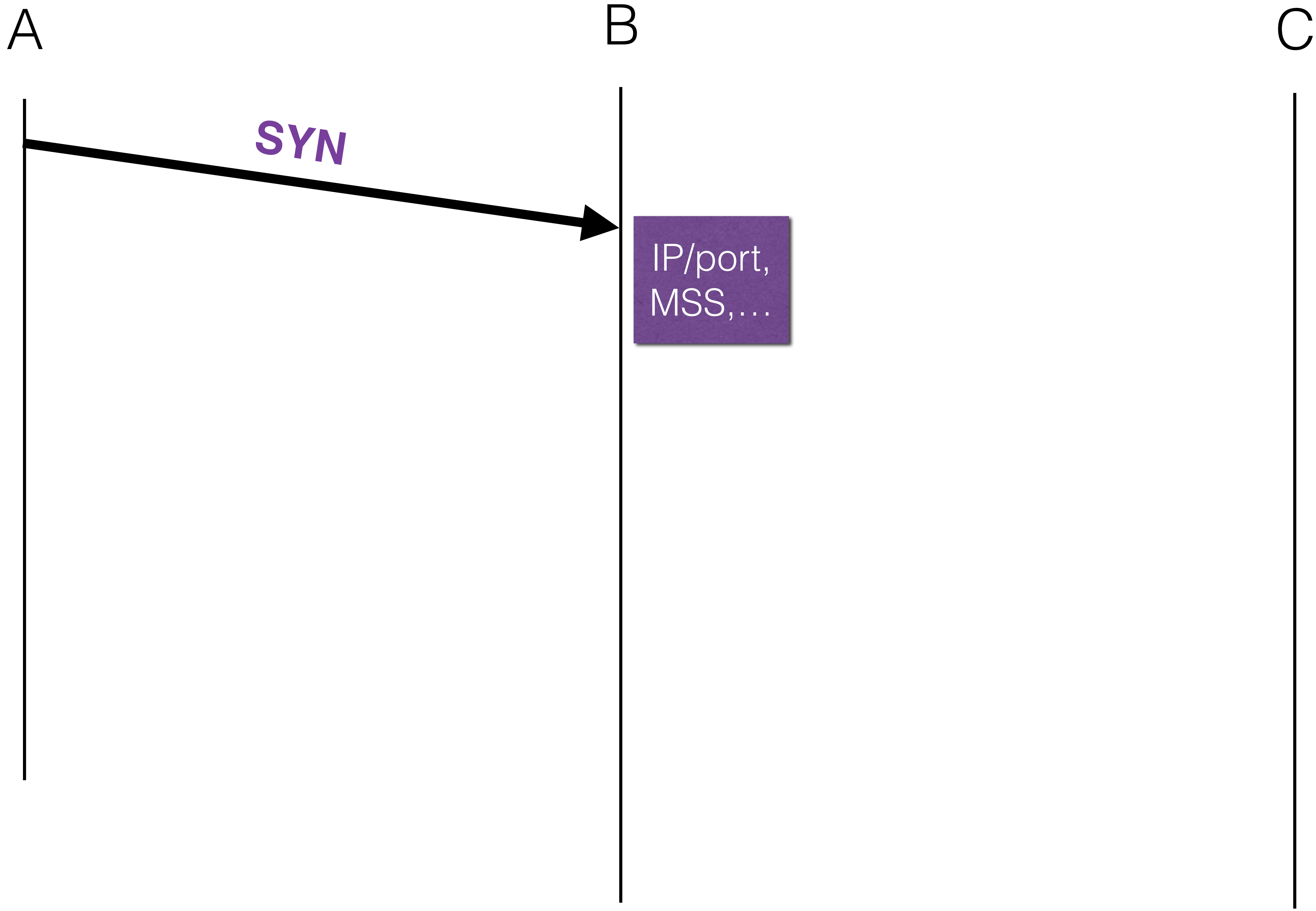
SYN flooding

The attack



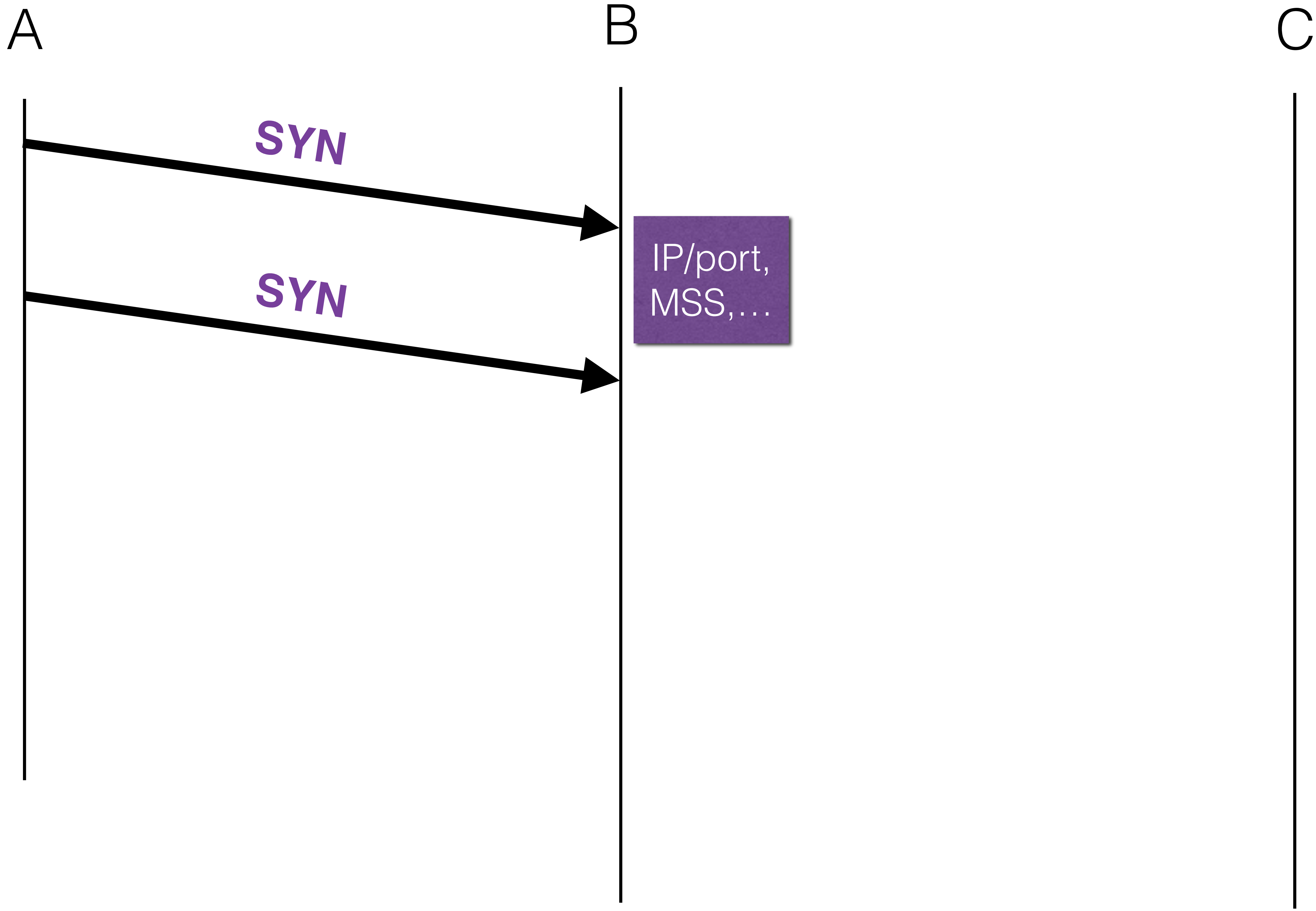
SYN flooding

The attack



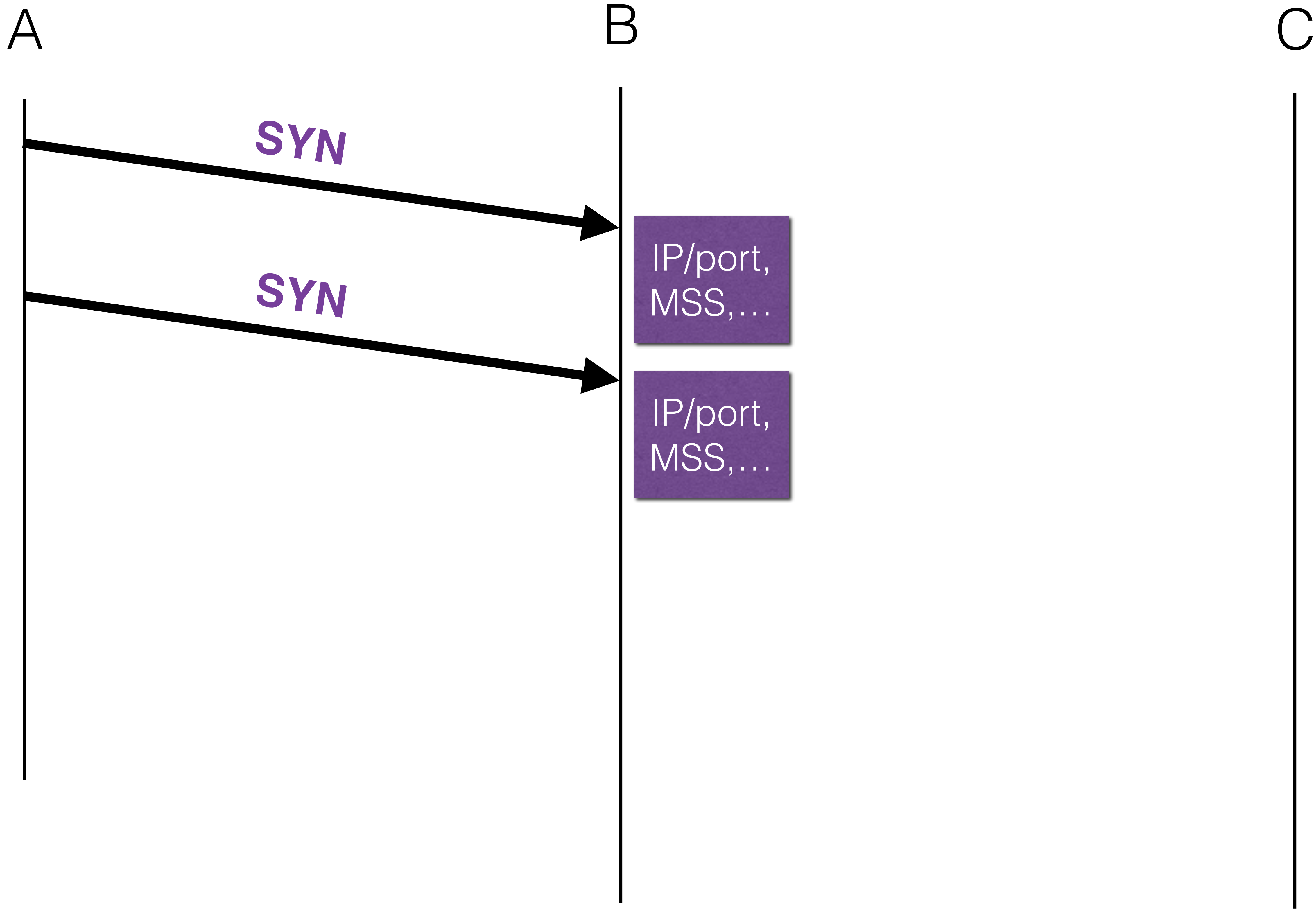
SYN flooding

The attack



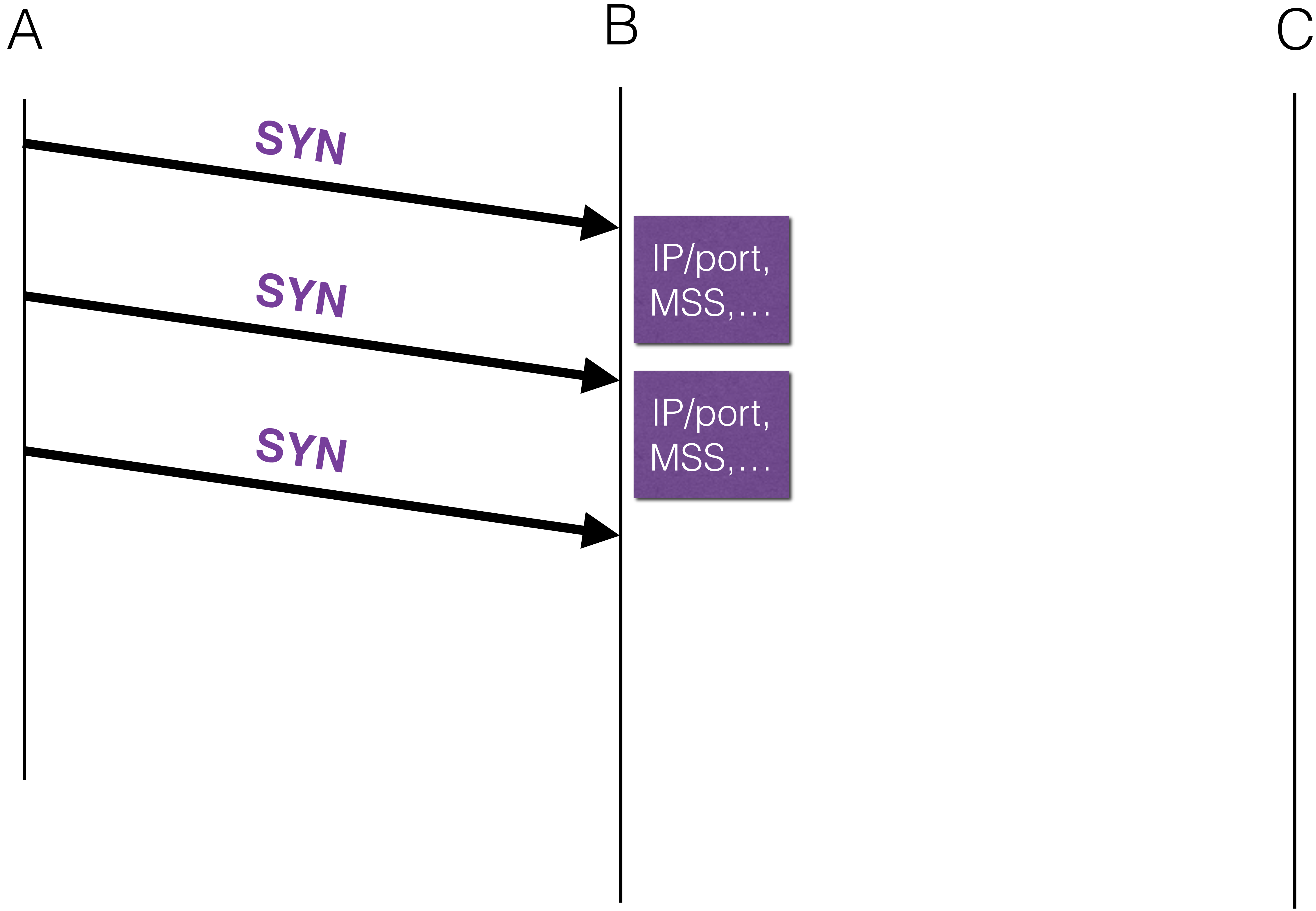
SYN flooding

The attack



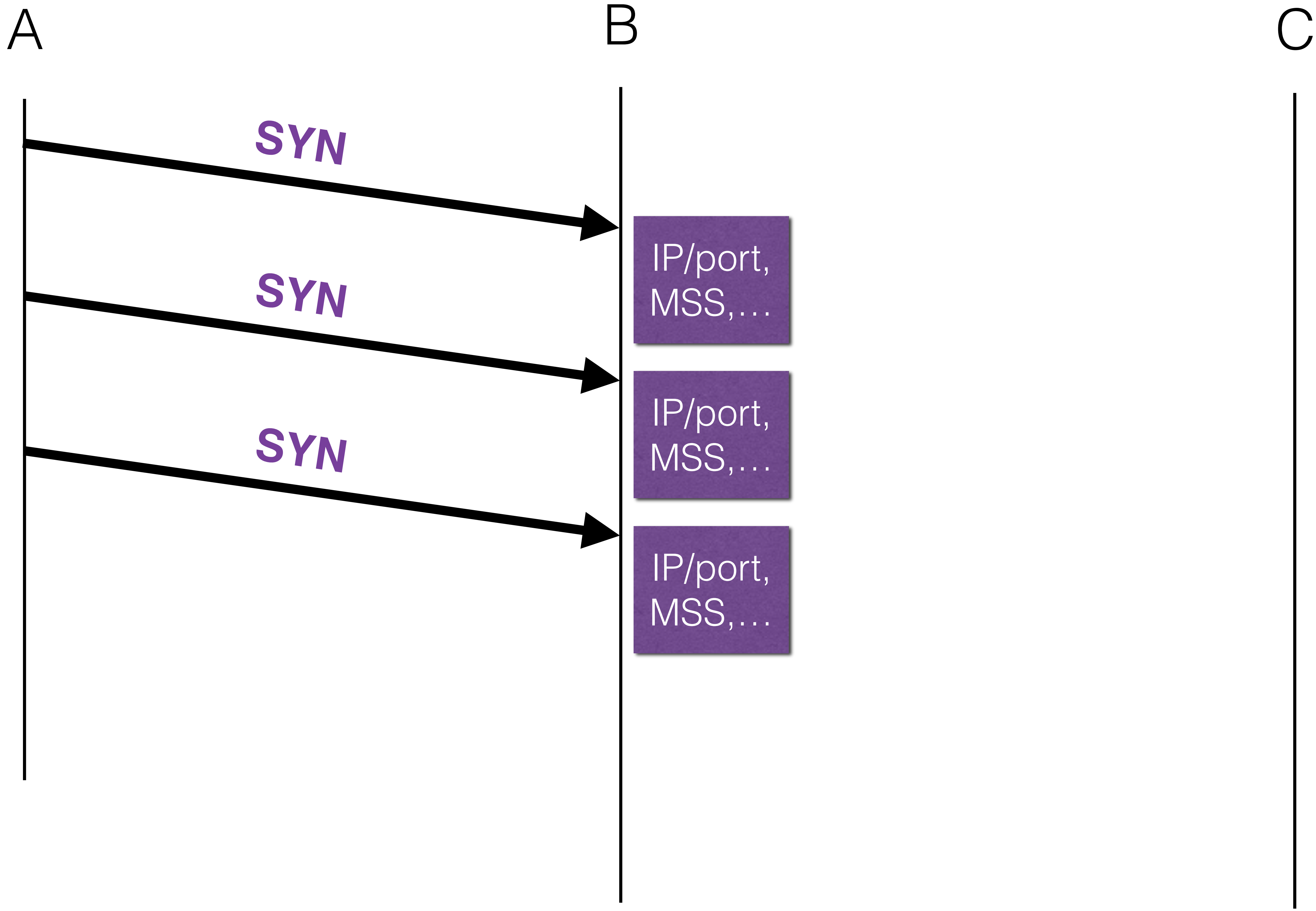
SYN flooding

The attack



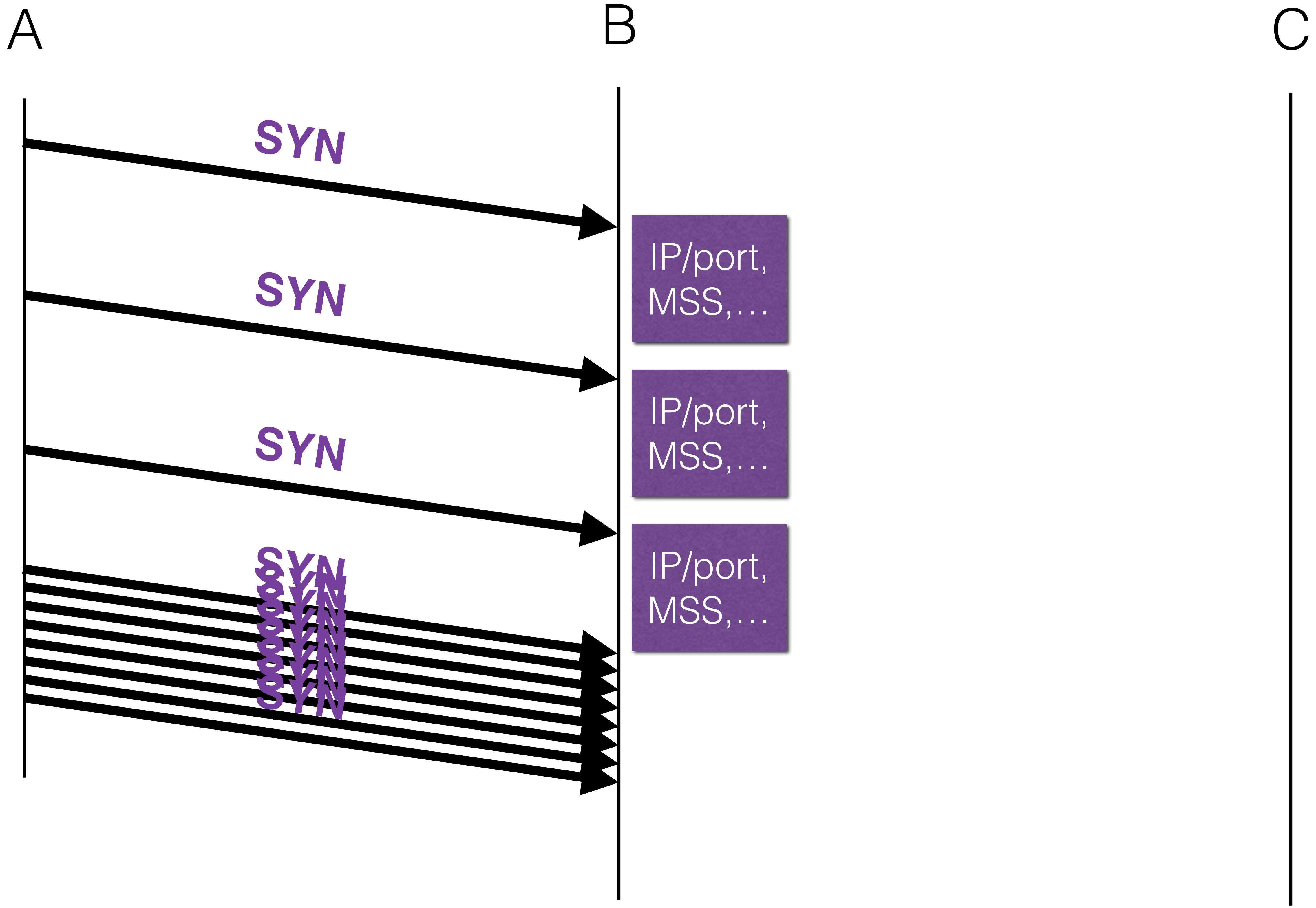
SYN flooding

The attack



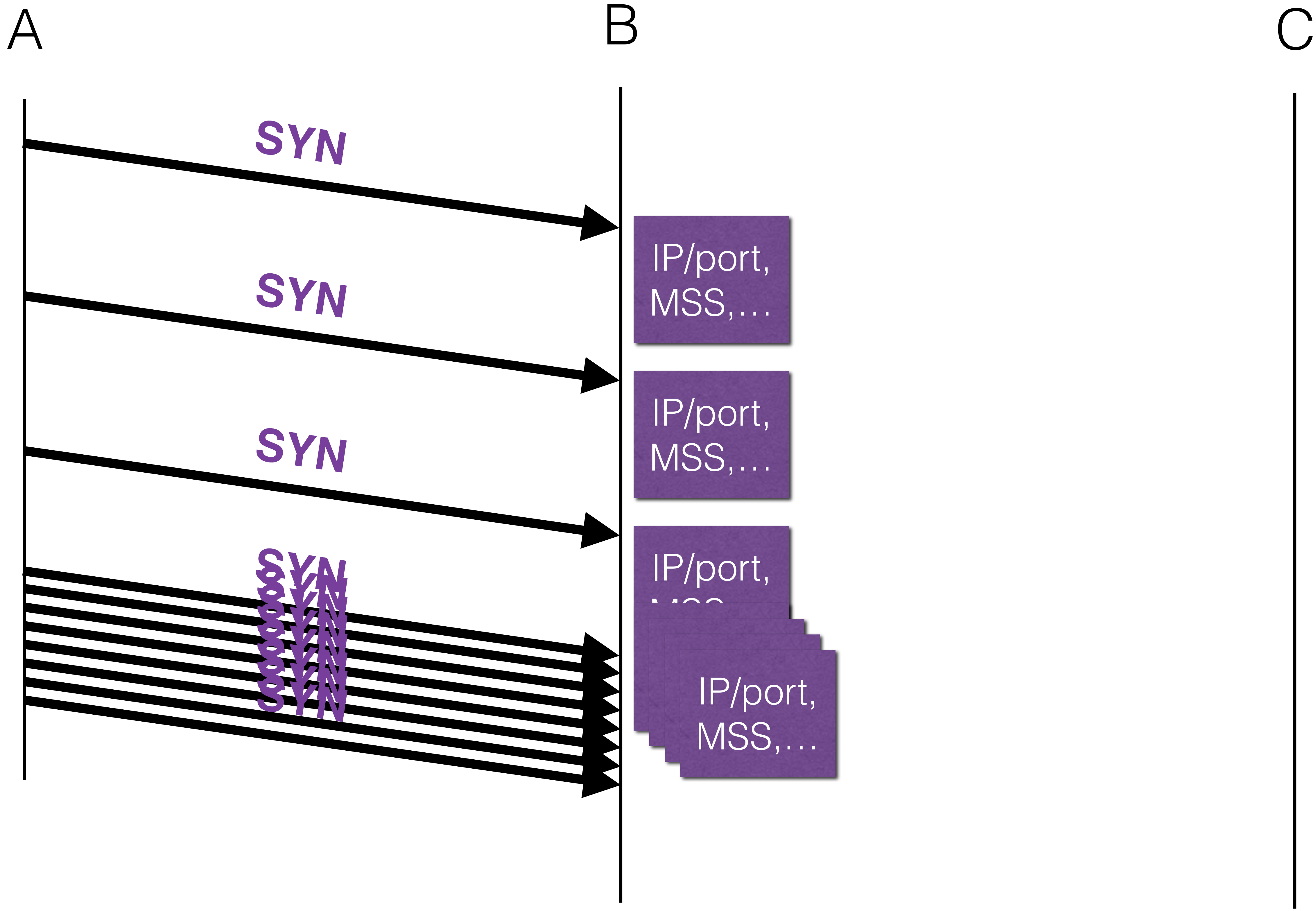
SYN flooding

The attack



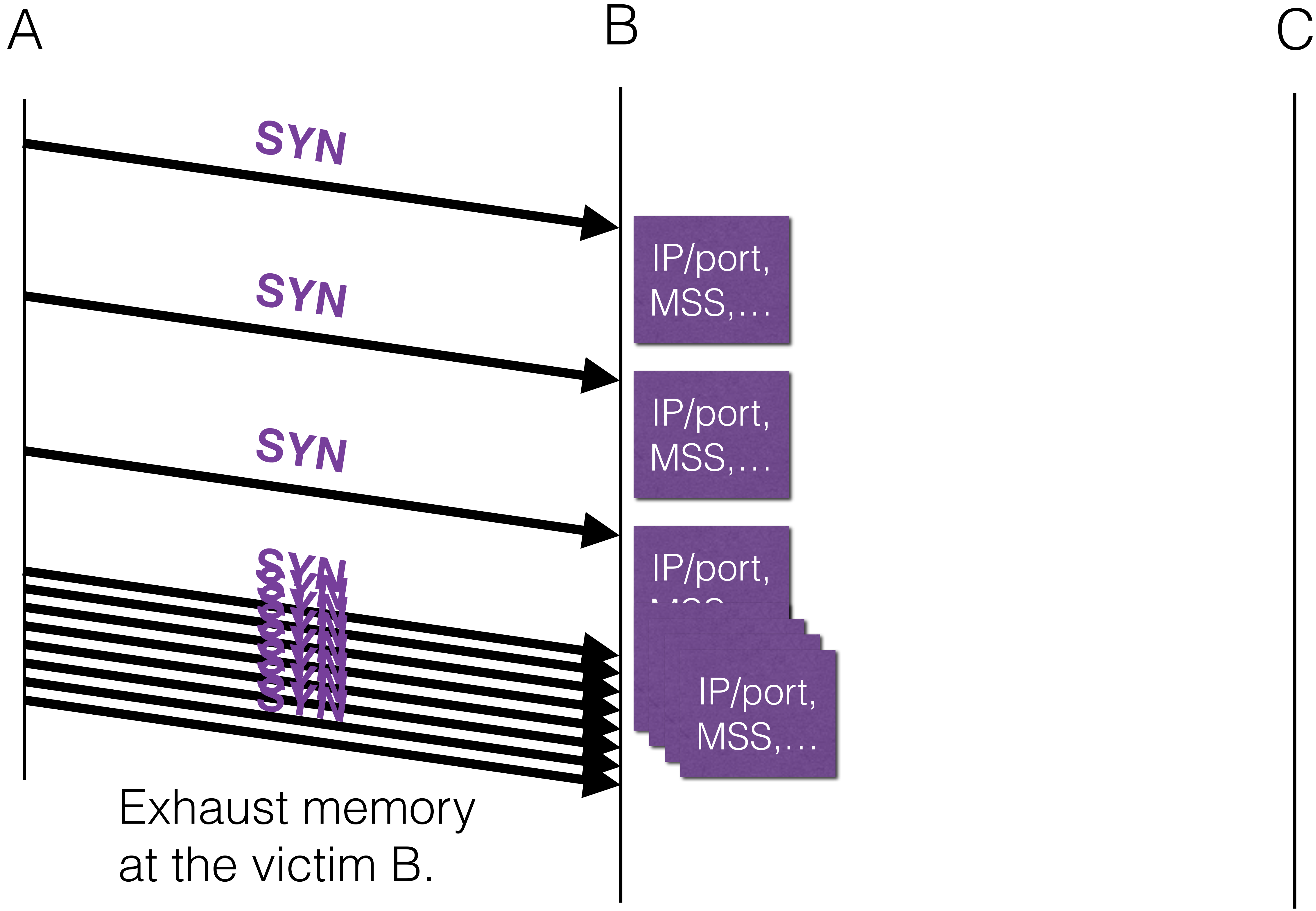
SYN flooding

The attack



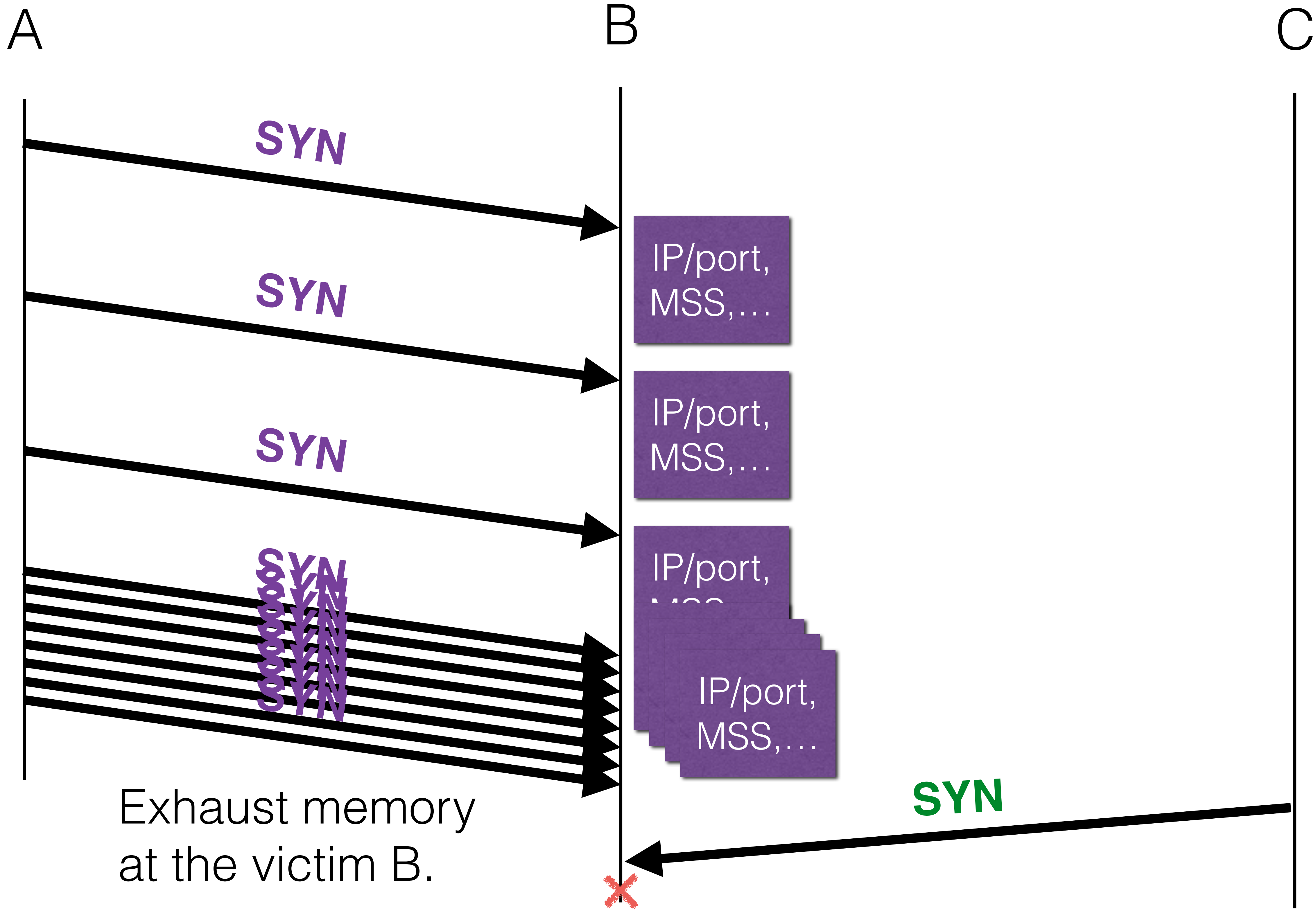
SYN flooding

The attack



SYN flooding

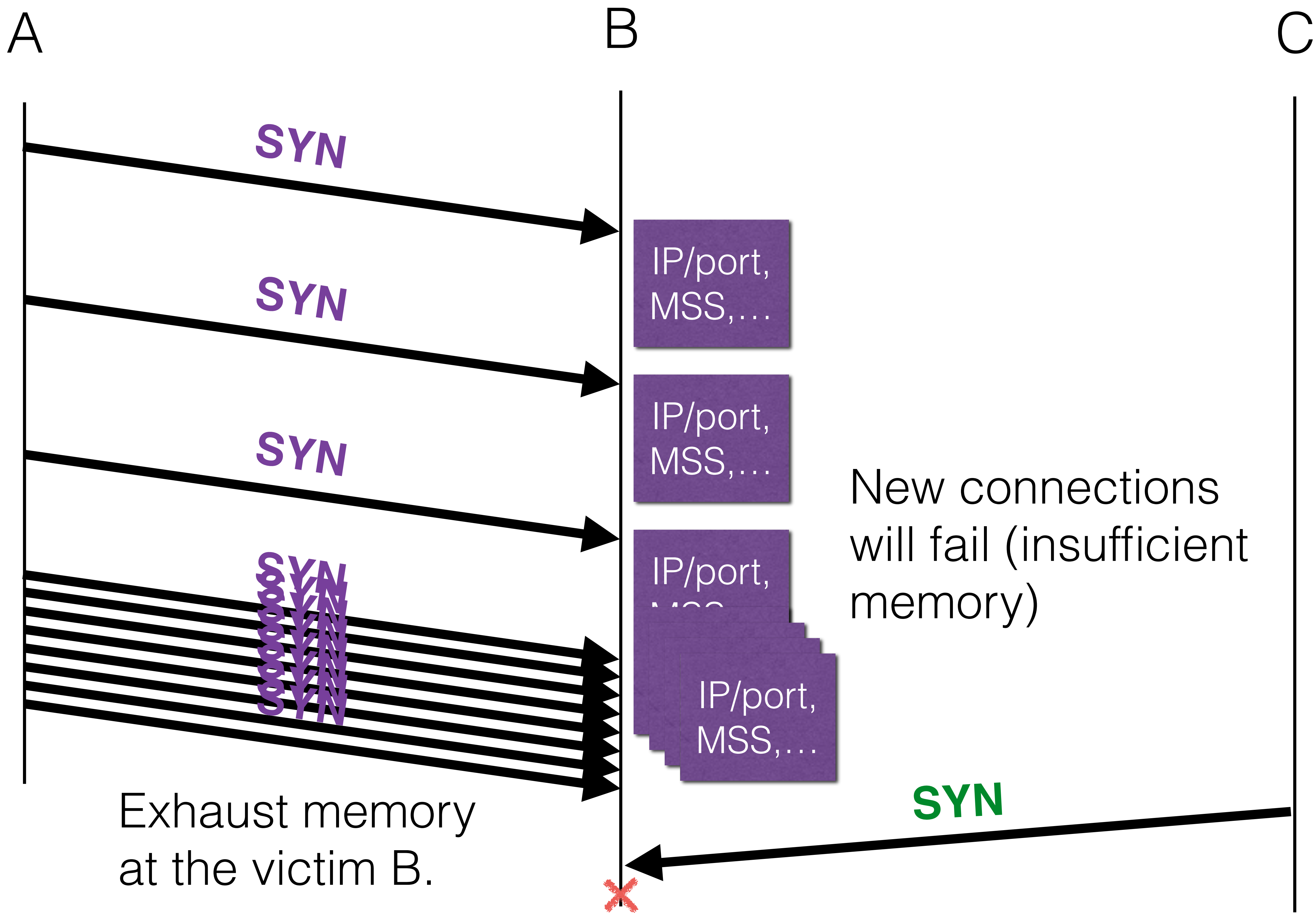
The attack



Exhaust memory
at the victim B.

SYN flooding

The attack

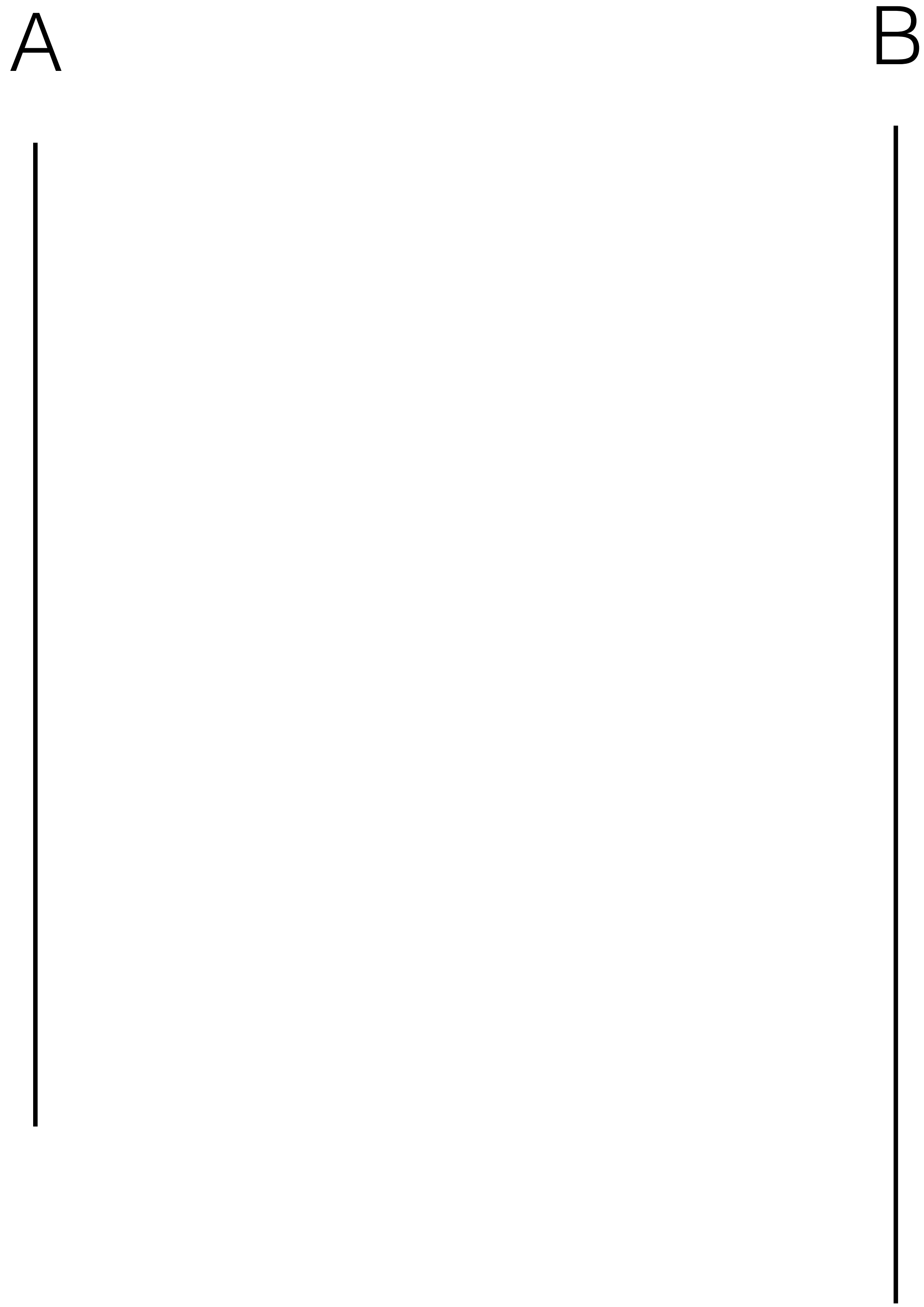


SYN flooding details

- 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

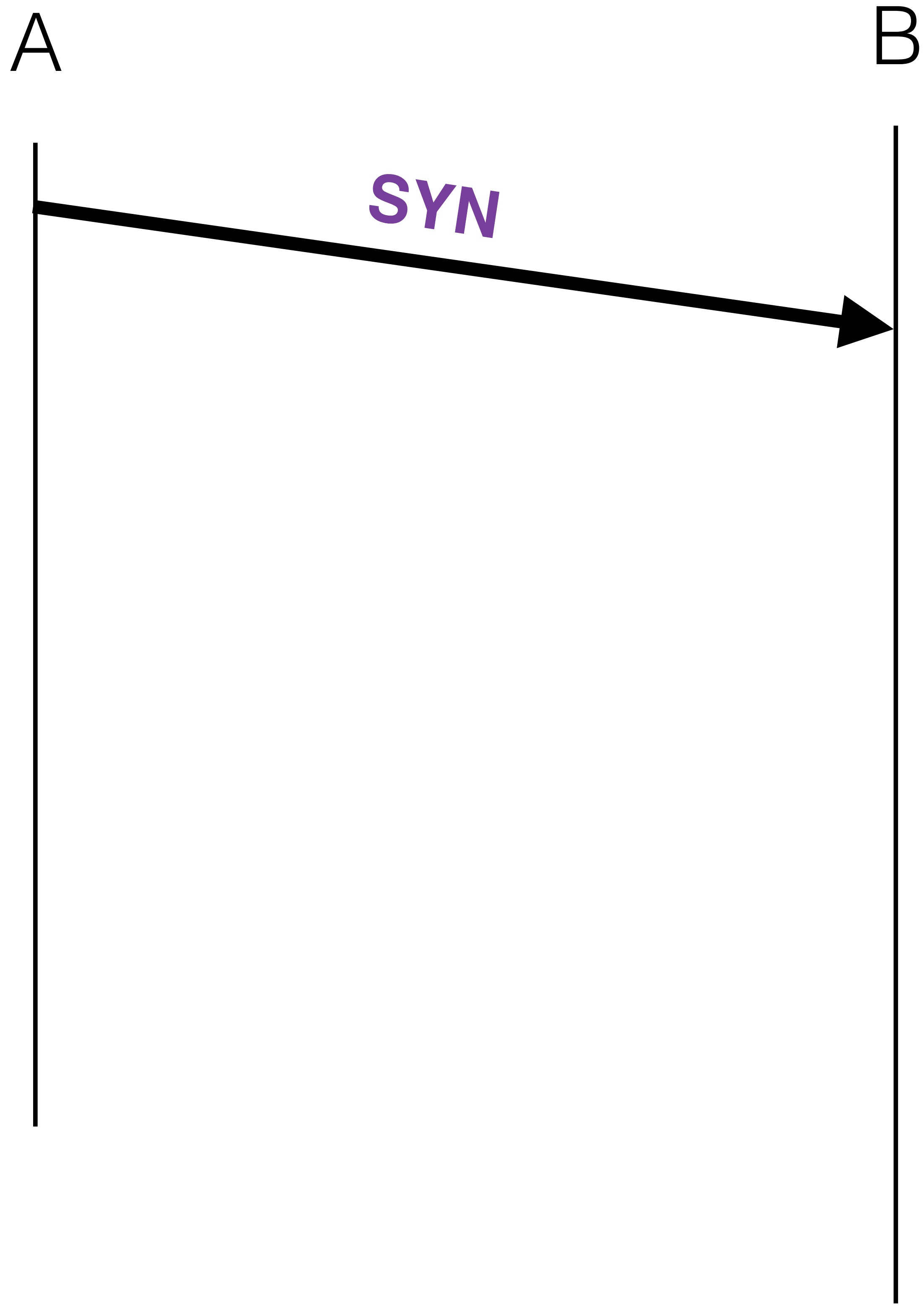
SYN cookies

The defense



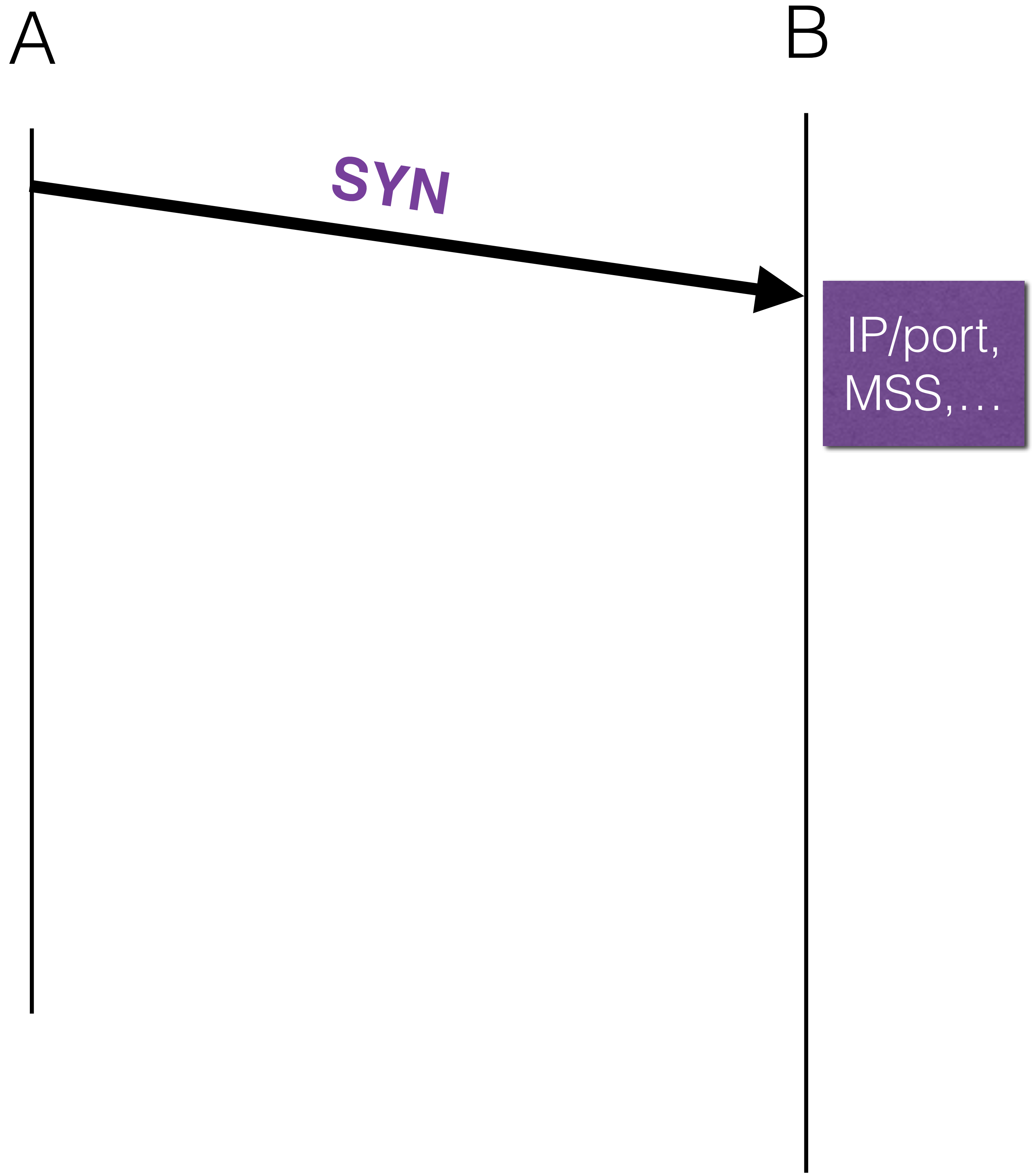
SYN cookies

The defense



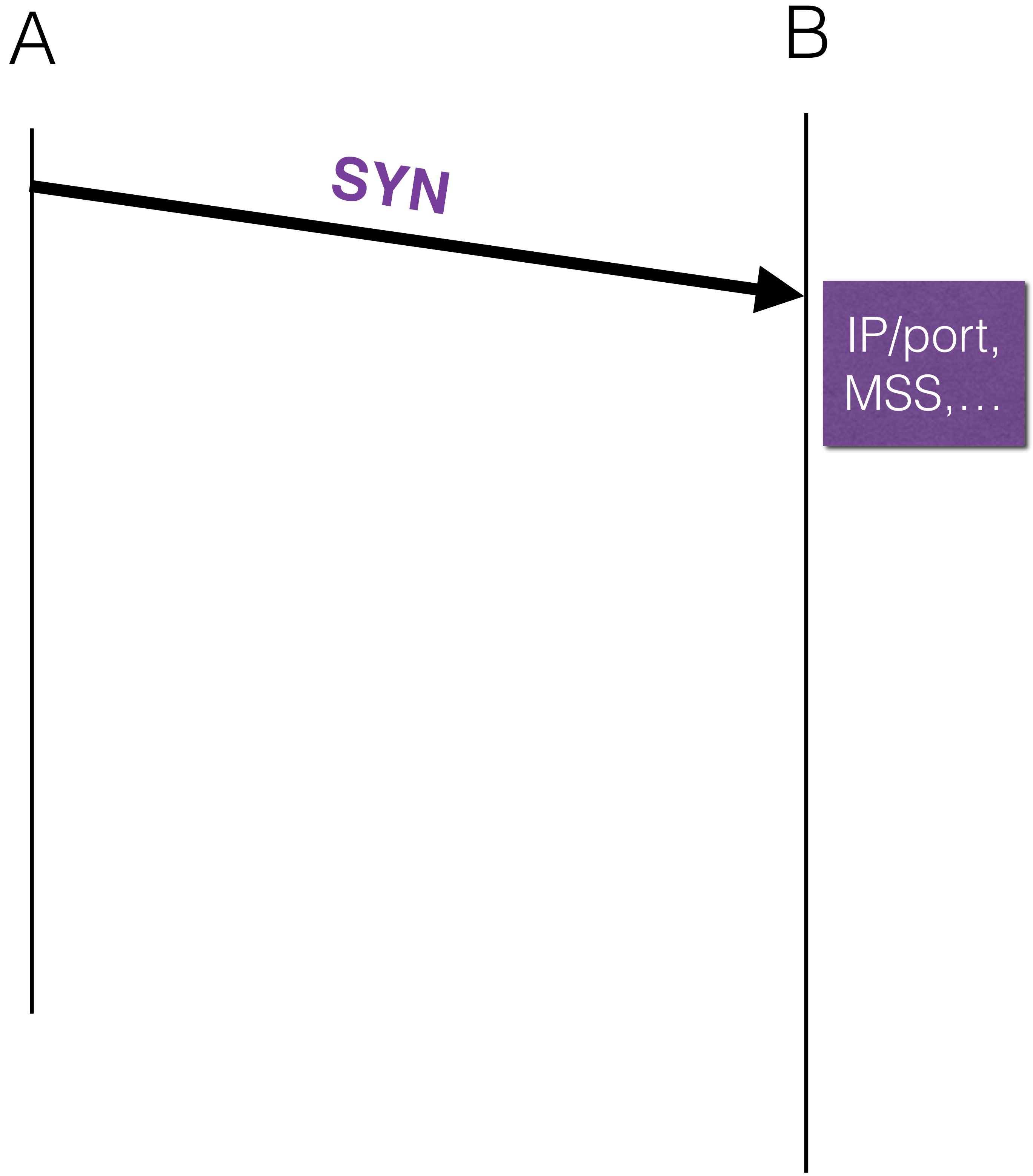
SYN cookies

The defense



SYN cookies

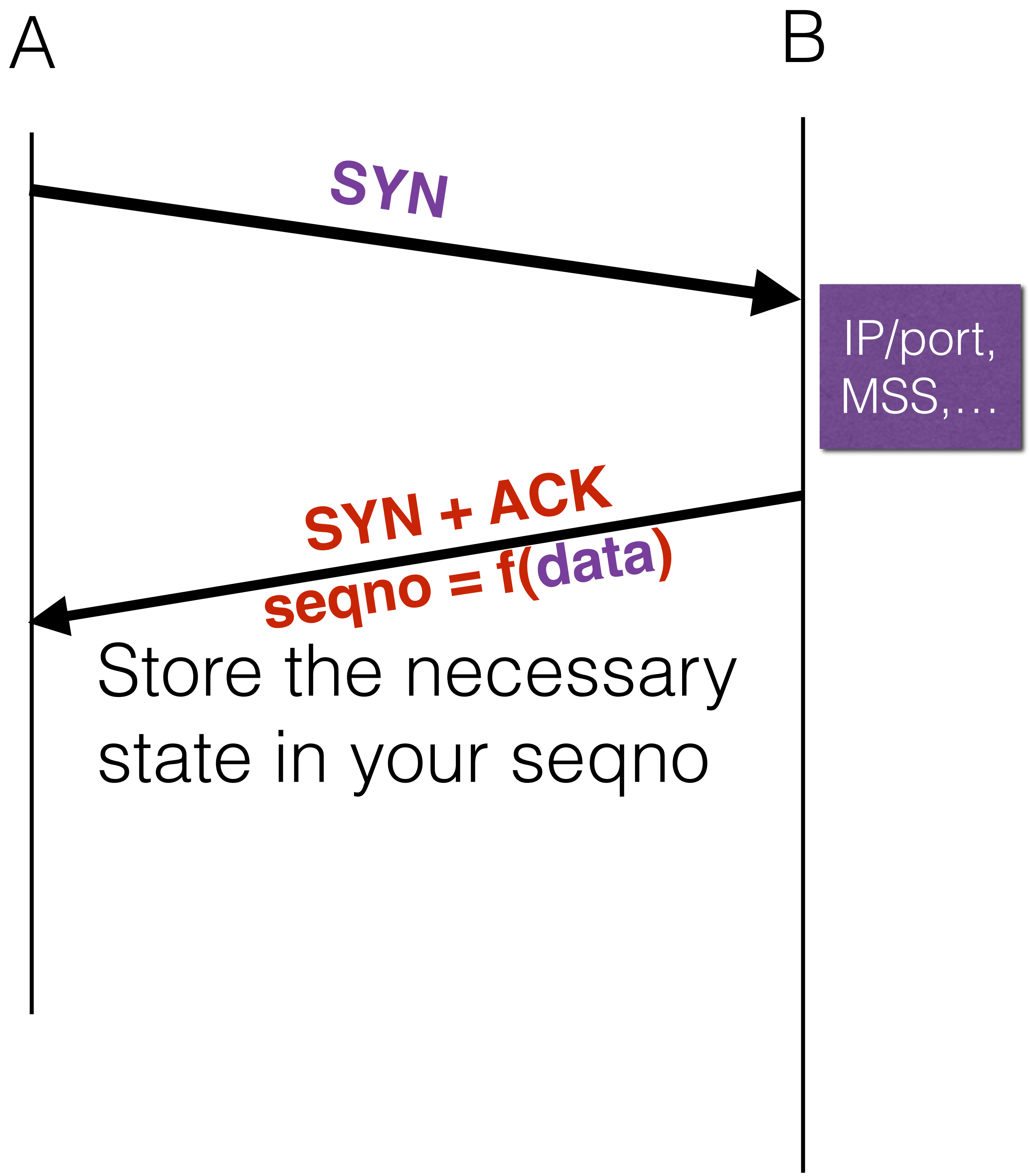
The defense



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

SYN cookies

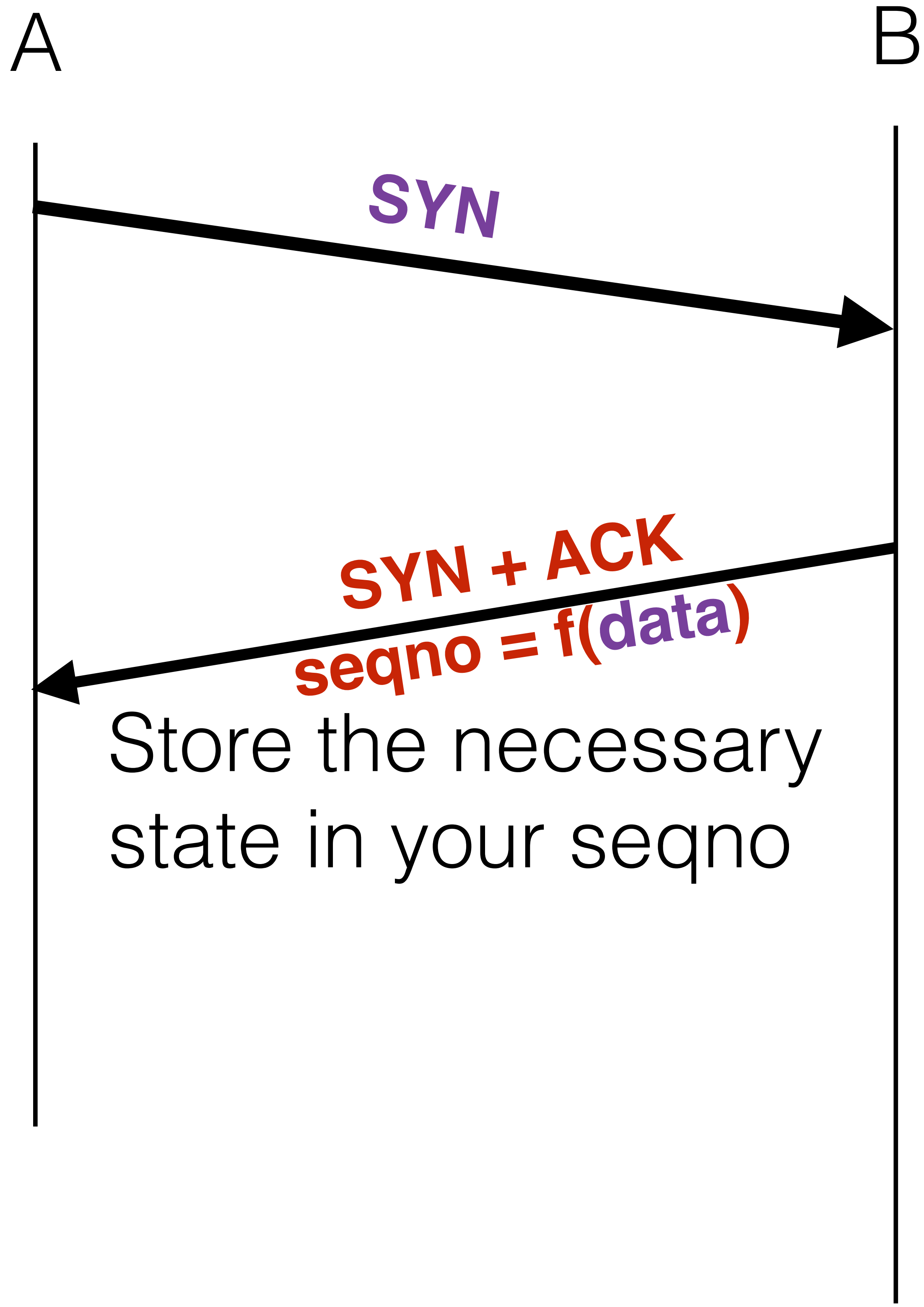
The defense



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

SYN cookies

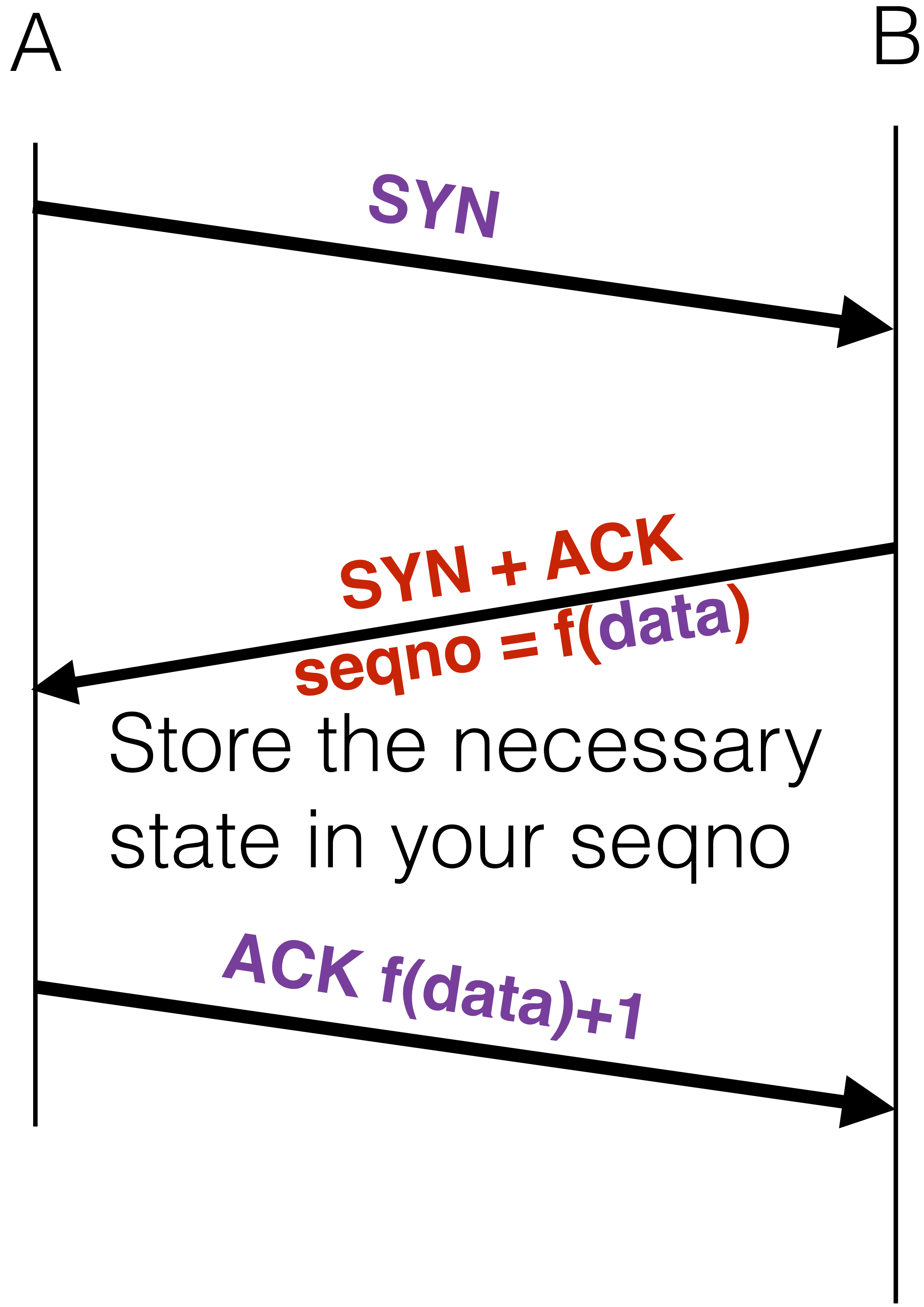
The defense



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

SYN cookies

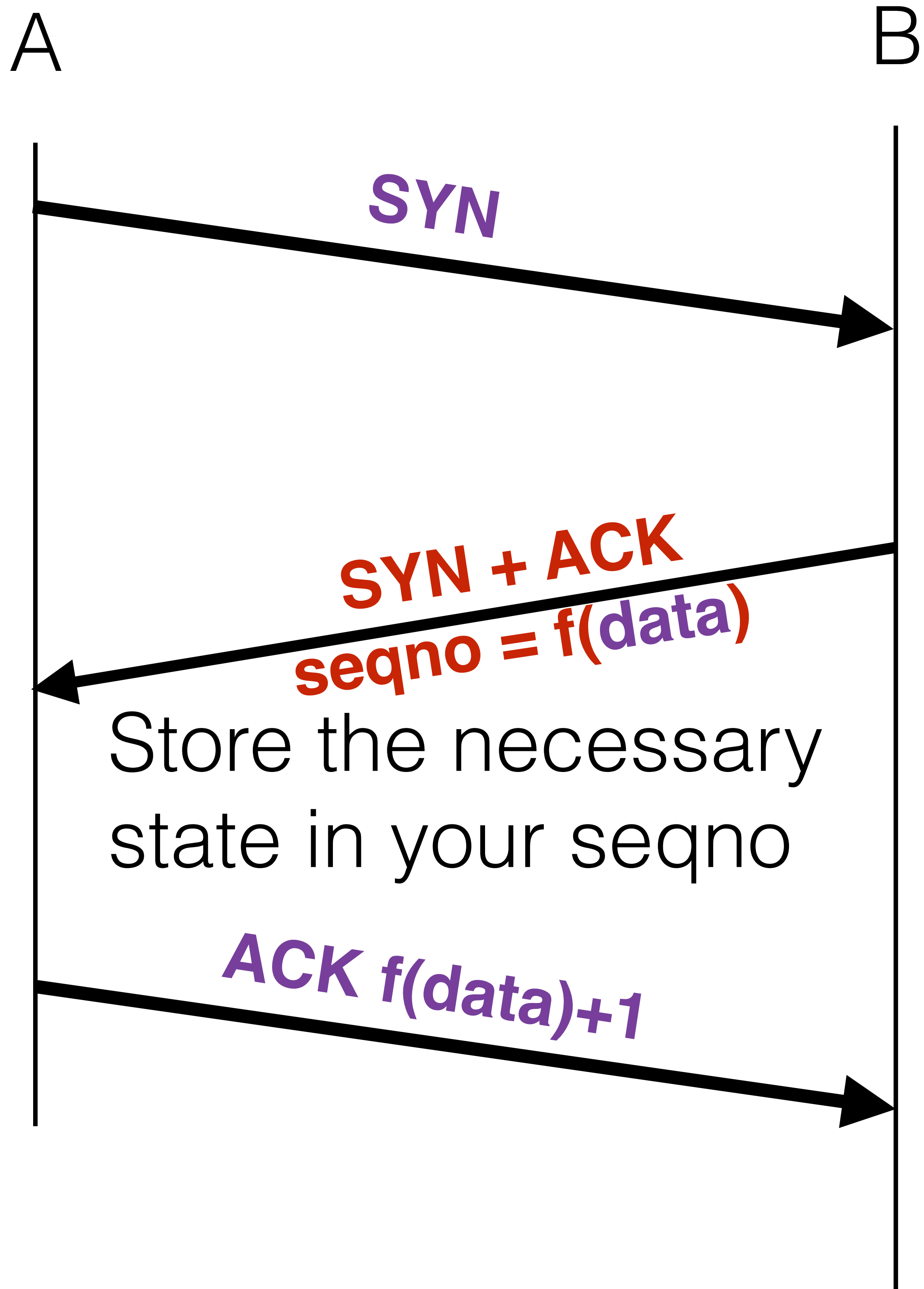
The defense



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

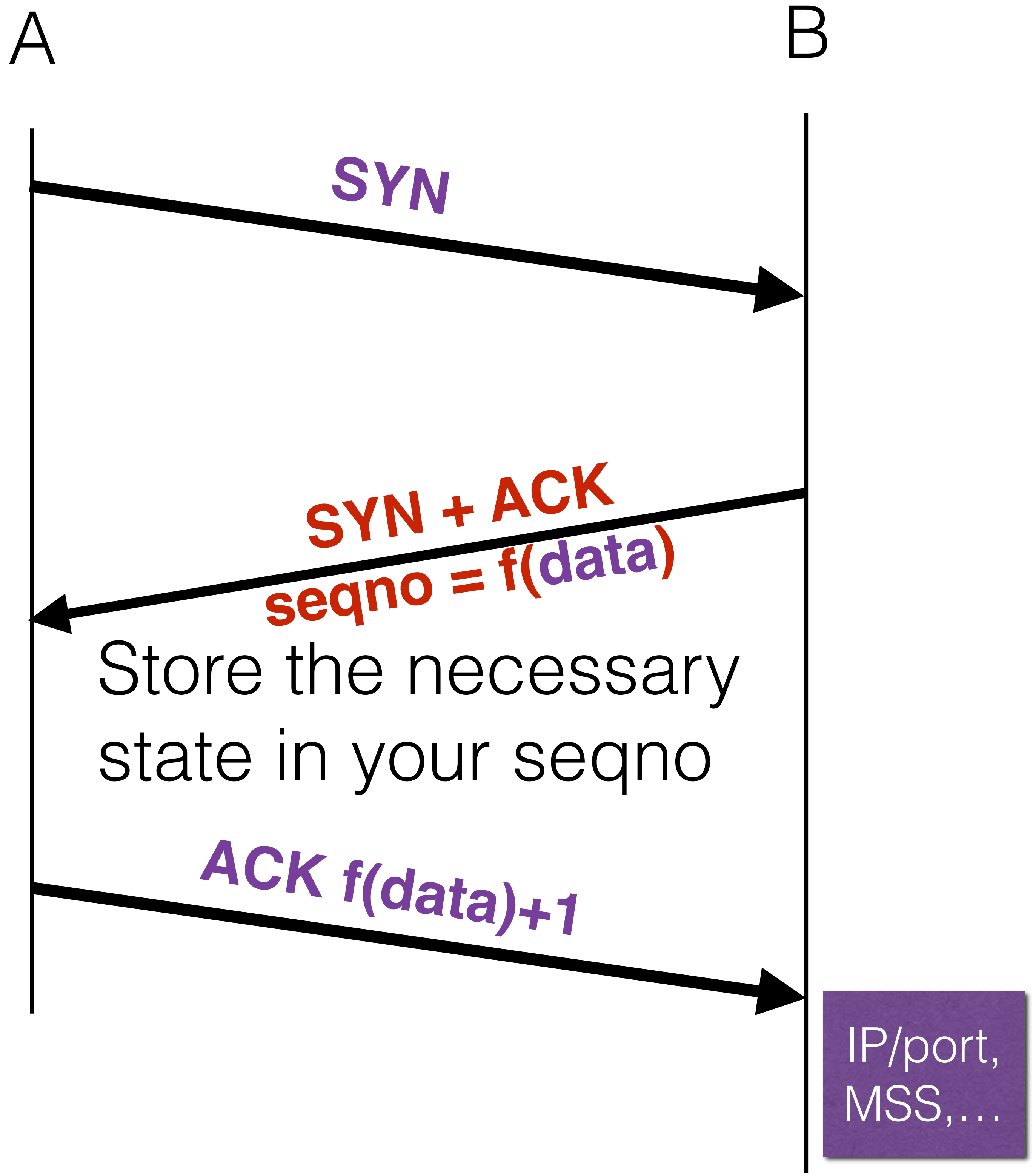


Rather than store this data, send it to the host who is initiating the connection and have him return it to you

Check that f(data) is valid for this connection. Only at that point do you allocate state.

SYN cookies

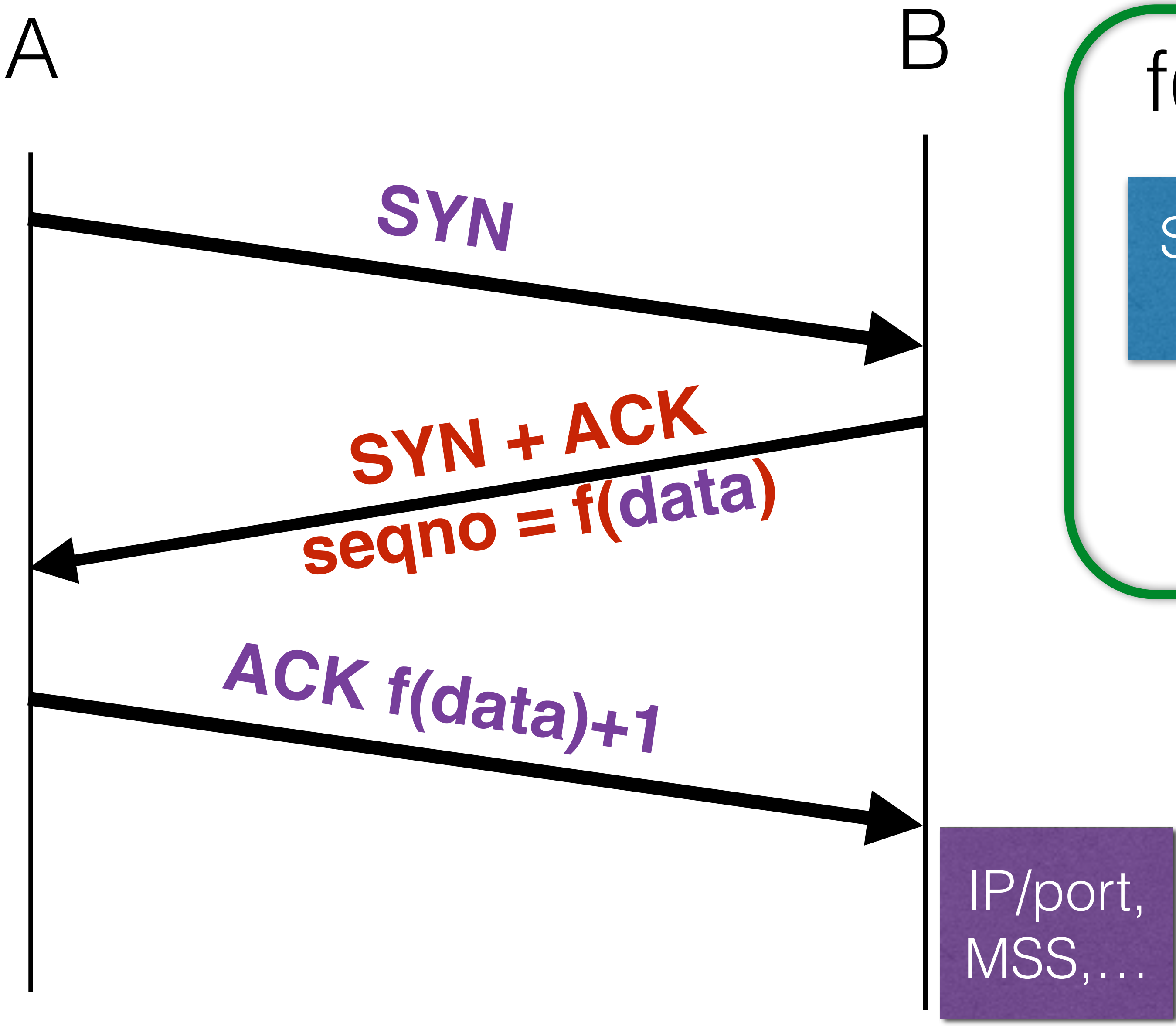
The defense



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(data) is valid for this connection. Only at that point do you allocate state.

SYN cookie format



f(.) = 32-bit seqno

Slow-moving timestamp	MSS	Secure hash
Prevents replay attacks	The info we need for this connection	Includes: IPs/ports, MSS, timestamp

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

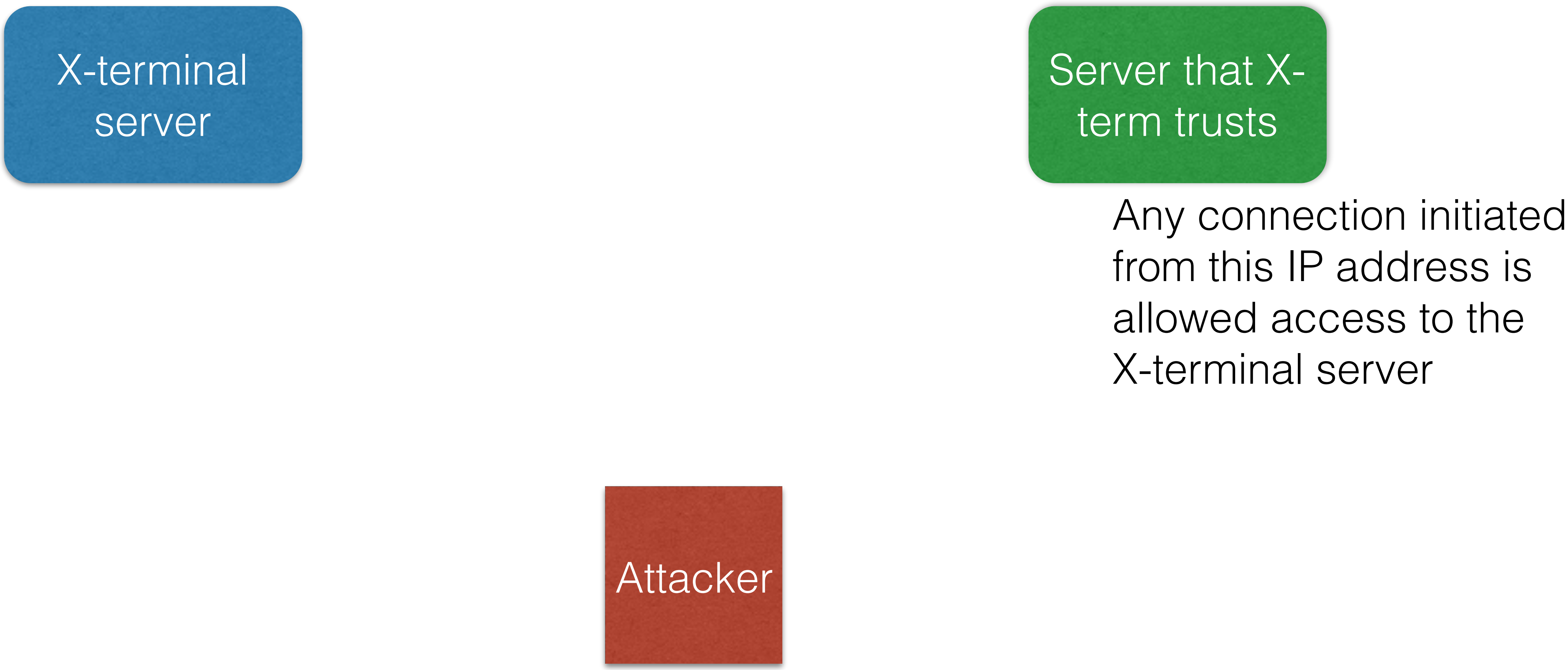
- Suppose you are on the path between src and dst; what can you do?
  - Trivial to inject packets with the correct sequence number

- What if you are not on the path?
  - Need to guess the sequence number
  - Is this difficult to do?

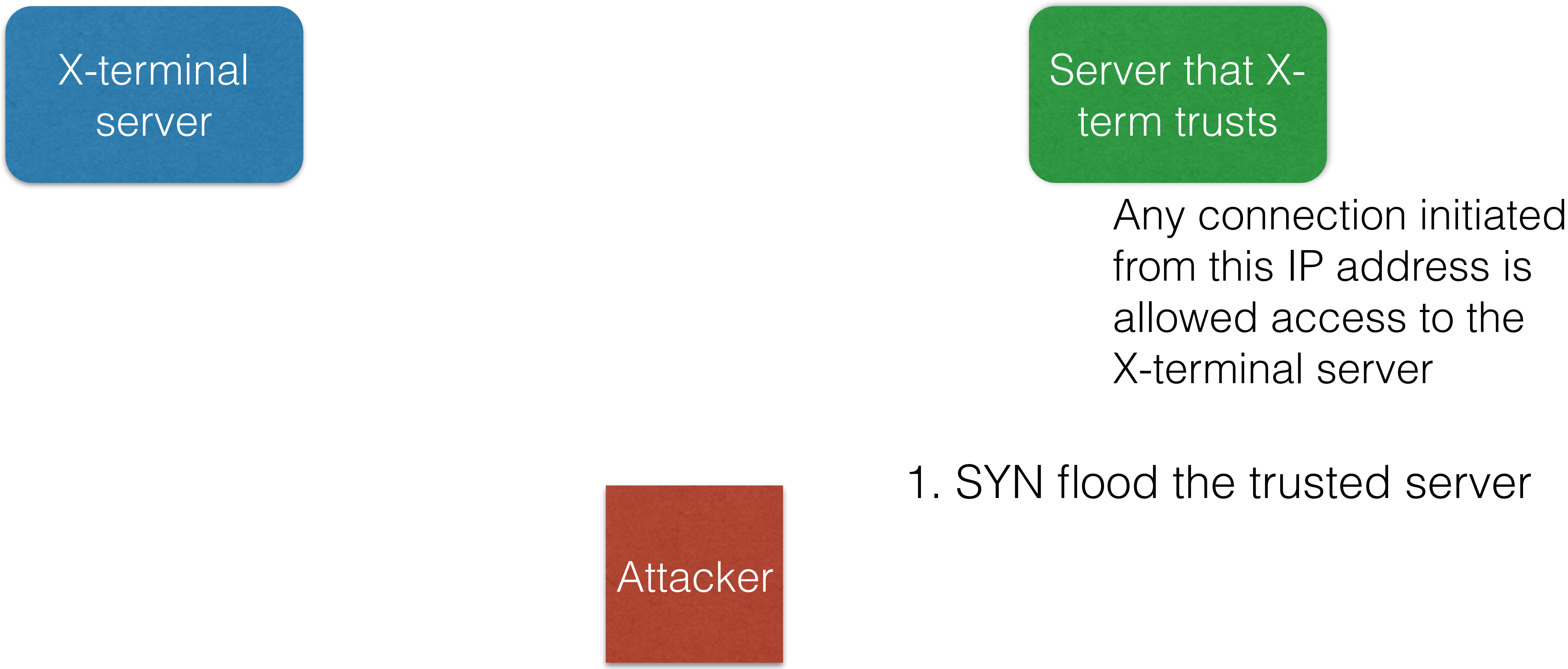
Initial sequence numbers

- 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*

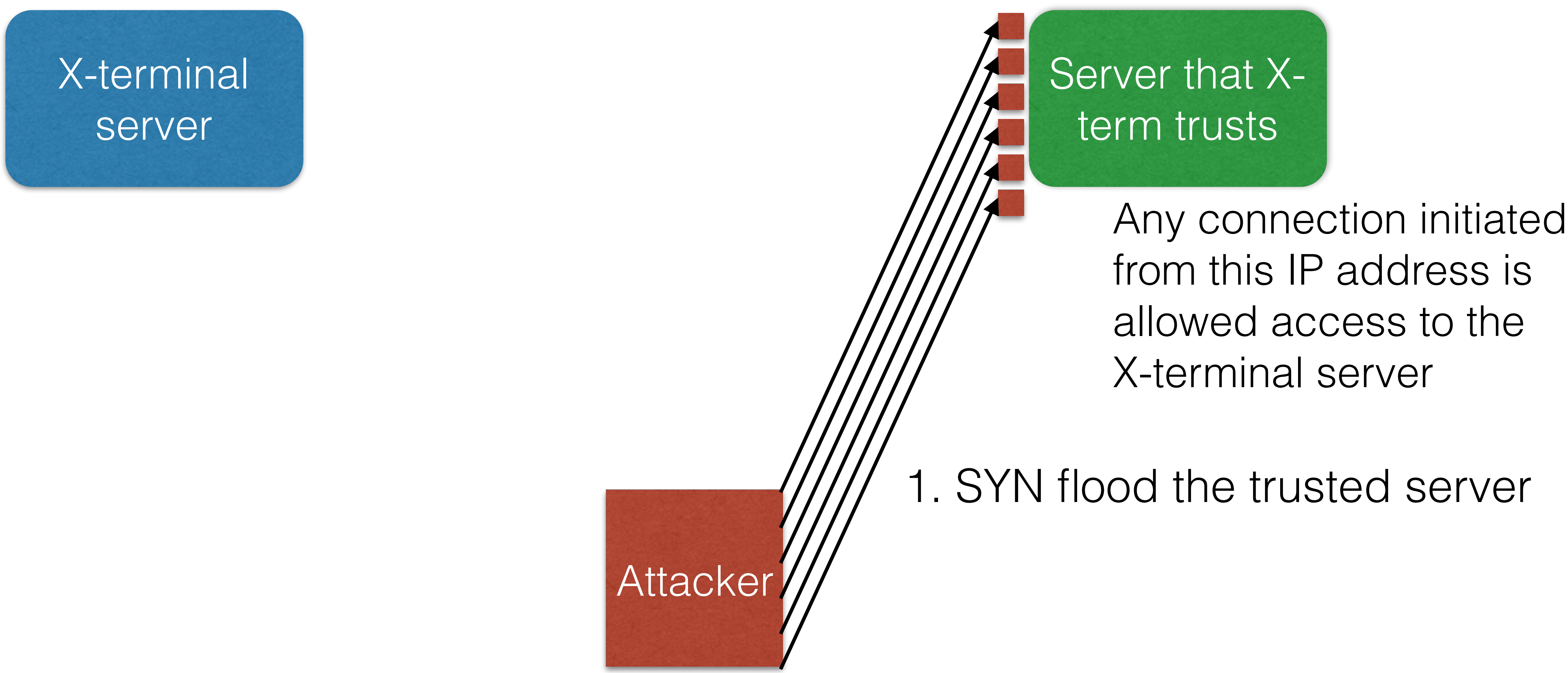
Mitnick attack



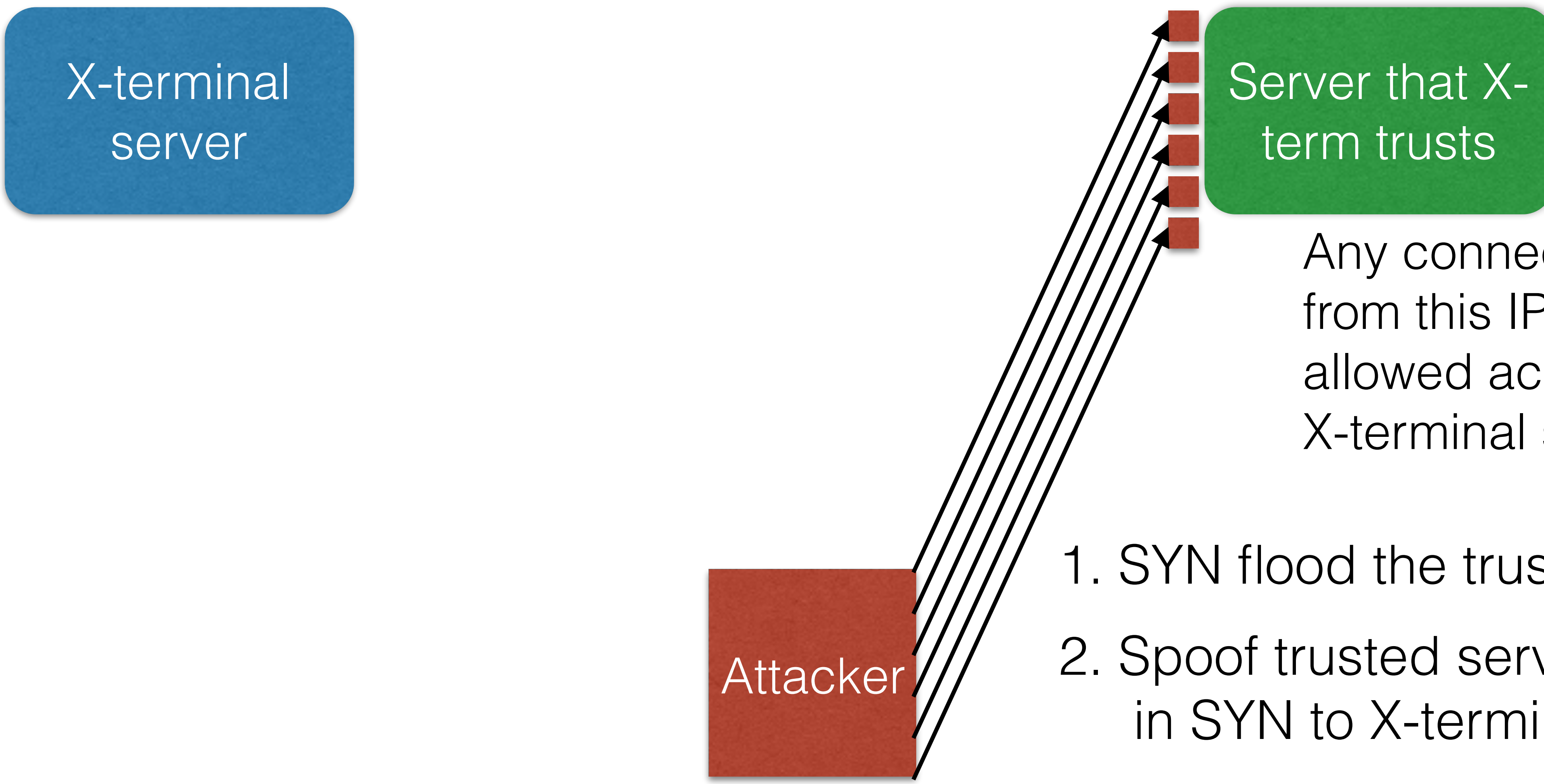
Mitnick attack



Mitnick attack



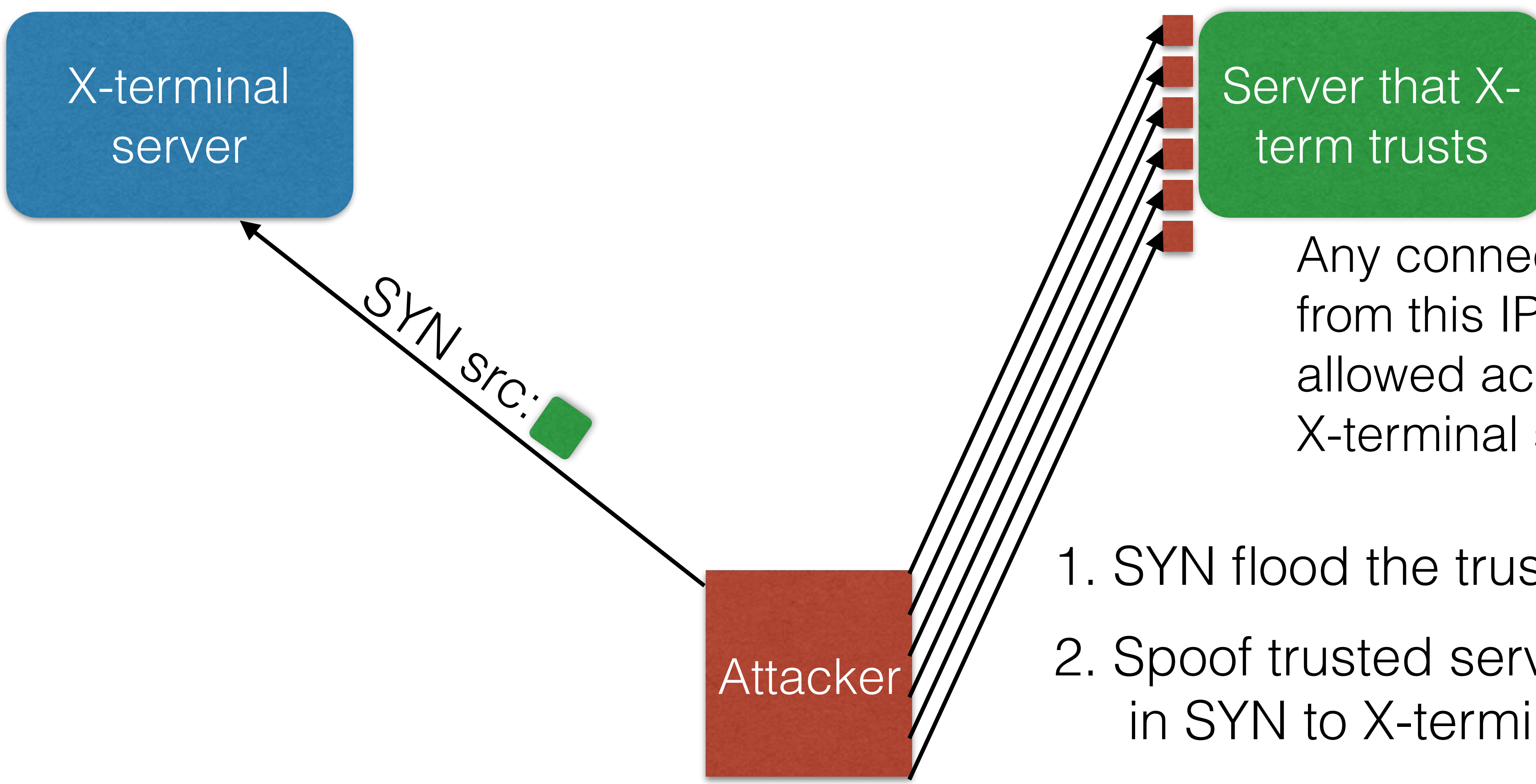
Mitnick attack



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

1. SYN flood the trusted server
2. Spoof trusted server's IP addr in SYN to X-terminal

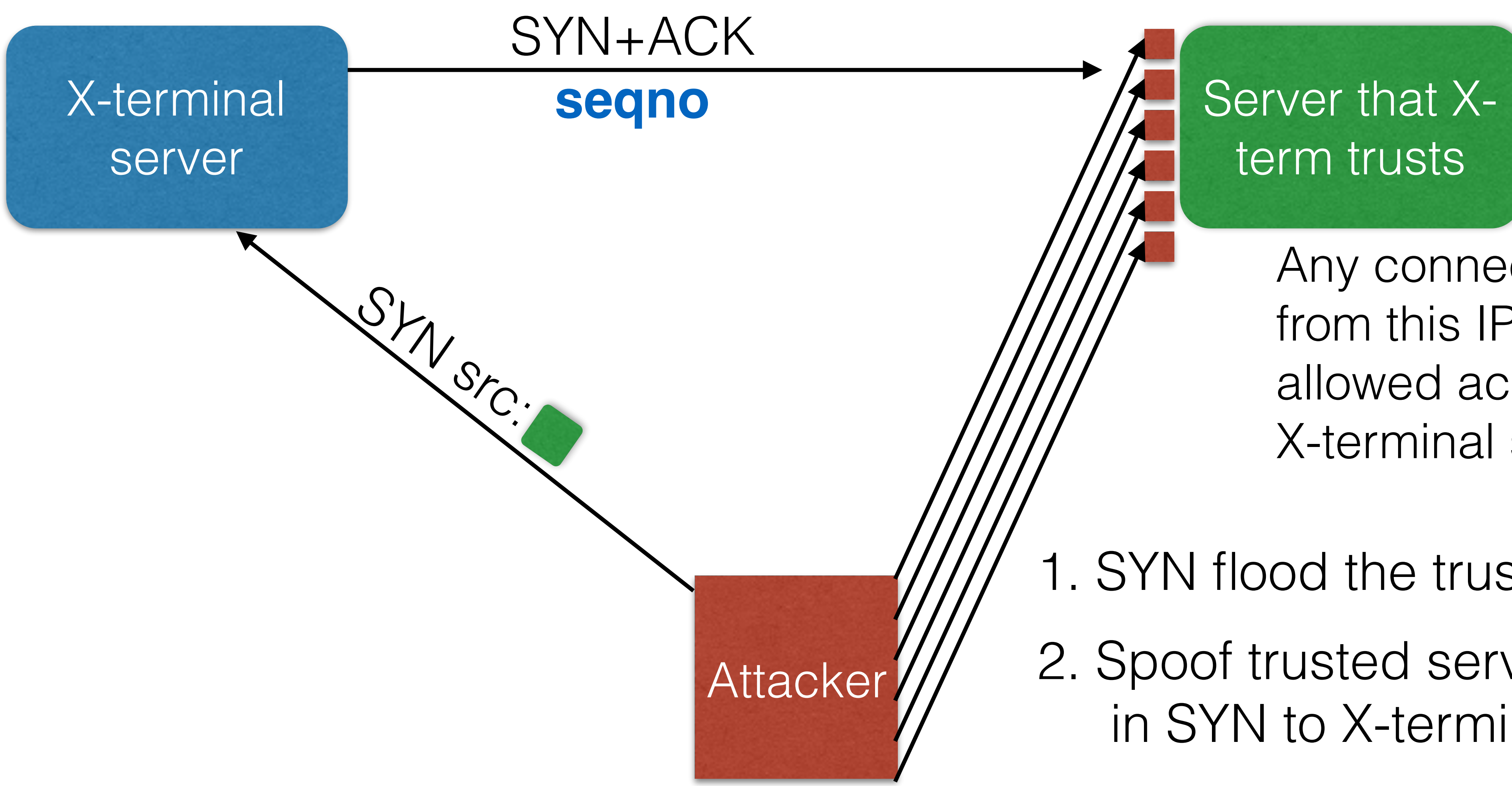
Mitnick attack



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

1. SYN flood the trusted server
2. Spoof trusted server's IP addr in SYN to X-terminal

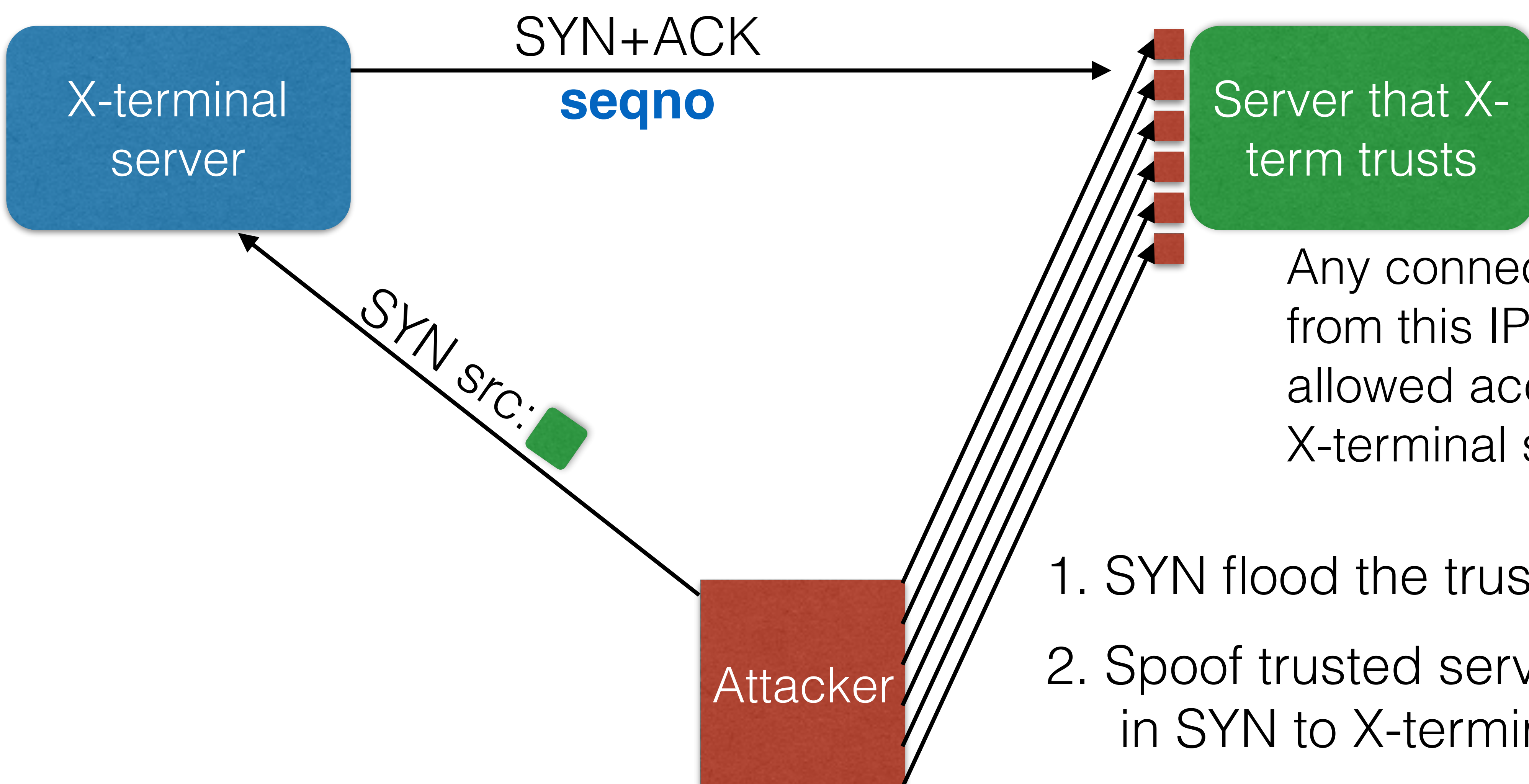
Mitnick attack



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

1. SYN flood the trusted server
2. Spoof trusted server's IP addr in SYN to X-terminal

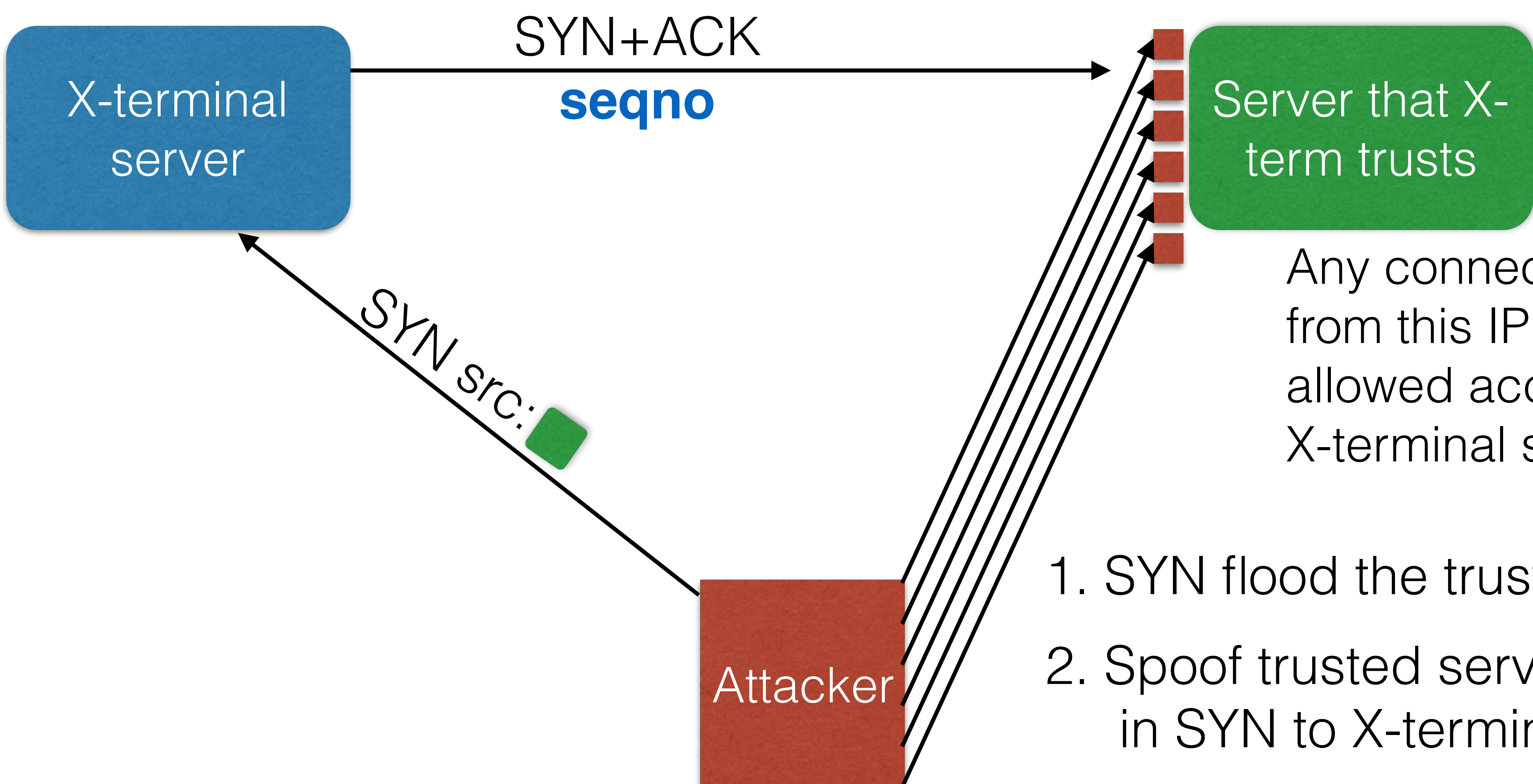
Mitnick attack



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

1. SYN flood the trusted server
2. Spoof trusted server's IP addr in SYN to X-terminal
3. Trusted server too busy to RST

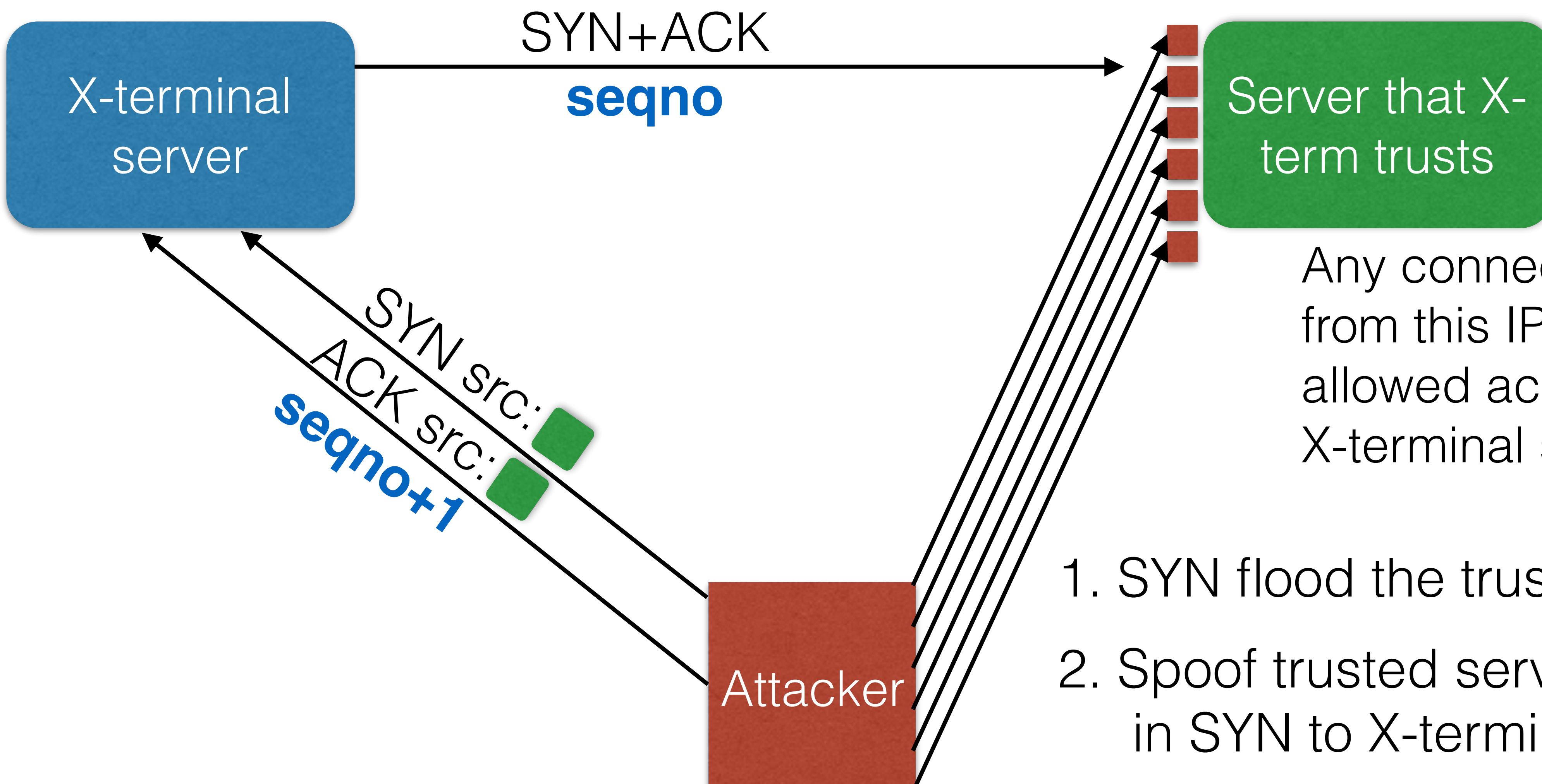
Mitnick attack



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

1. SYN flood the trusted server
2. Spoof trusted server's IP addr in SYN to X-terminal
3. Trusted server too busy to RST
4. ACK with the guessed **seqno**

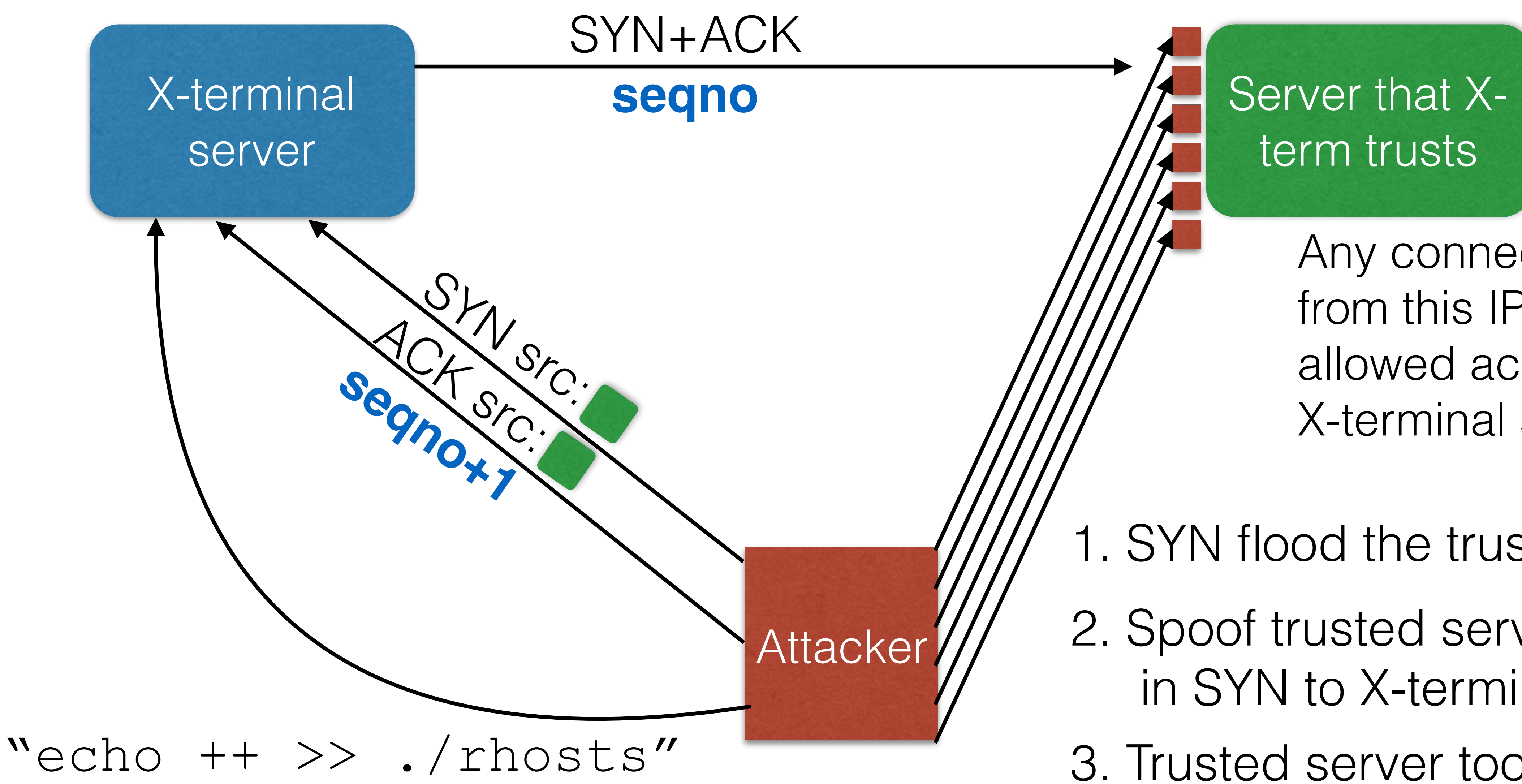
Mitnick attack



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

1. SYN flood the trusted server
2. Spoof trusted server's IP addr in SYN to X-terminal
3. Trusted server too busy to RST
4. ACK with the guessed **seqno**

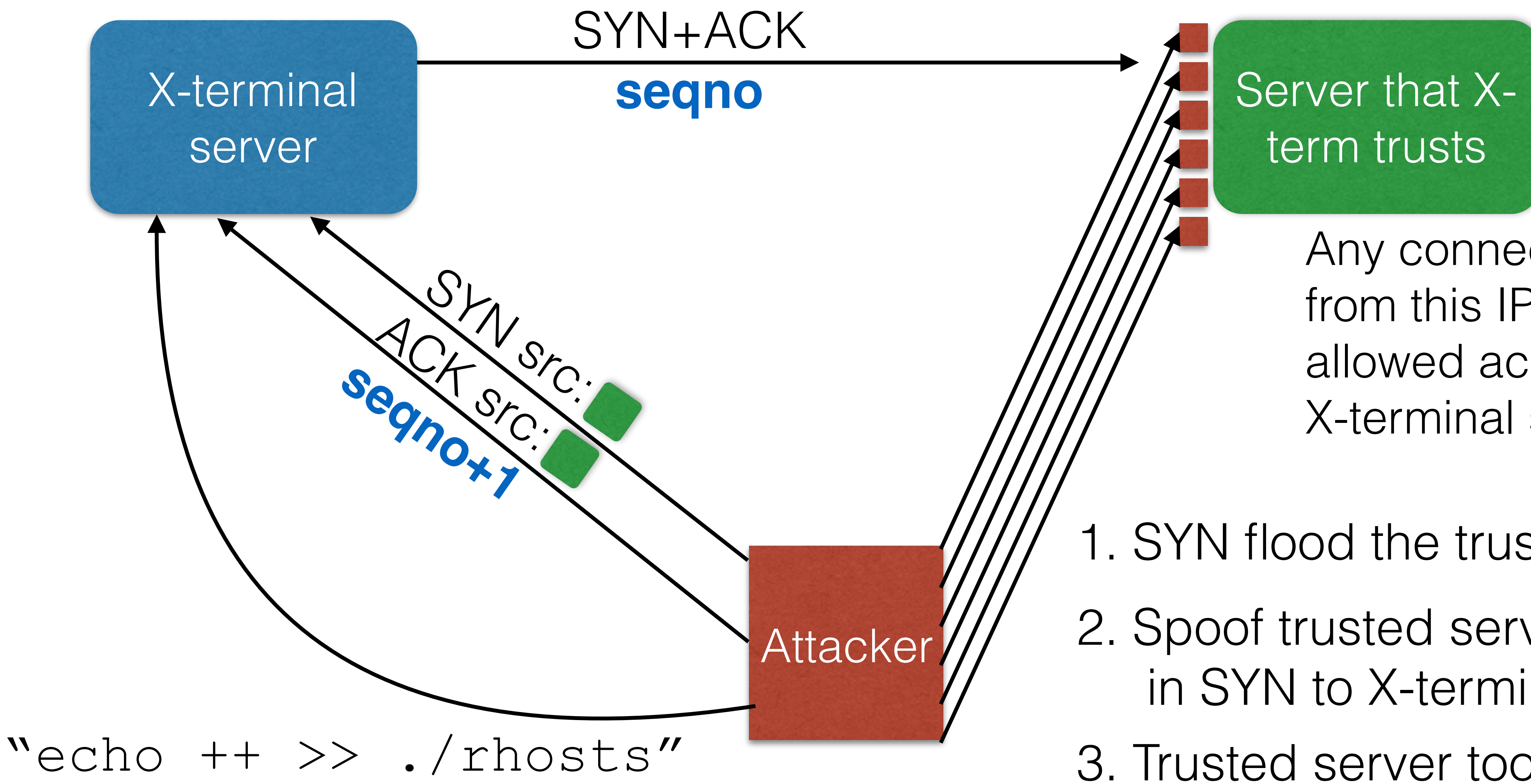
Mitnick attack



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

1. SYN flood the trusted server
2. Spoof trusted server's IP addr in SYN to X-terminal
3. Trusted server too busy to RST
4. ACK with the guessed **seqno**

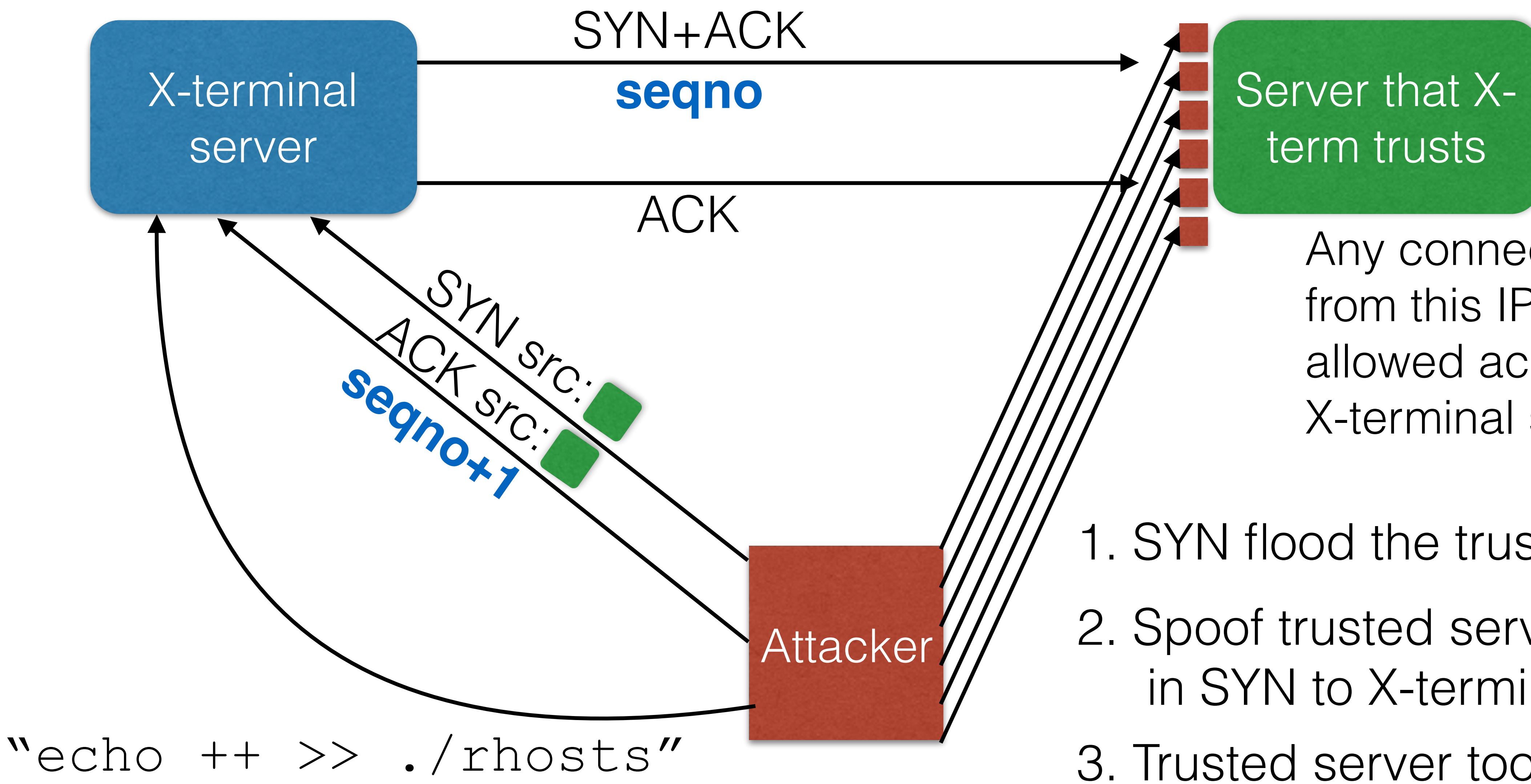
Mitnick attack



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

1. SYN flood the trusted server
2. Spoof trusted server's IP addr in SYN to X-terminal
3. Trusted server too busy to RST
4. ACK with the guessed **seqno**
5. Grant access to all sources

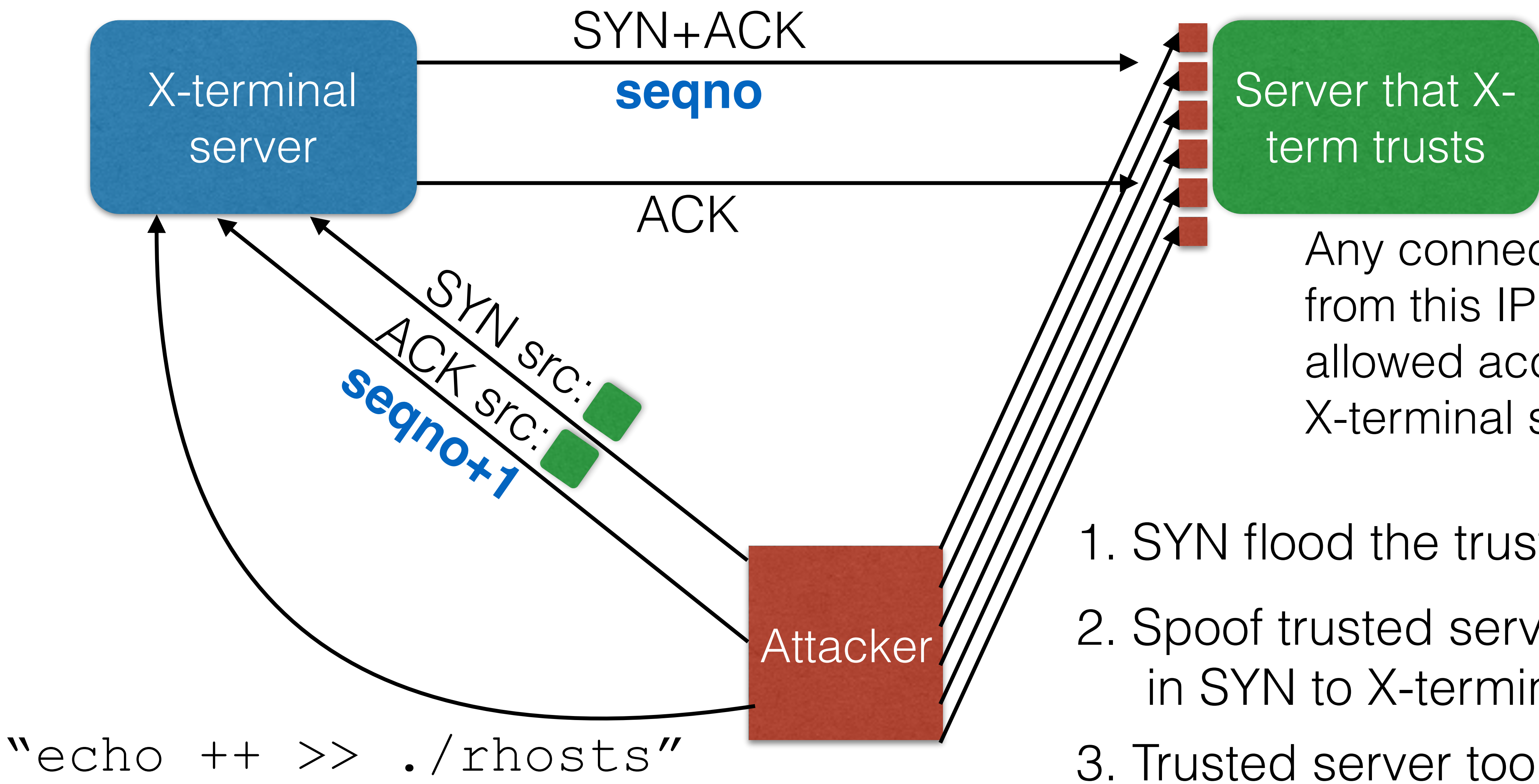
Mitnick attack



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

1. SYN flood the trusted server
2. Spoof trusted server's IP addr in SYN to X-terminal
3. Trusted server too busy to RST
4. ACK with the guessed **seqno**
5. Grant access to all sources

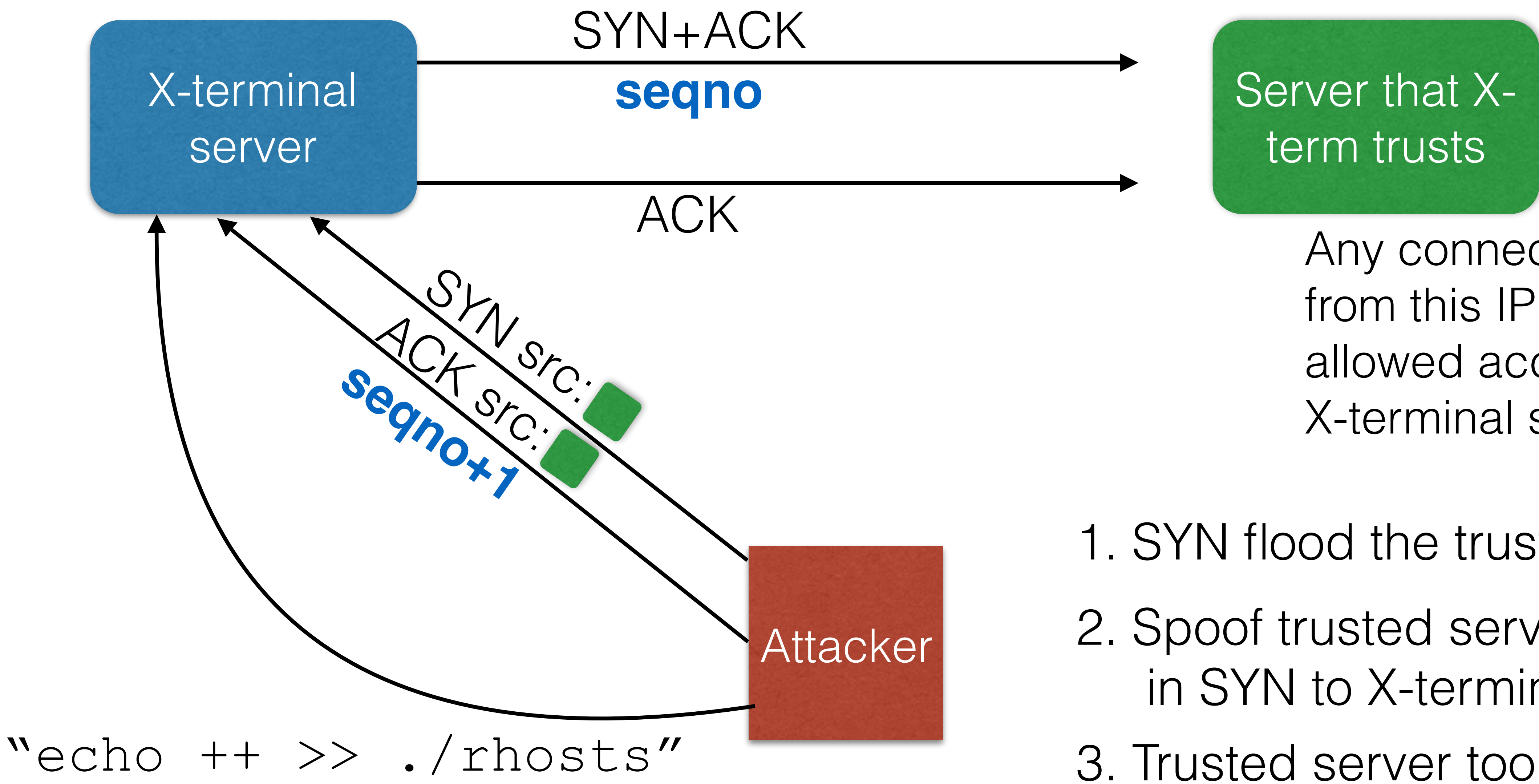
Mitnick attack



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

1. SYN flood the trusted server
2. Spoof trusted server's IP addr in SYN to X-terminal
3. Trusted server too busy to RST
4. ACK with the guessed **seqno**
5. Grant access to all sources
6. RSTs to trusted server (cleanup)

Mitnick attack

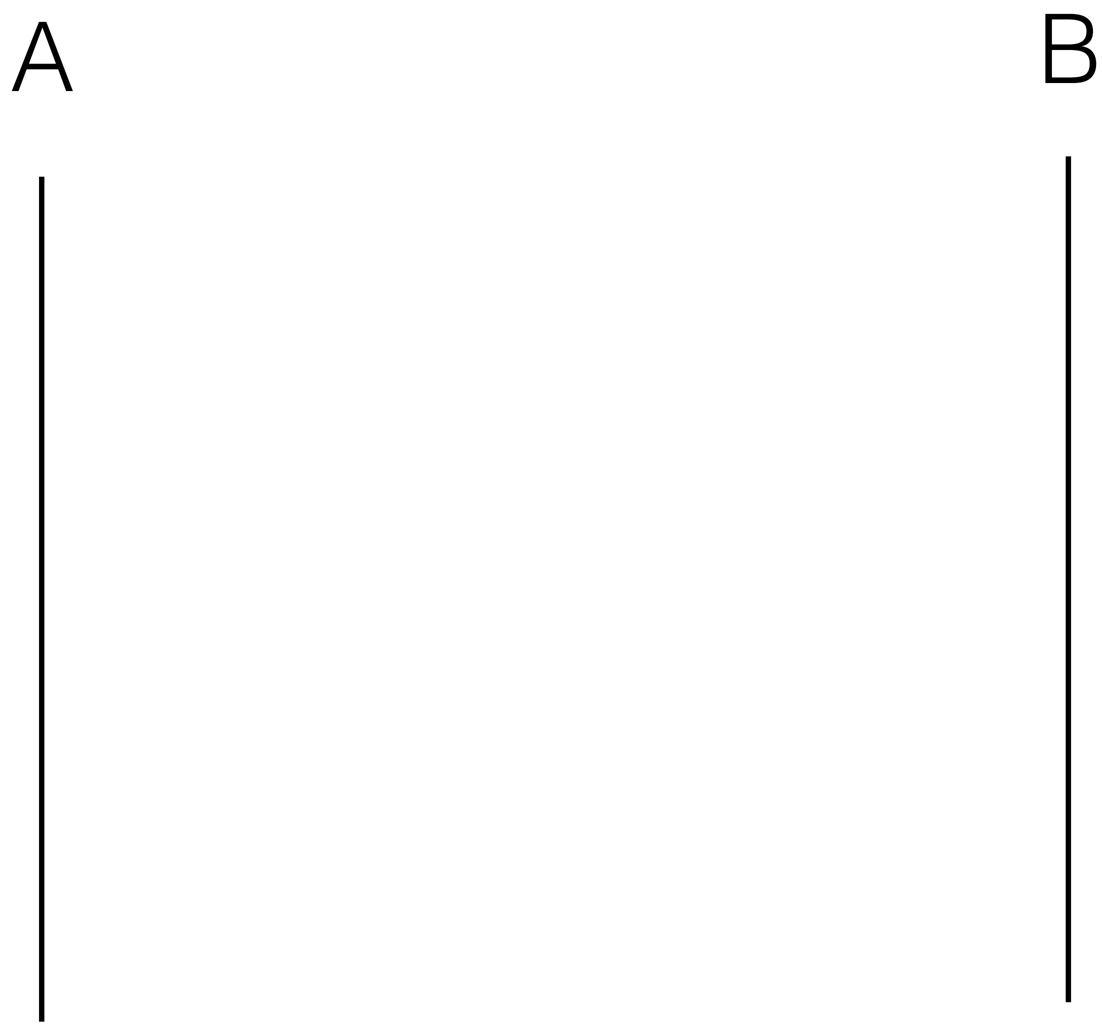


1. SYN flood the trusted server
2. Spoof trusted server's IP addr in SYN to X-terminal
3. Trusted server too busy to RST
4. ACK with the guessed **seqno**
5. Grant access to all sources
6. RSTs to trusted server (cleanup)

Defenses

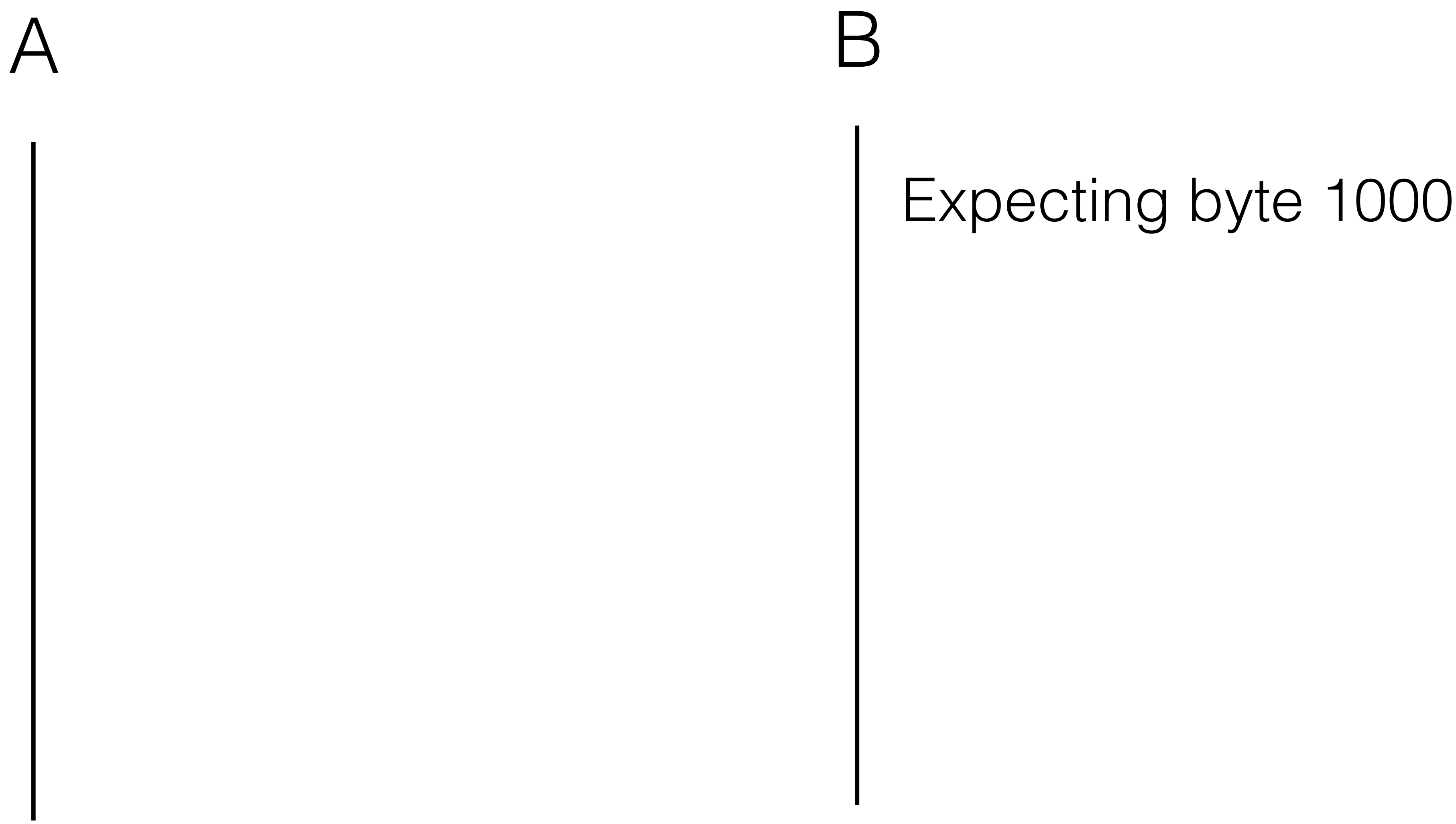
- Initial sequence number must be difficult to predict!

Opt-ack attack



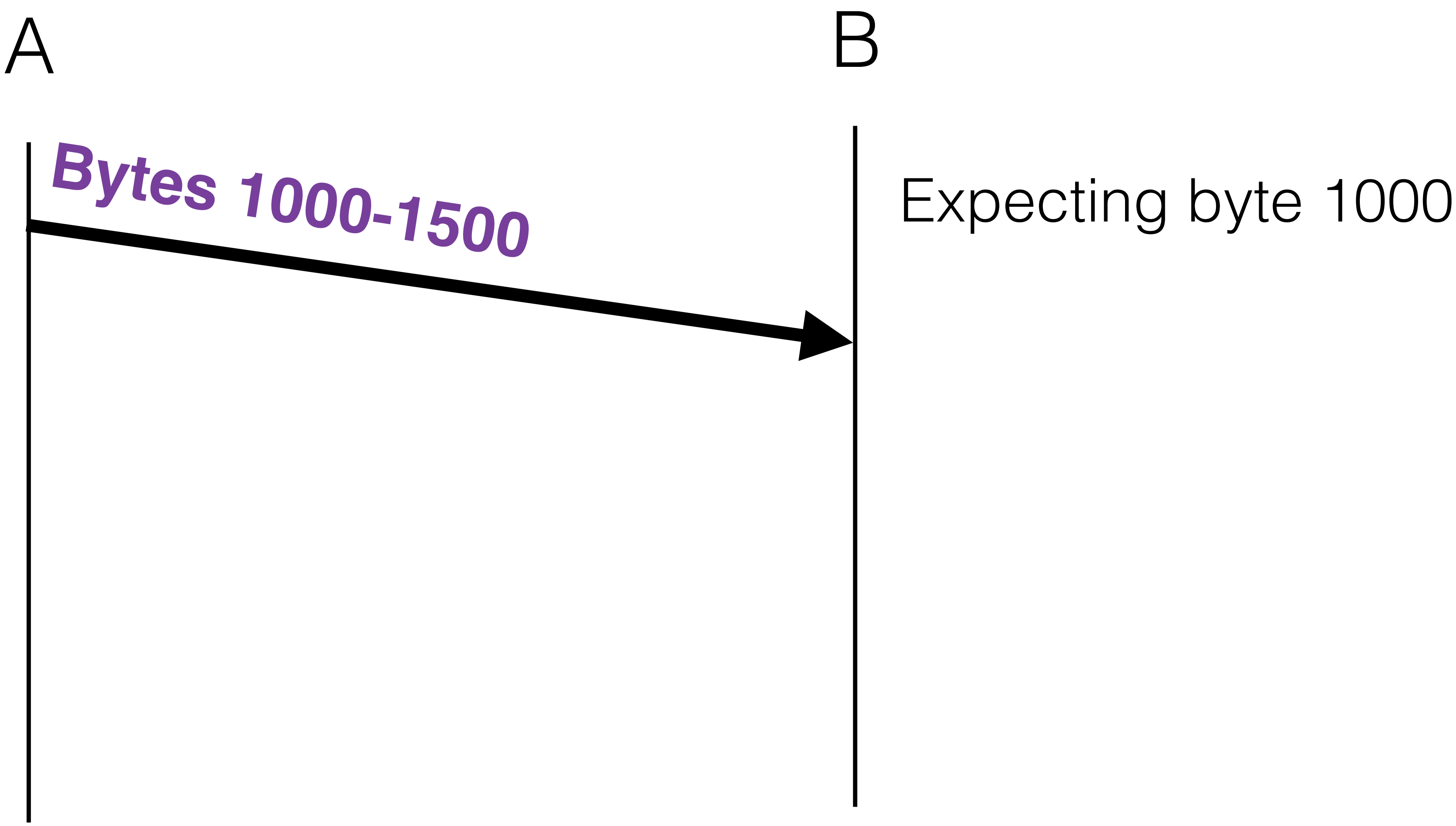
TCP uses ACKs not only for reliability, but also for **congestion control:** the more ACKs come back, the faster I can send

Opt-ack attack



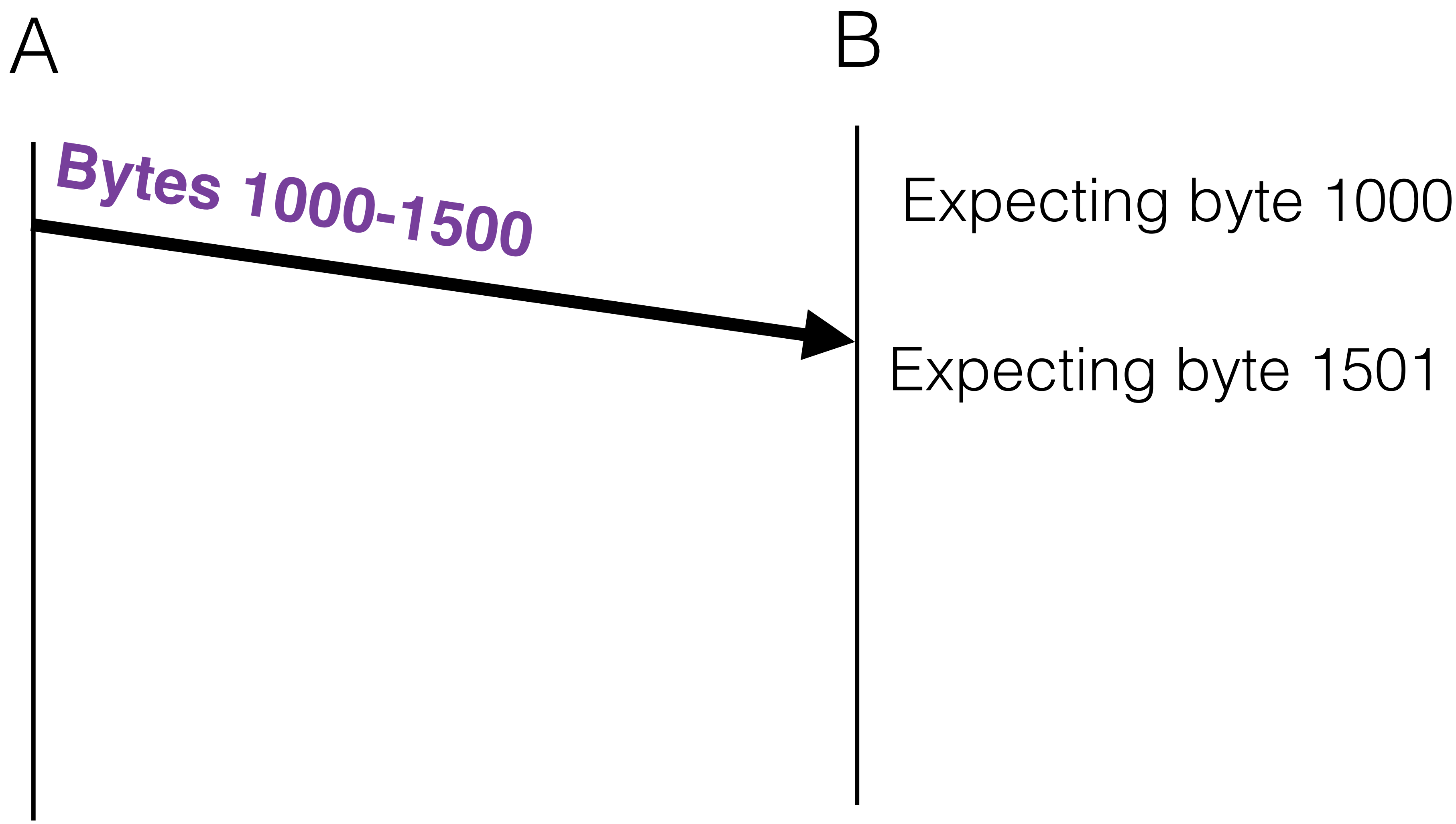
TCP uses ACKs not only for reliability, but also for
congestion control:
the more ACKs come back, the faster I can send

Opt-ack attack



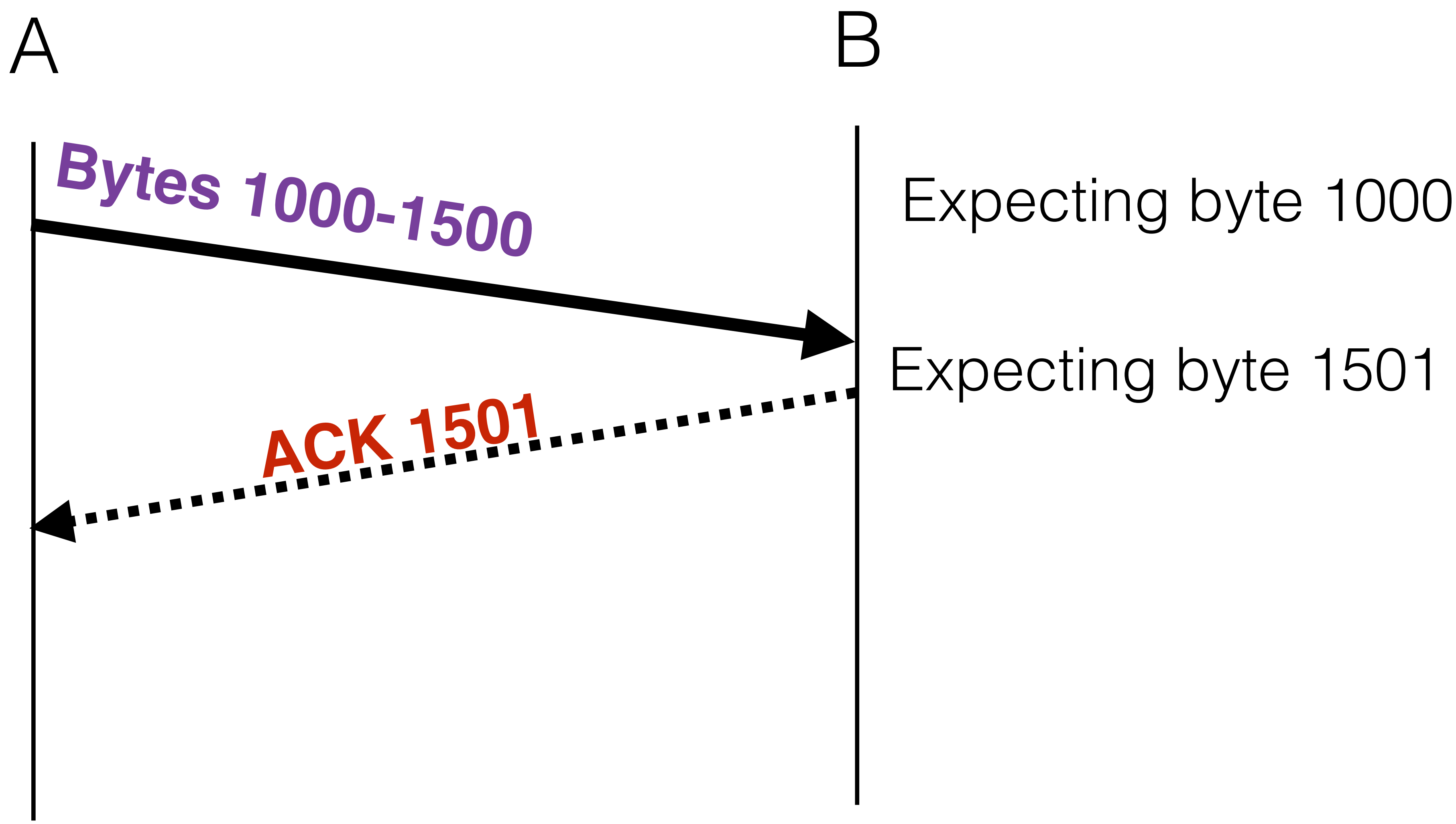
TCP uses ACKs not only for reliability, but also for **congestion control:** the more ACKs come back, the faster I can send

Opt-ack attack



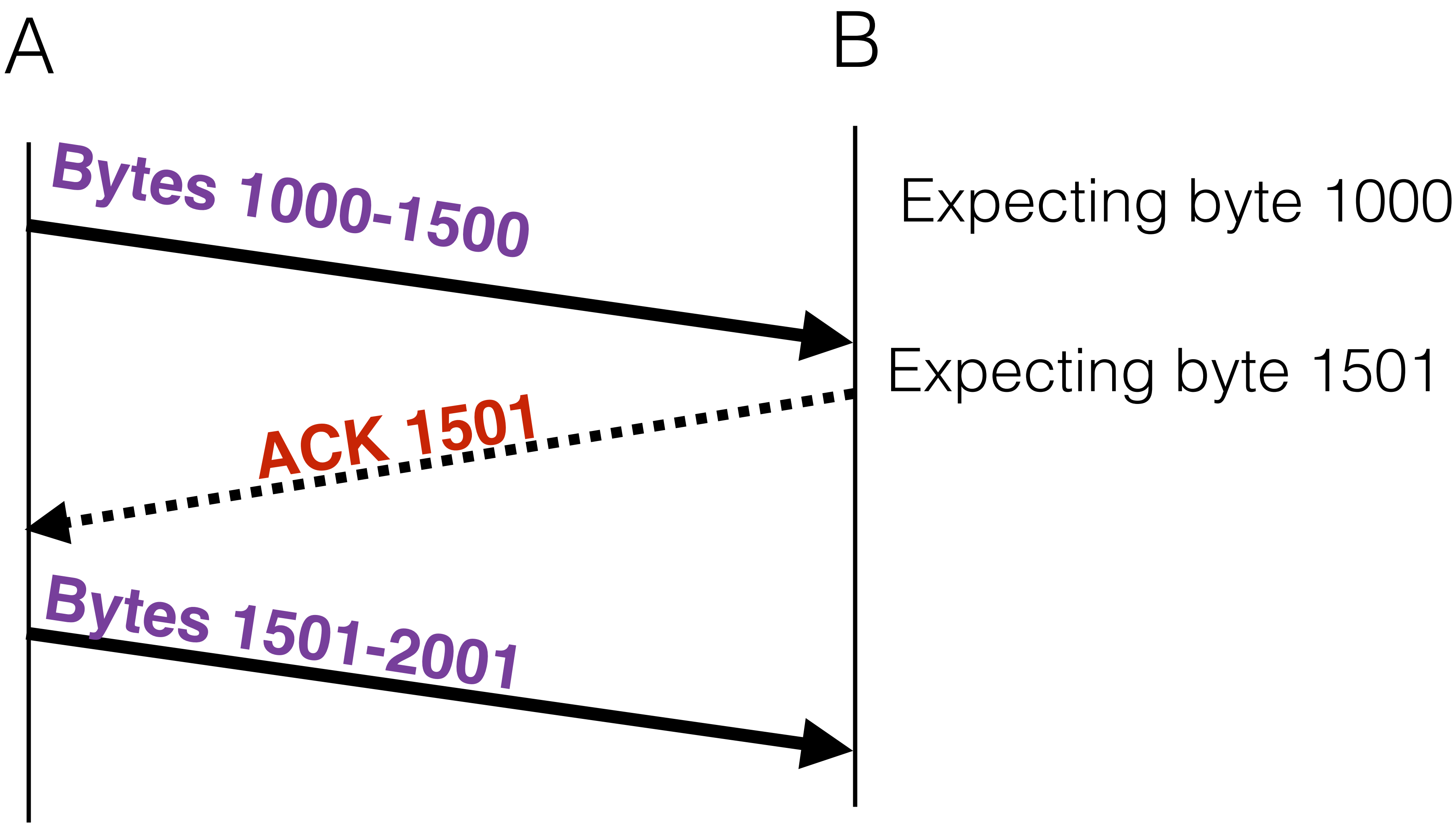
TCP uses ACKs not only for reliability, but also for **congestion control:** the more ACKs come back, the faster I can send

Opt-ack attack



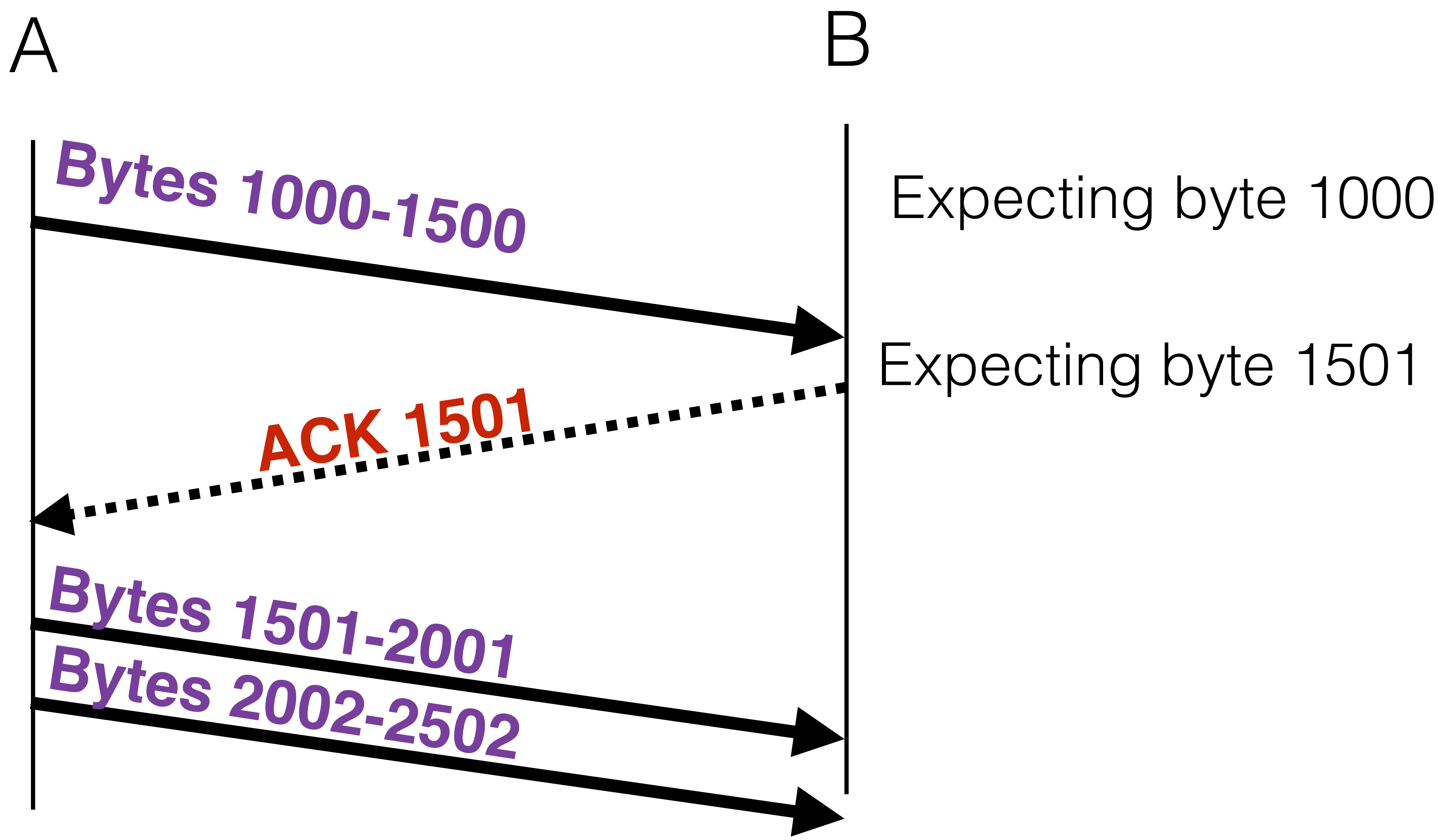
TCP uses ACKs not only for reliability, but also for **congestion control:**
the more ACKs come back, the faster I can send

Opt-ack attack



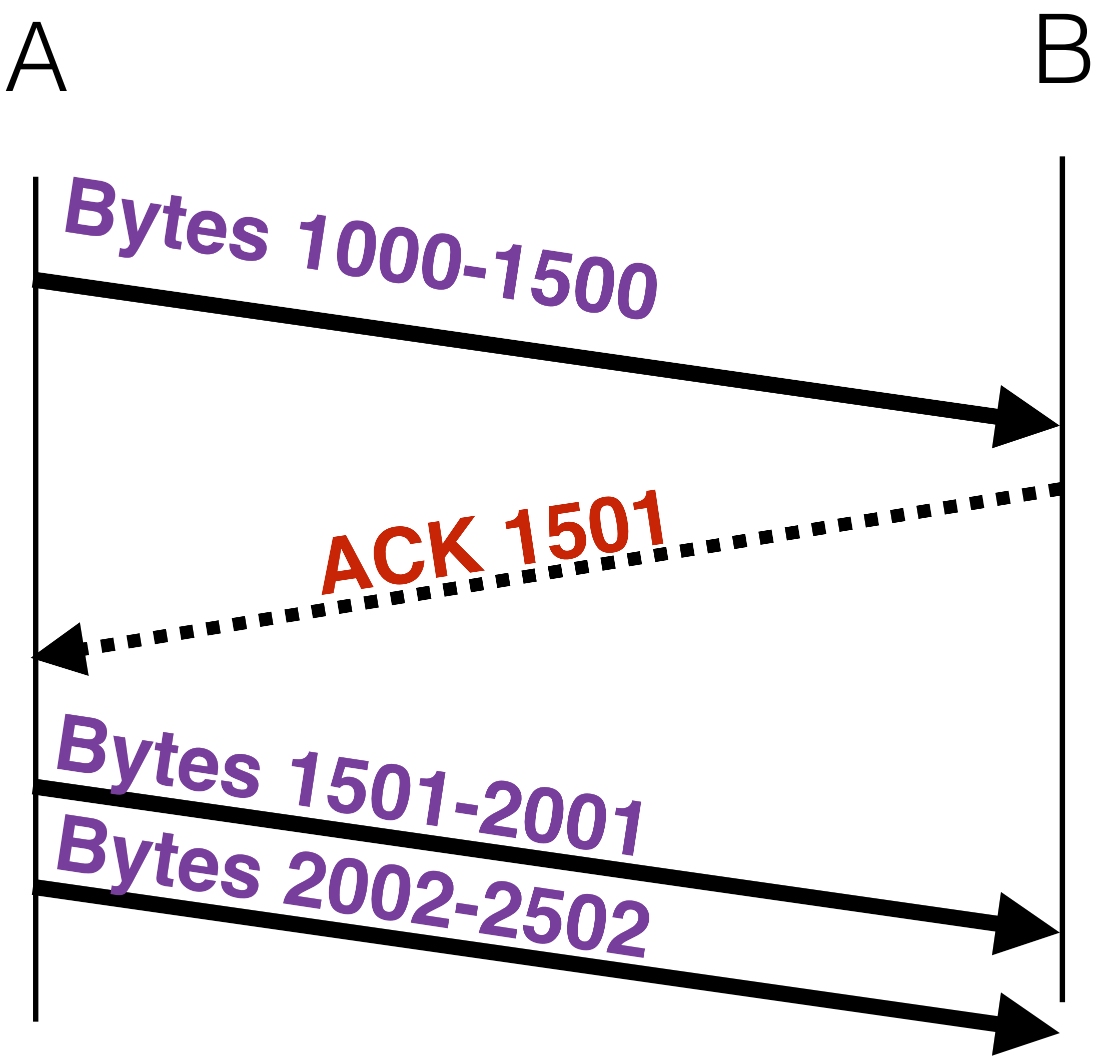
TCP uses ACKs not only for reliability, but also for **congestion control:** the more ACKs come back, the faster I can send

Opt-ack attack

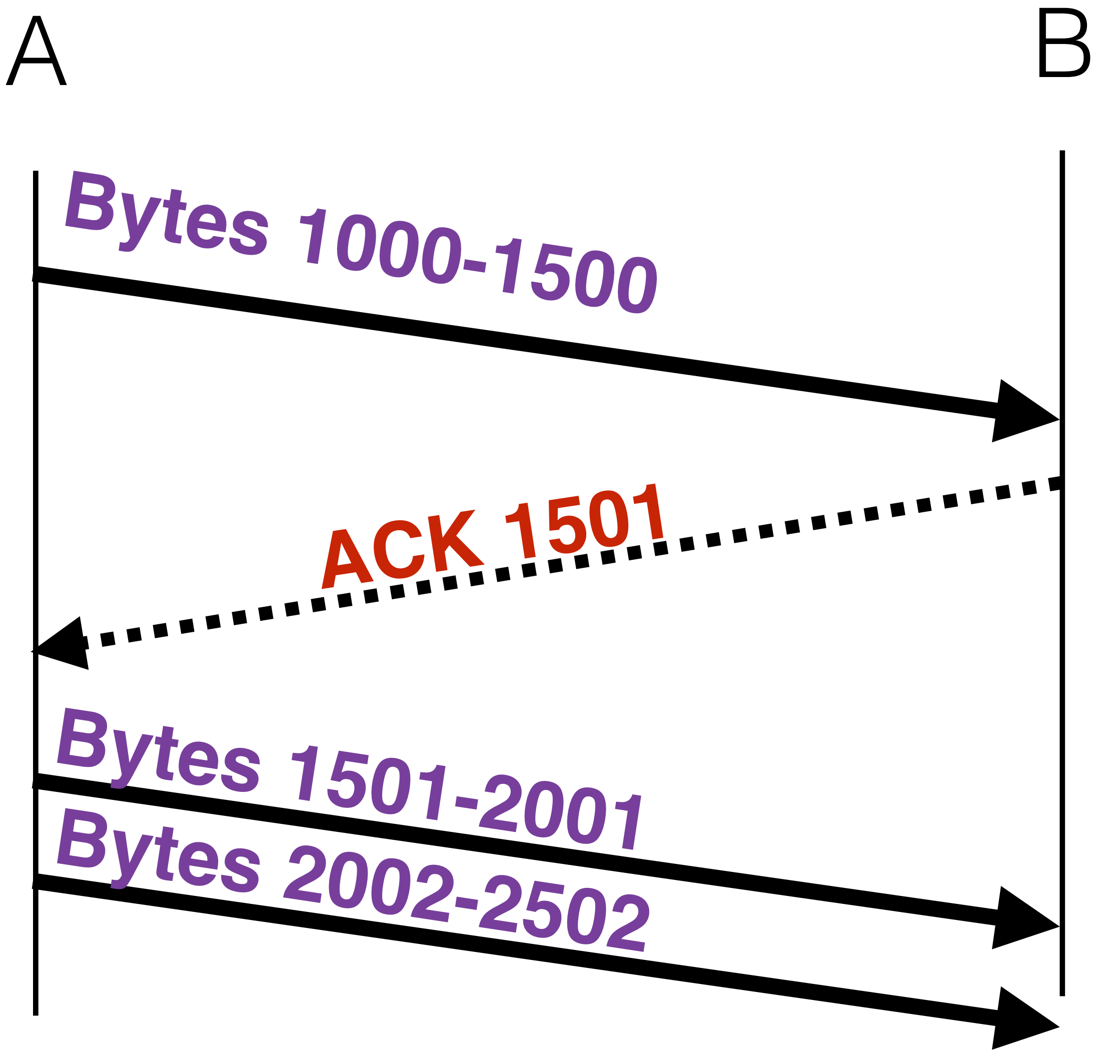


TCP uses ACKs not only for reliability, but also for **congestion control:**
the more ACKs come back, the faster I can send

Opt-ack attack

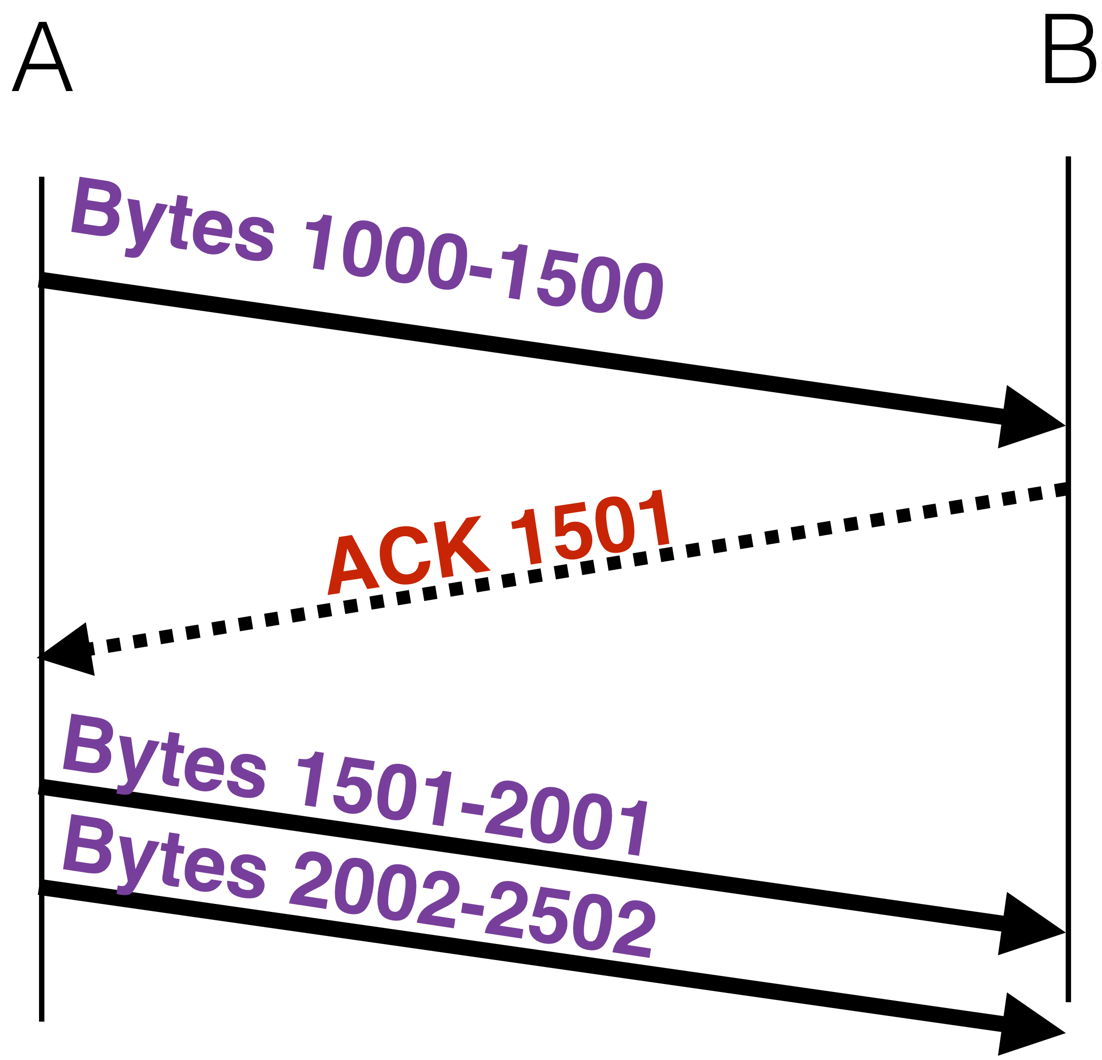


Opt-ack attack



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

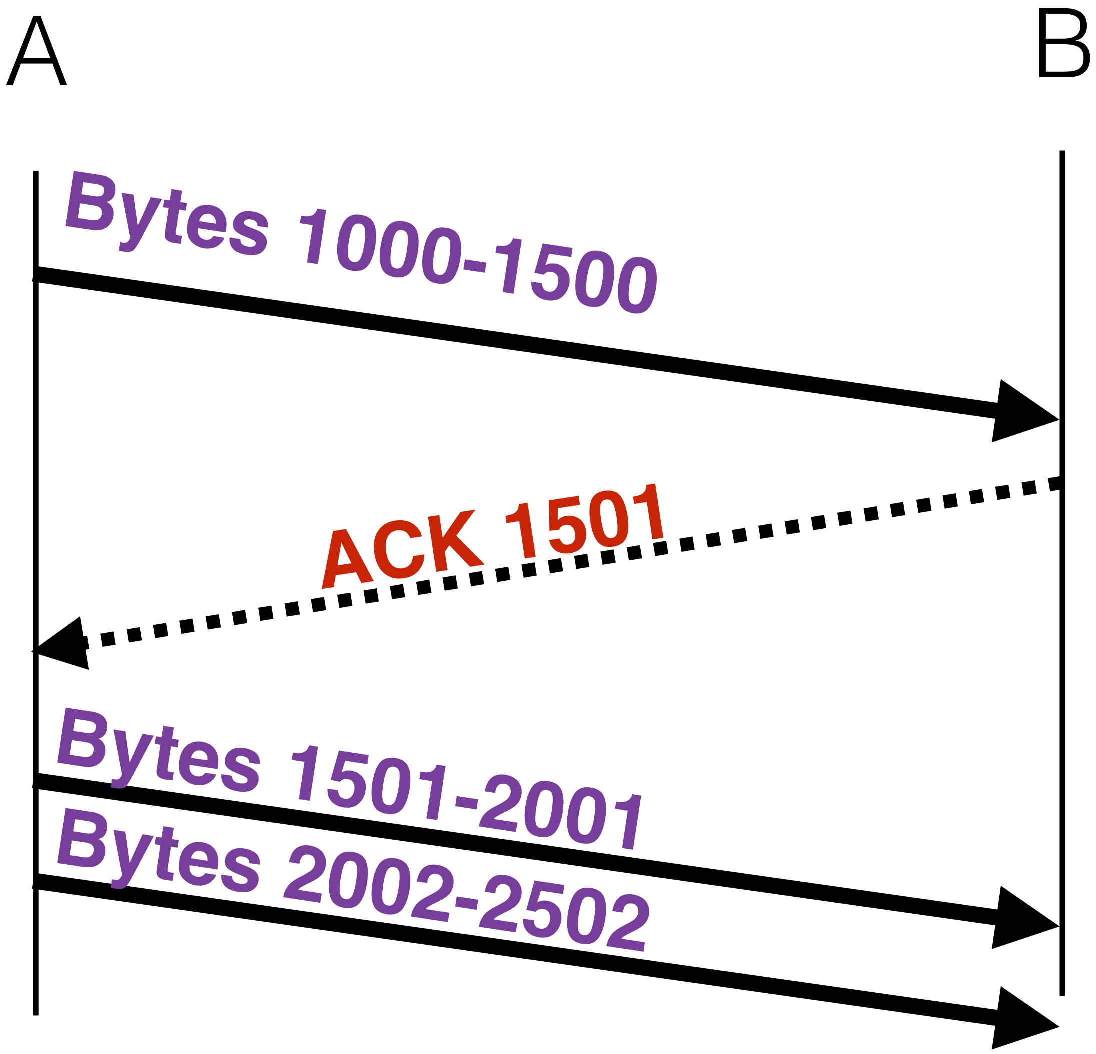
Opt-ack attack



But to get you to send faster, I need to get data in order to ACK, so I need to receive quickly

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

Opt-ack attack

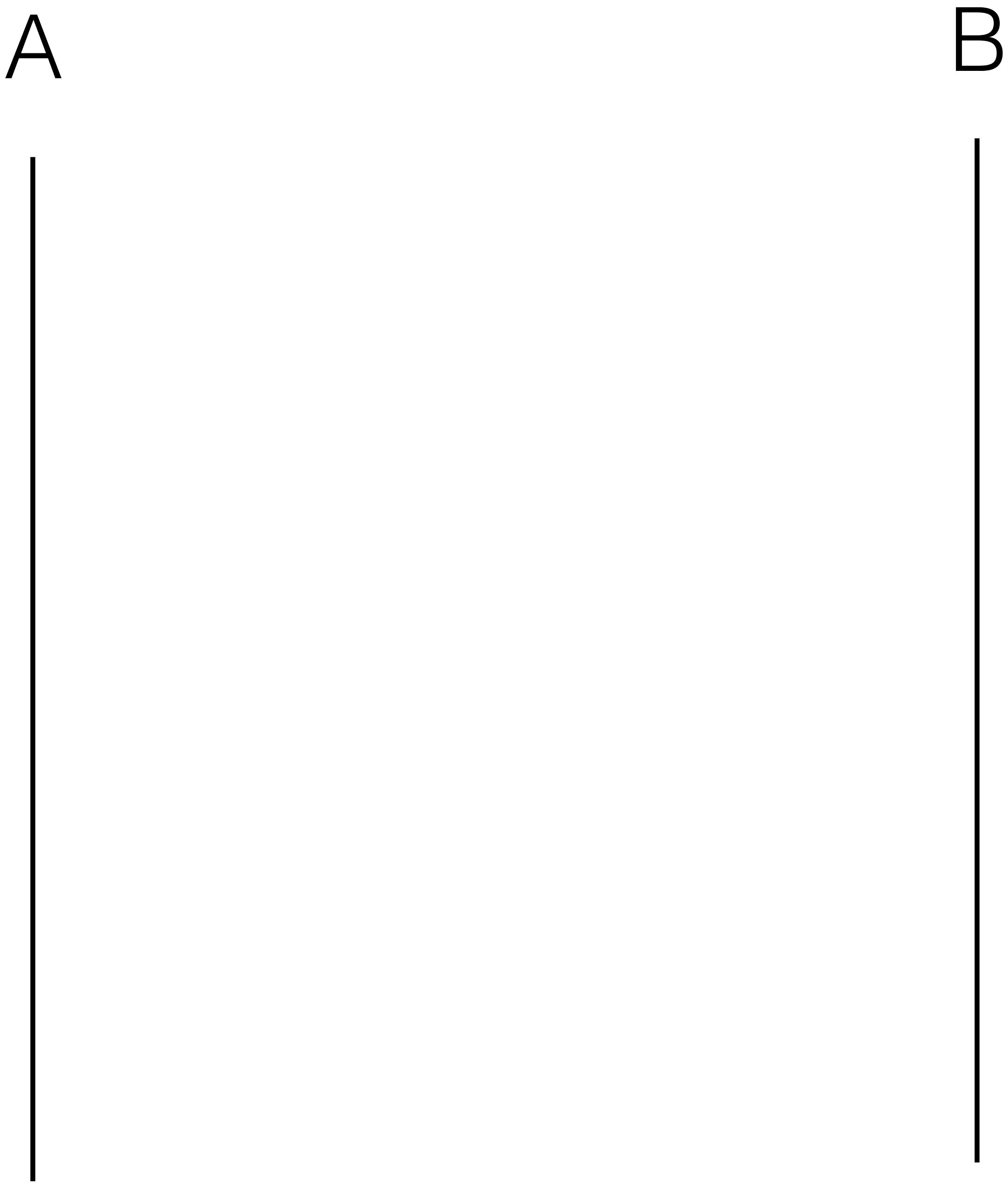


But to get you to send faster, I need to get data in order to ACK, so I need to receive quickly

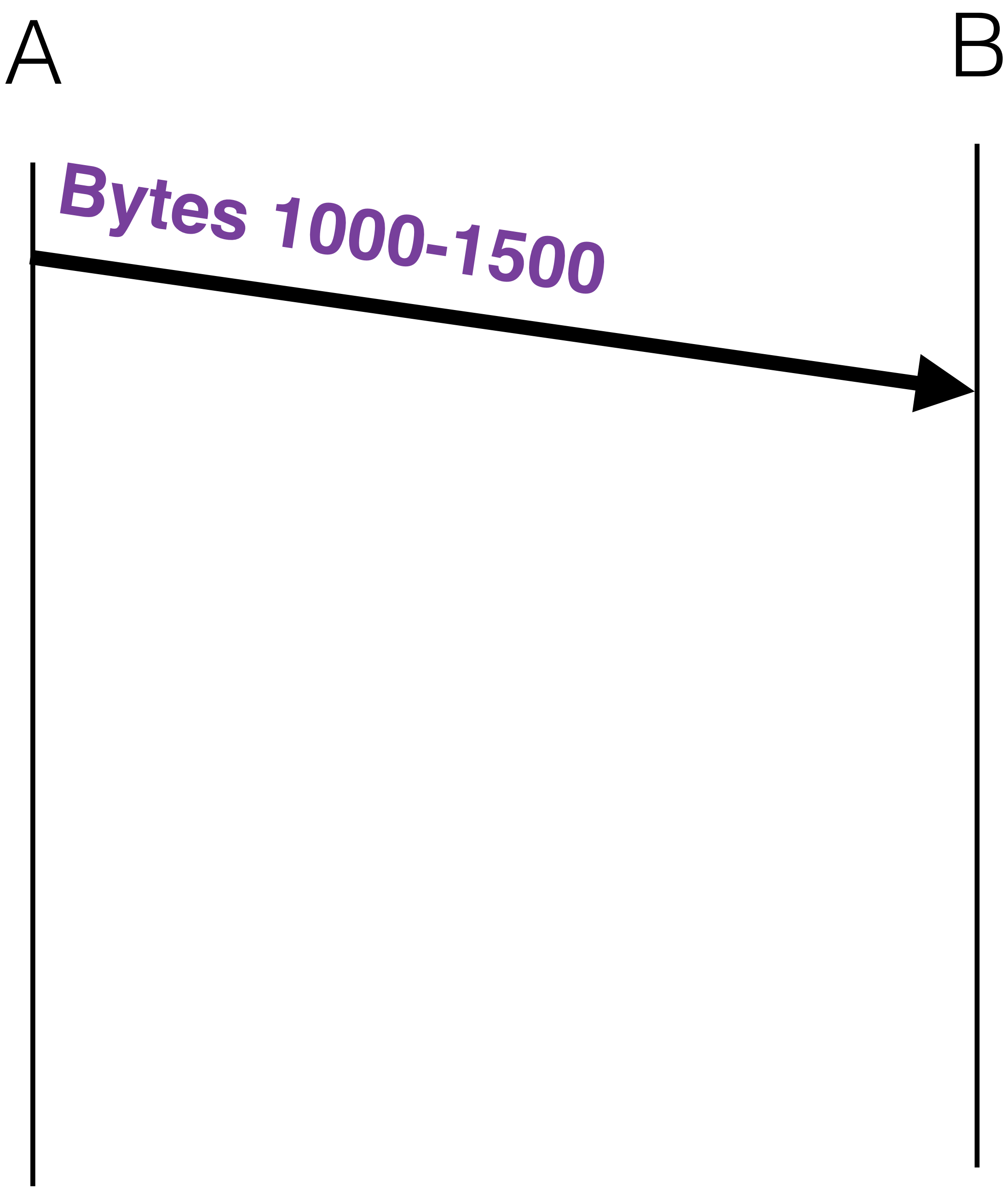
...or do I?

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

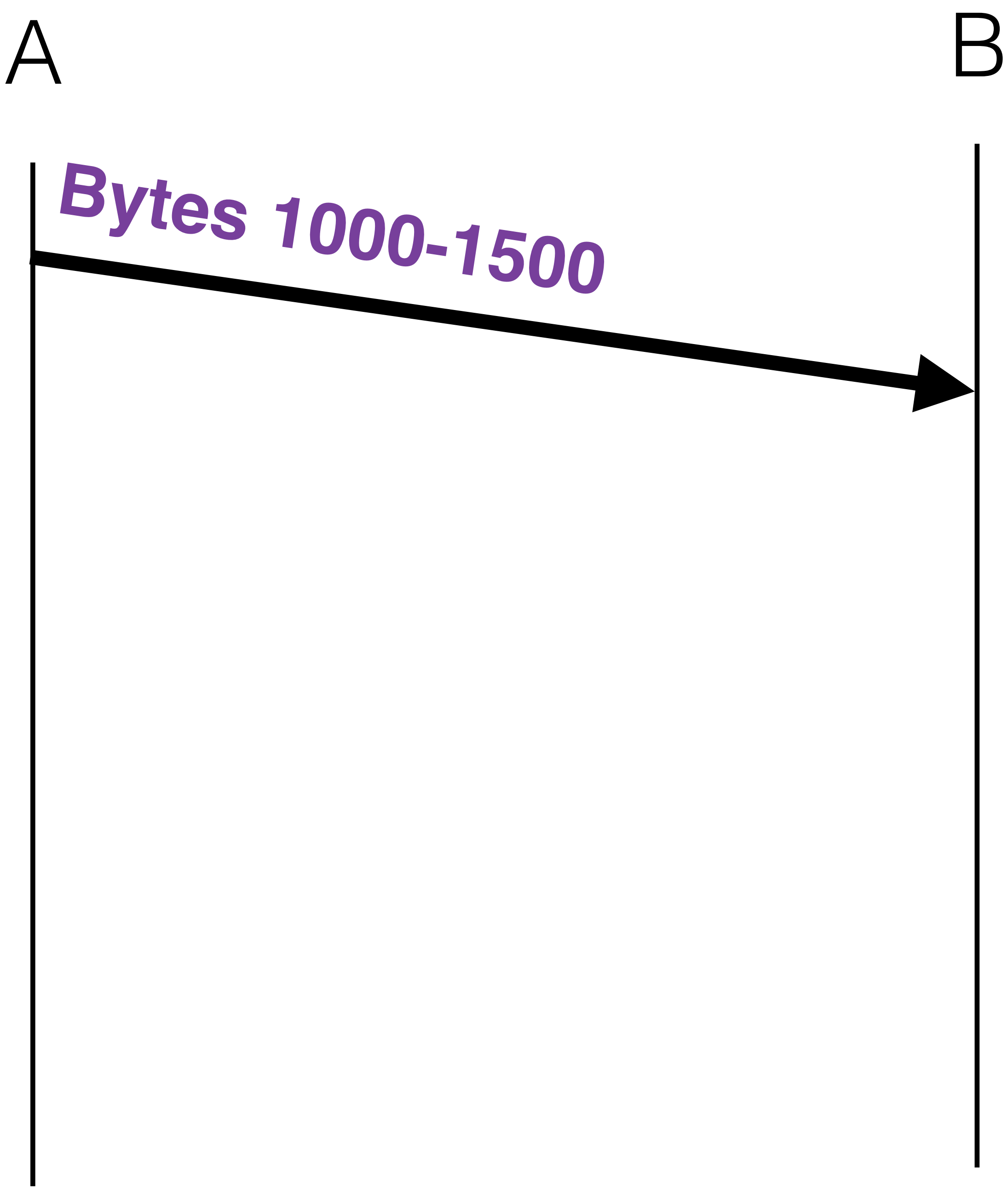
Opt-ack attack



Opt-ack attack

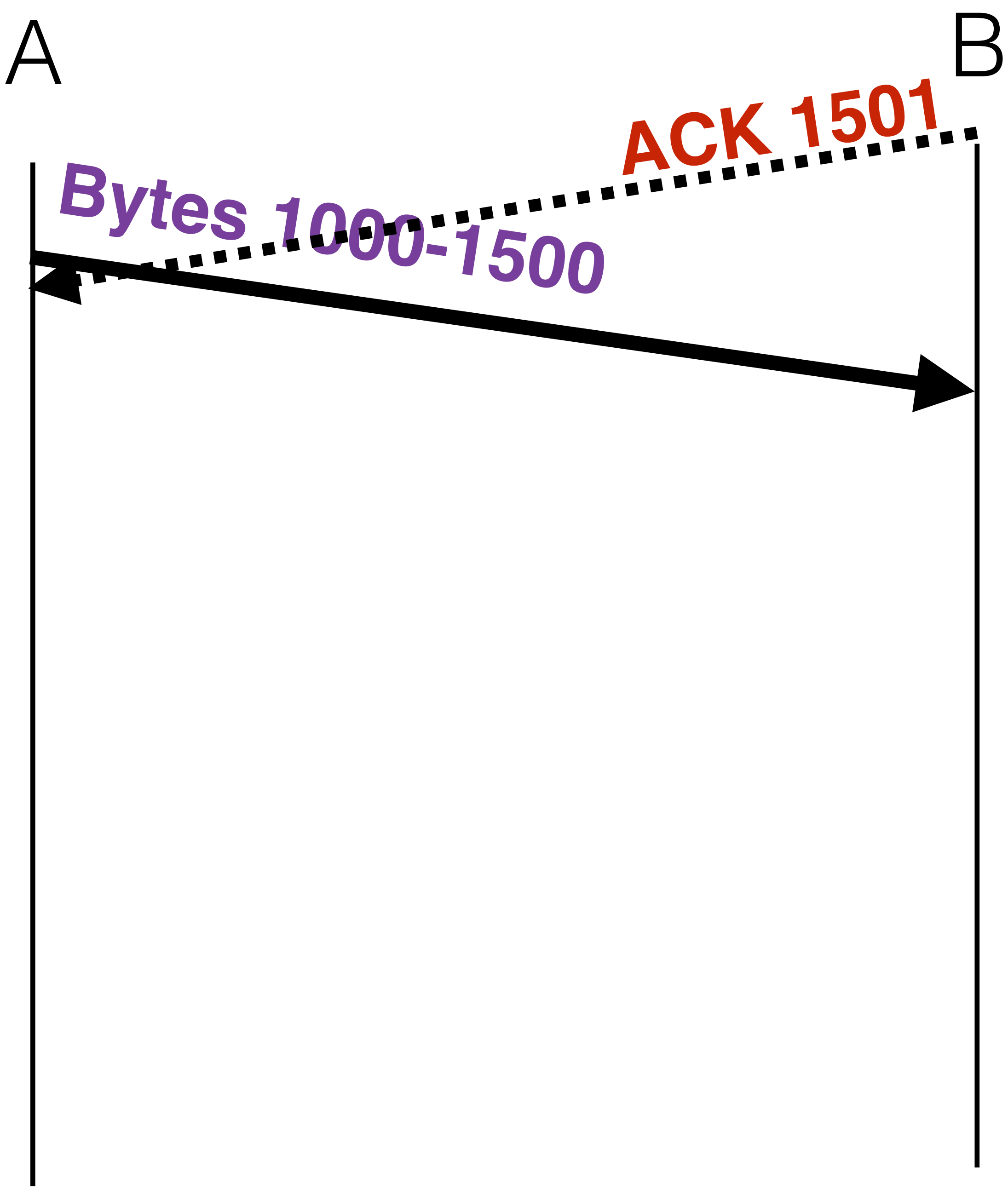


Opt-ack attack



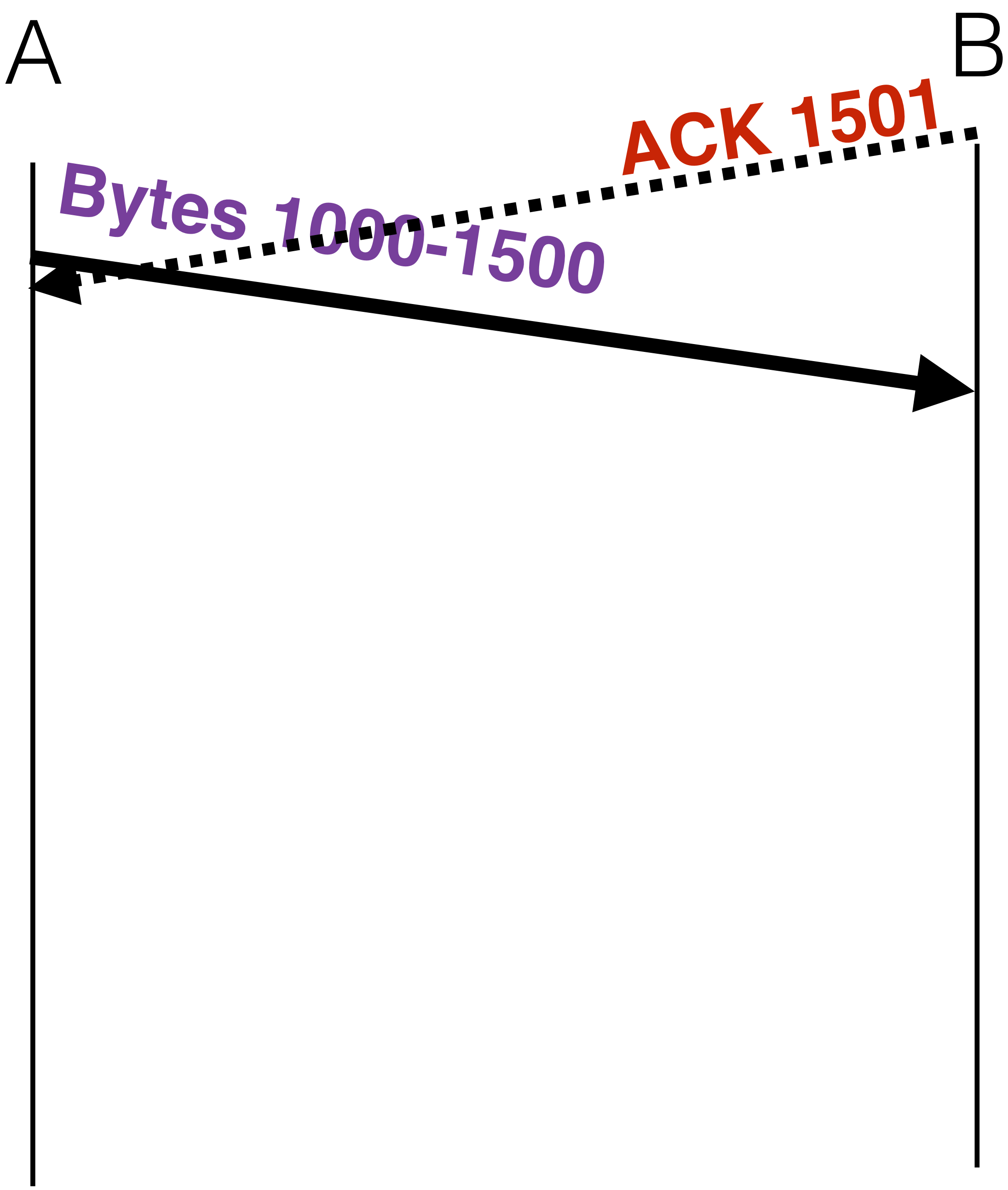
If I can predict what the last seqno will be *and* when A will send it

Opt-ack attack



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

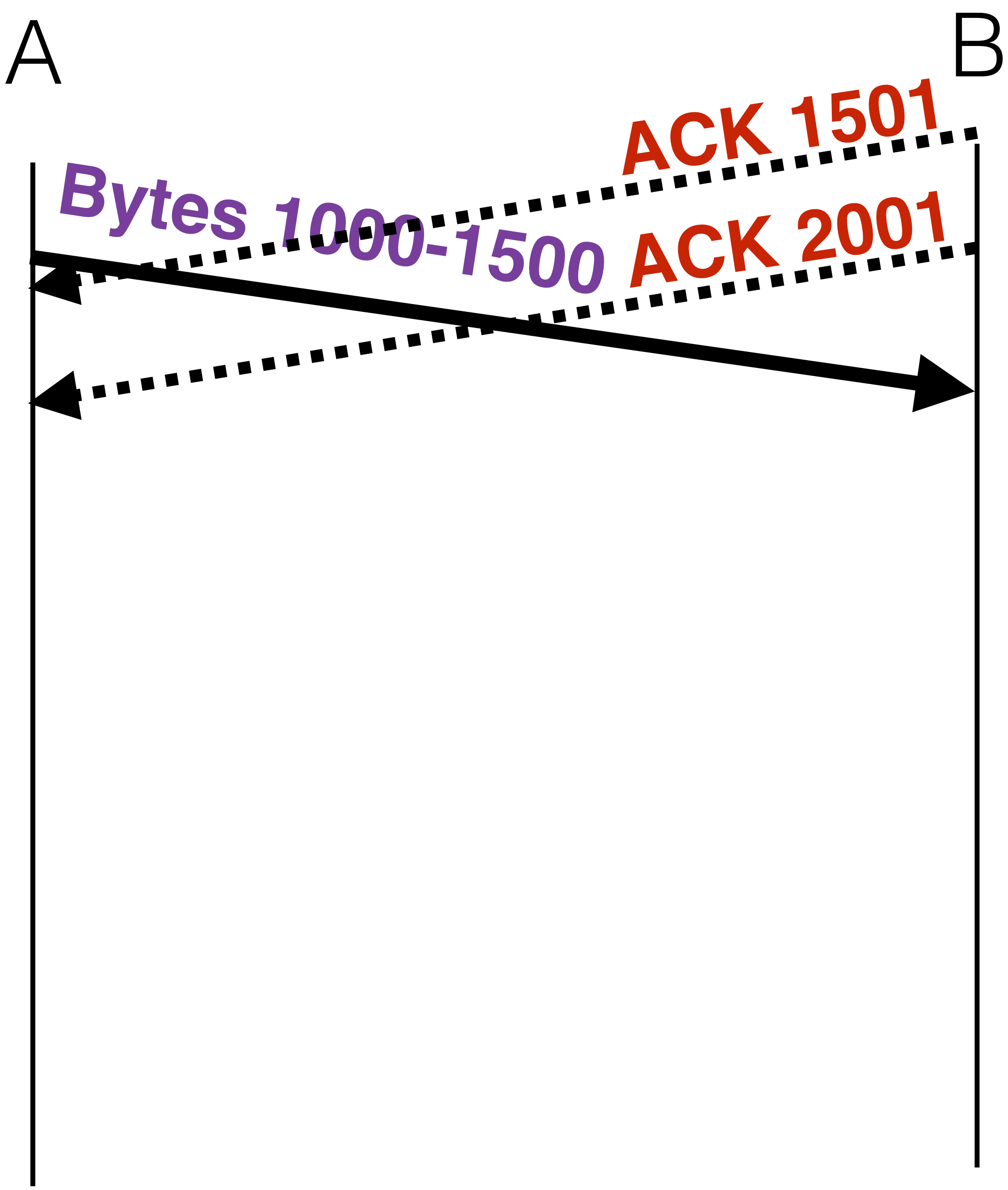
Opt-ack attack



Then I could ACK early! (“optimistically”)

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

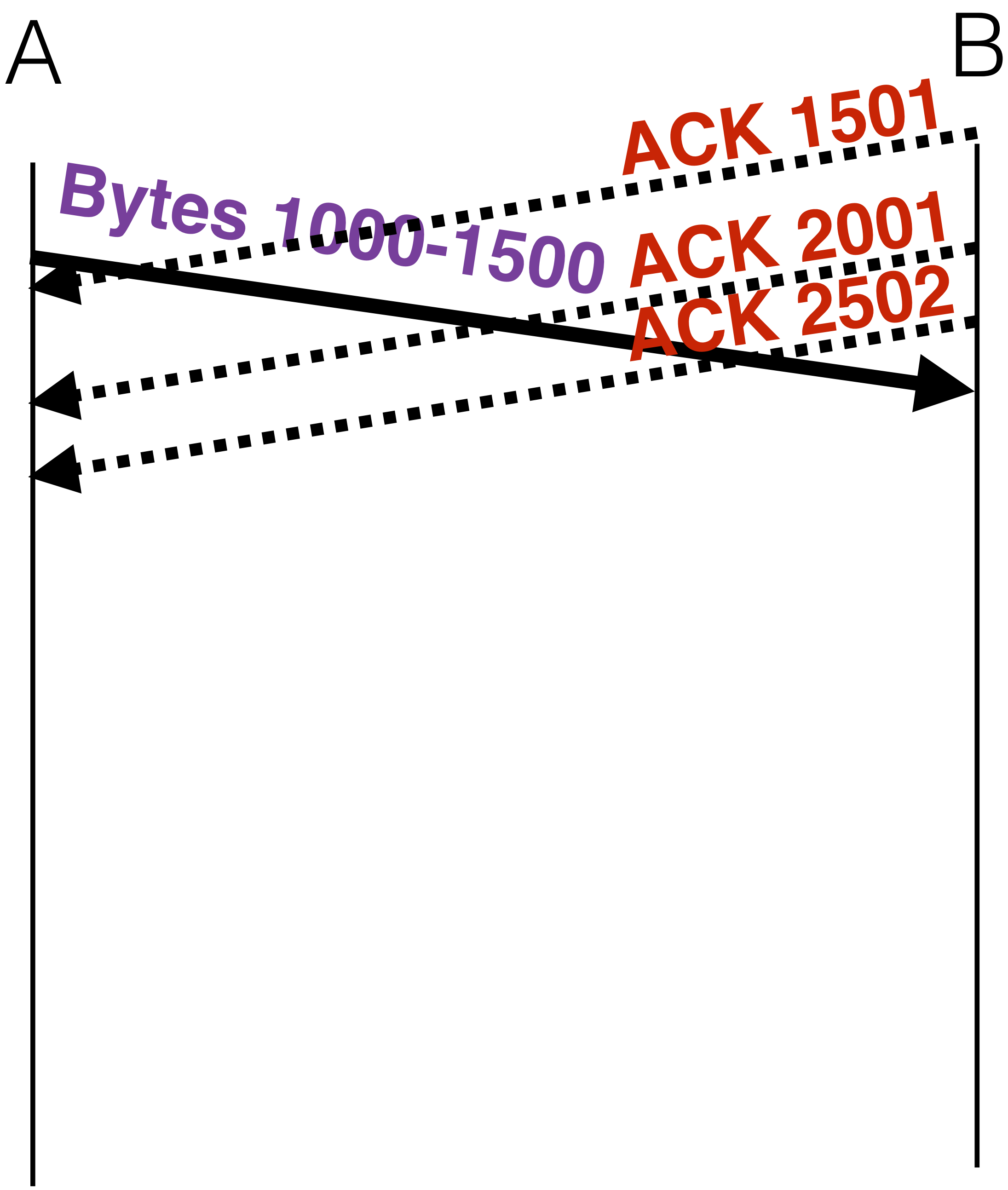
Opt-ack attack



Then I could ACK early! (“optimistically”)

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

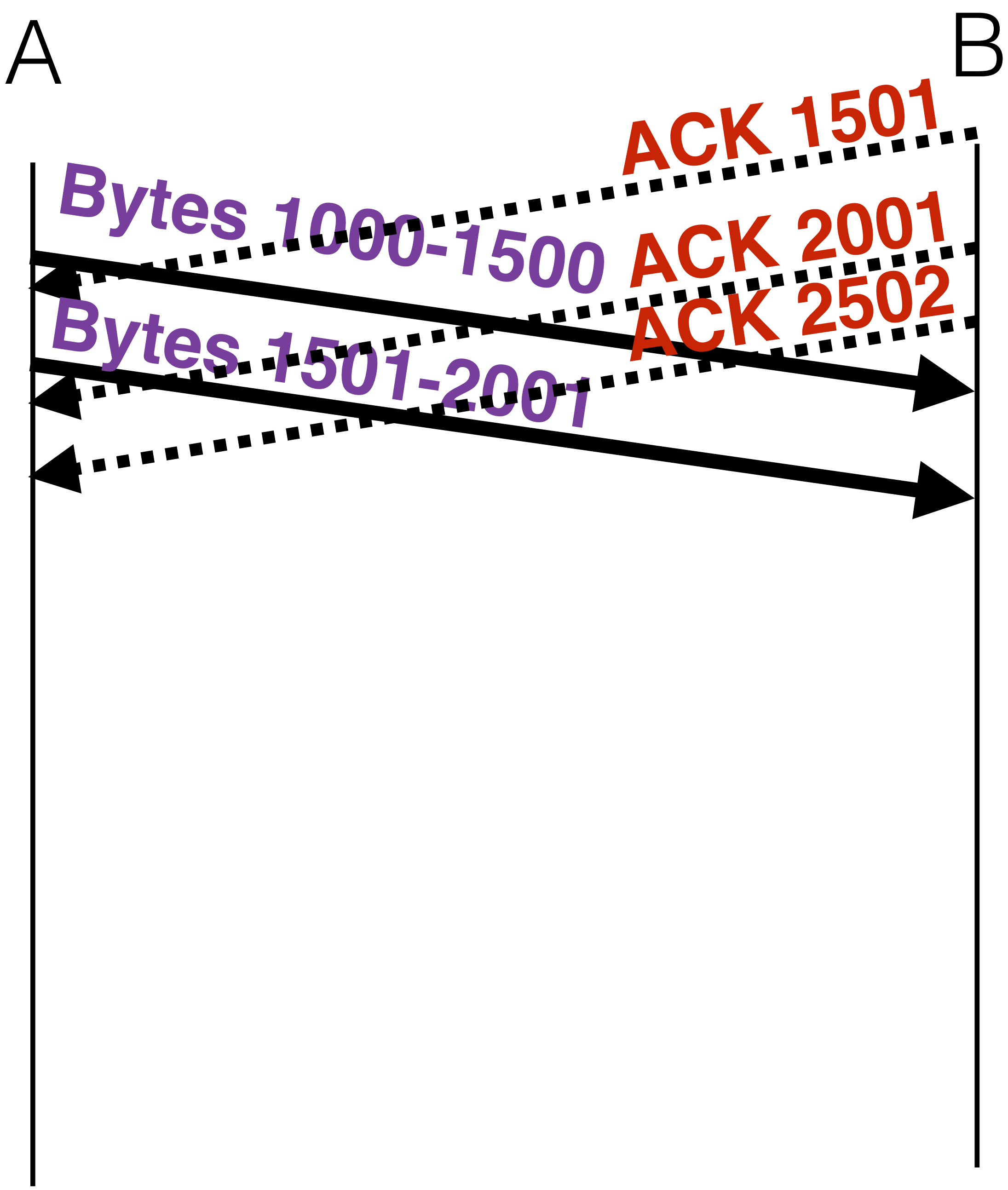
Opt-ack attack



Then I could ACK early! (“optimistically”)

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

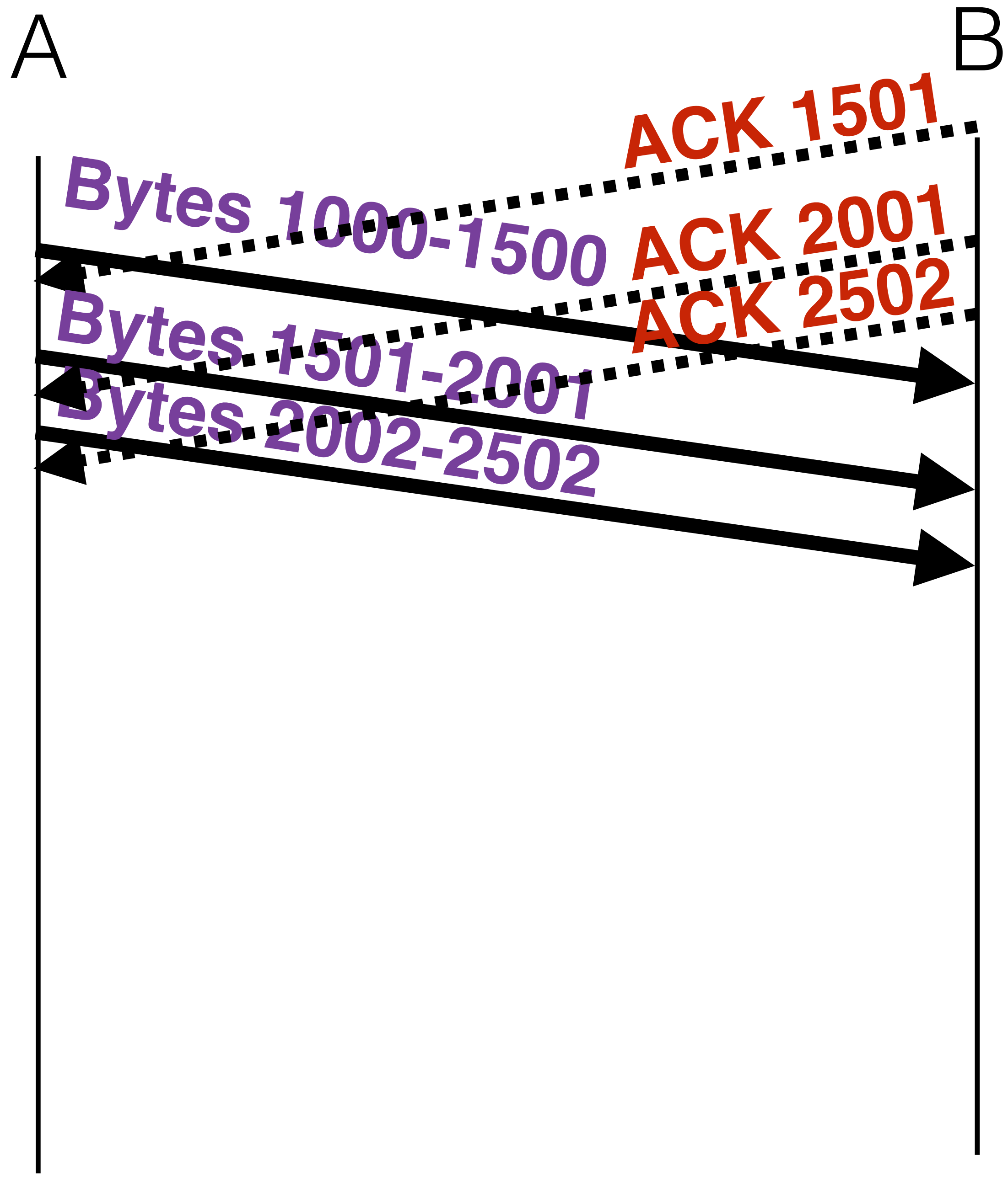
Opt-ack attack



Then I could ACK early! ("optimistically")

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

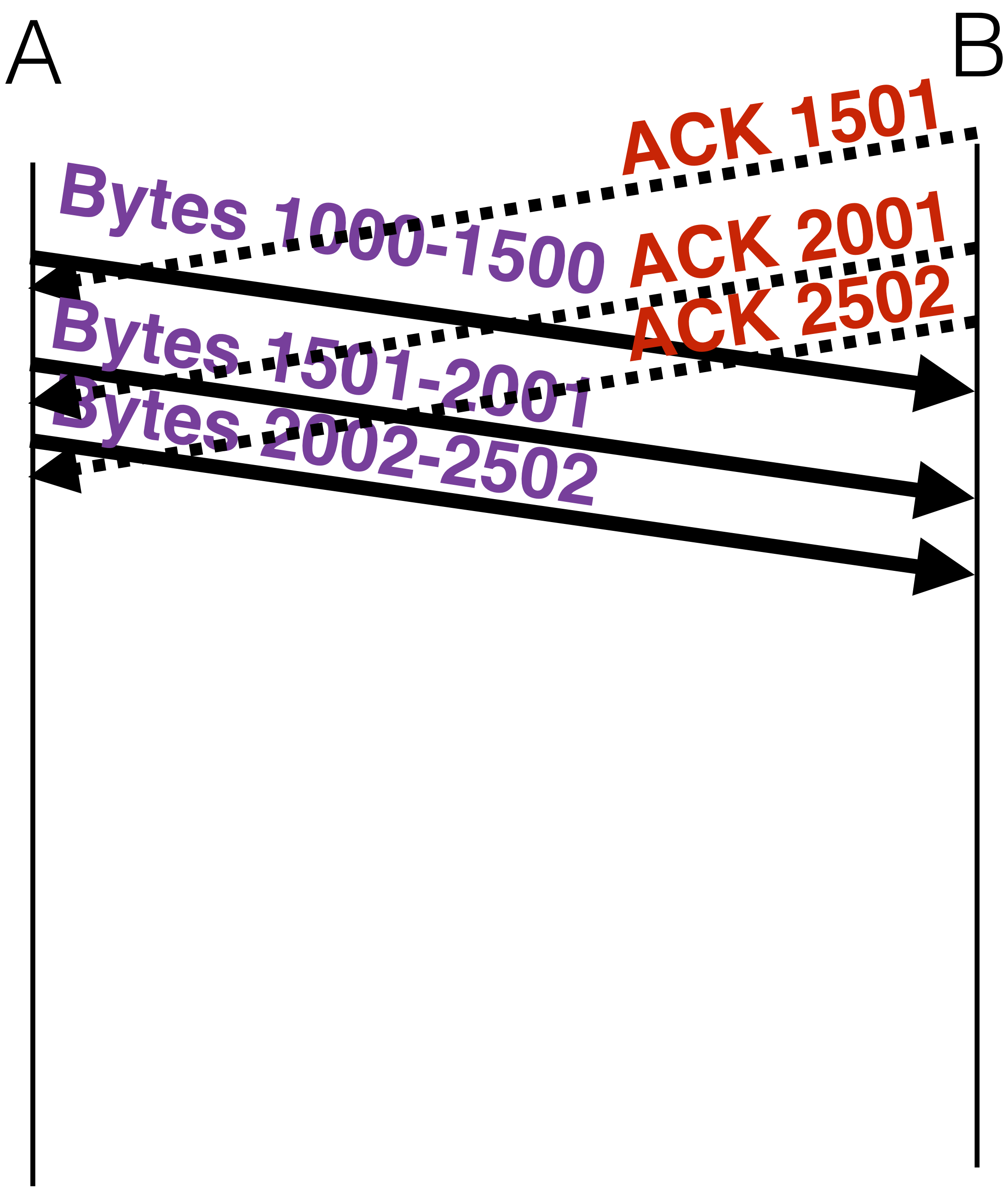
Opt-ack attack



Then I could ACK early! (“optimistically”)

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

Opt-ack attack

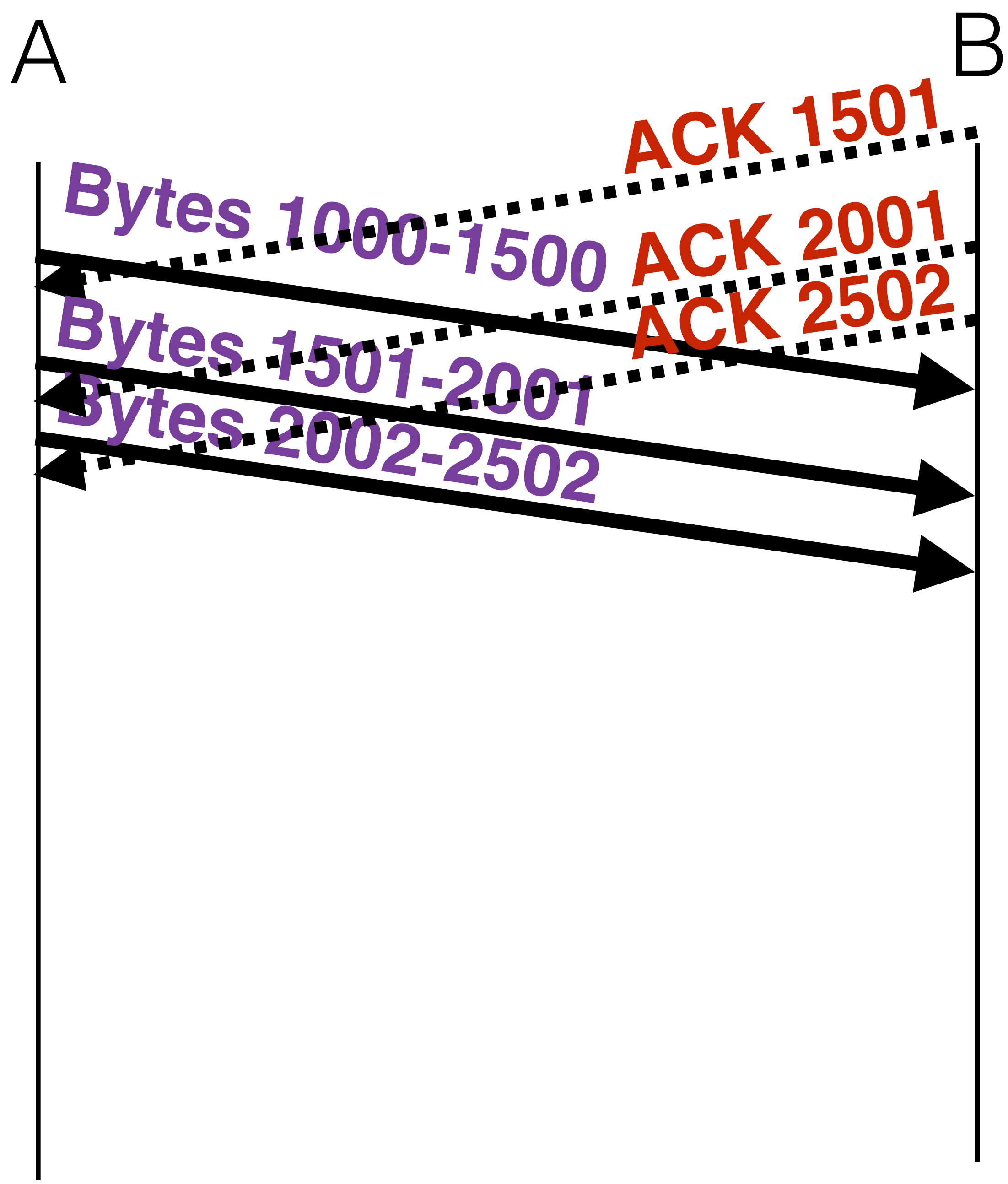


Then I could ACK early! (“optimistically”)

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

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

Opt-ack attack



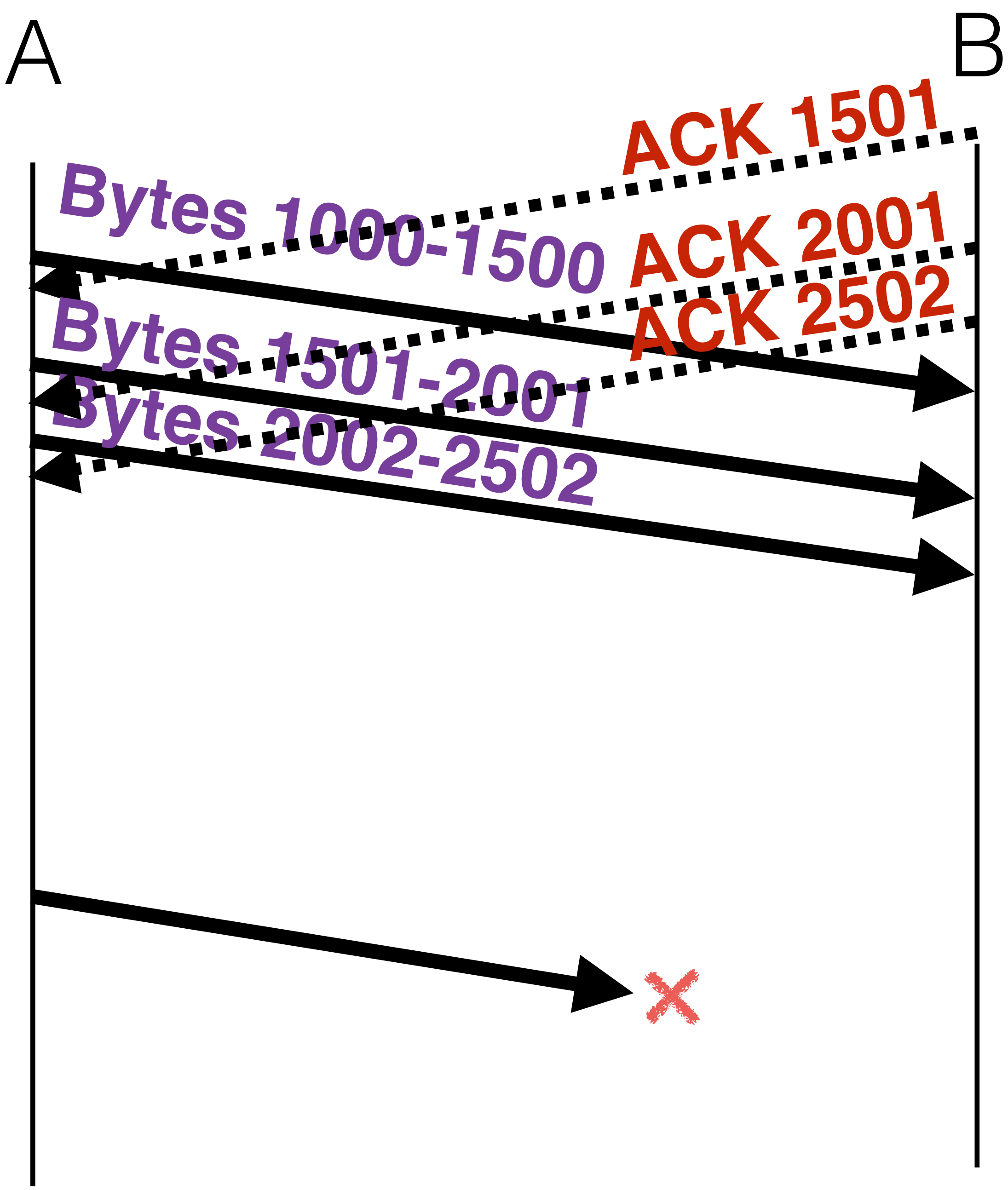
Then I could ACK early! (“optimistically”)

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



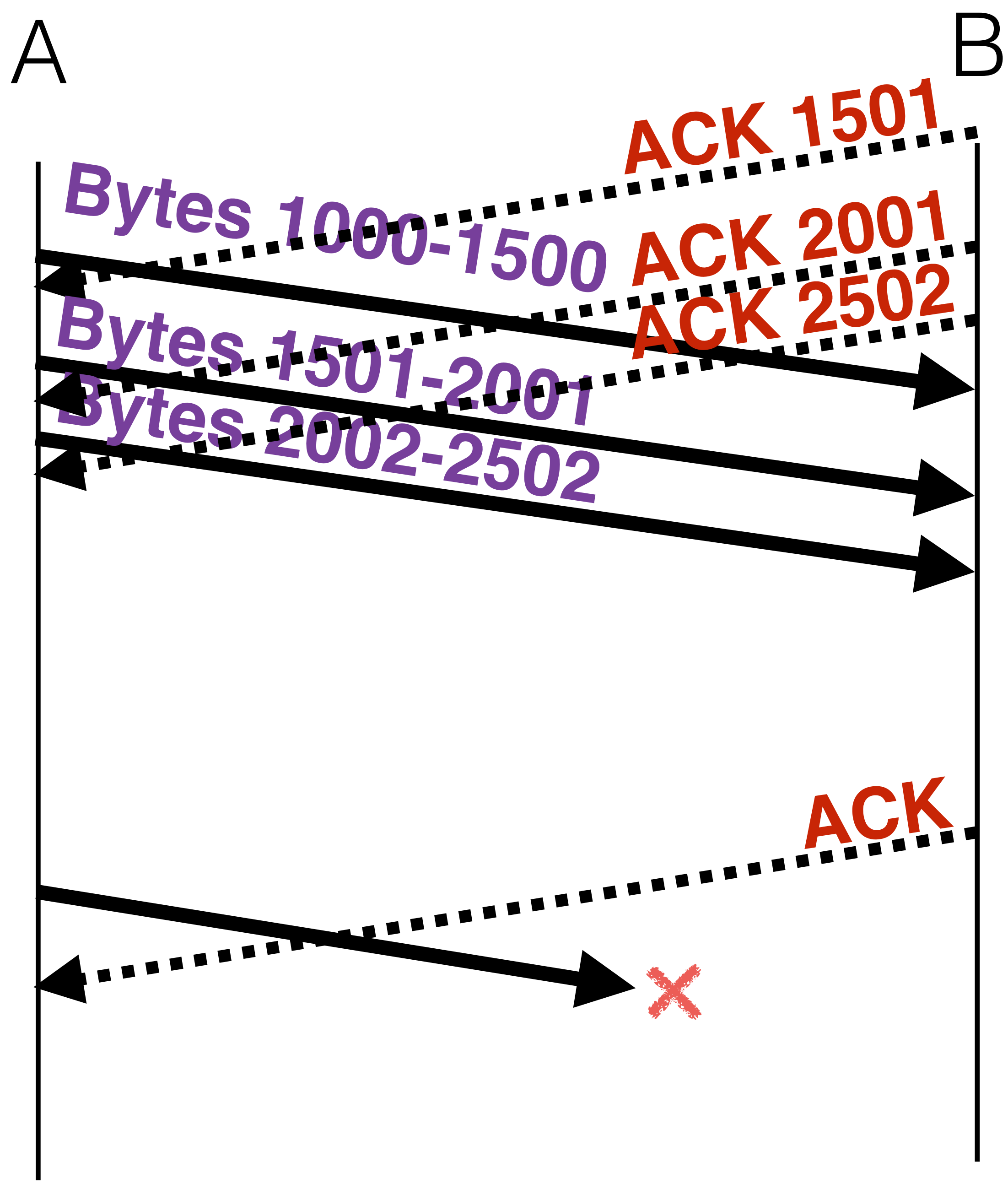
Then I could ACK early! (“optimistically”)

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



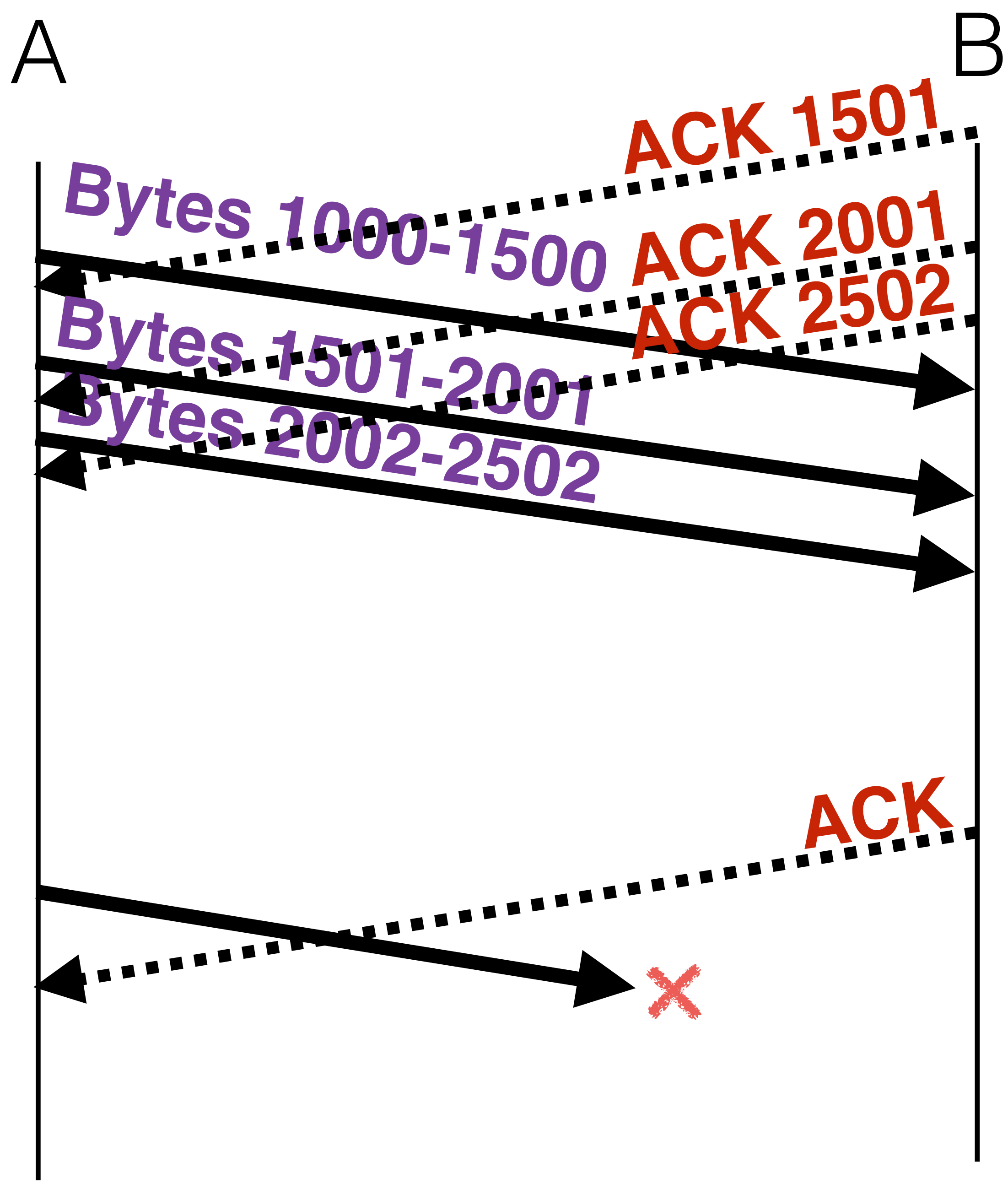
Then I could ACK early! (“optimistically”)

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.

Opt-ack attack



Then I could ACK early! (“optimistically”)

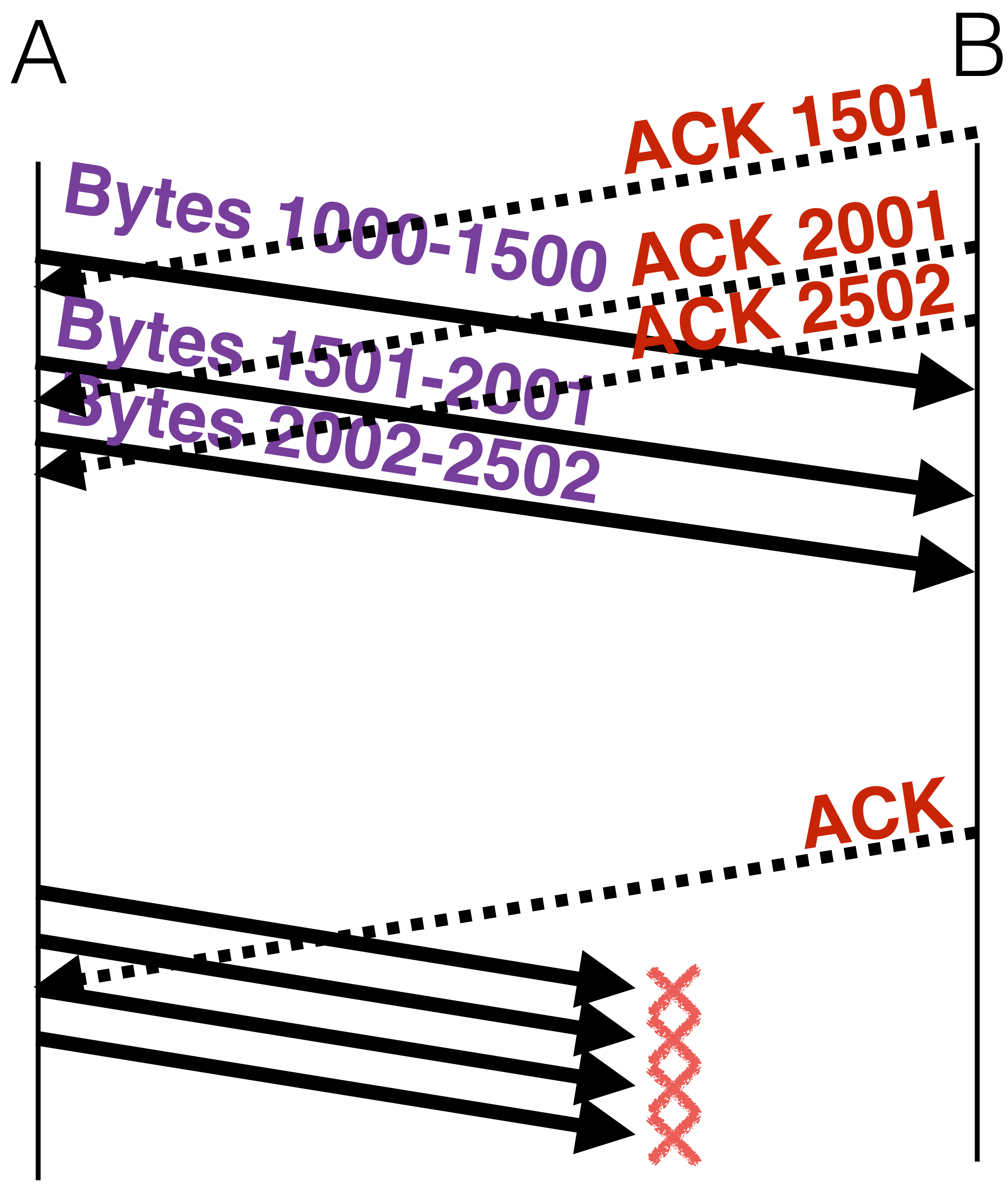
If I can predict what the last seqno will be *and* when A will send it

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:

$$\overbrace{\boxed{\frac{\text{Max window size}}{\text{MSS}}}}^{\text{Packets sent per ACK}} \times \overbrace{\boxed{(\underbrace{14}_{\text{Ethernet}} + \underbrace{40}_{\text{TCP/IP}} + \underbrace{\text{MSS}}_{\text{Payload}})}}^{\text{Bytes per packet}}$$

- Max ACKs attacker can send per second:

Opt-ack's amplification factor

- Max bytes sent by victim per ACK:

$$\underbrace{\boxed{\frac{\text{Max window size}}{\text{MSS}}}}_{\text{Packets sent per ACK}} \times \underbrace{\boxed{(\underbrace{14}_{\text{Ethernet}} + \underbrace{40}_{\text{TCP/IP}} + \underbrace{\text{MSS}}_{\text{Payload}})}}_{\text{Bytes per packet}}$$

- Max ACKs attacker can send per second:

$$\boxed{\frac{\text{Attacker bandwidth (bytes/sec)}}{\underbrace{(14 + 40)}_{\text{Size of ACK packet}}}}$$

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) ~ **1336x**
- Window scaling lets you increase this by a factor of 2^14
- Window scaling amp factor: ~1336 * 2^14 ~ **22M**
- Using minimum MSS of 88: ~ **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.

Opt-ack defenses

- Nonces
  - Mostly solve problem, but not incremental
- ACK alignment
  - Send ~MSS or MSS-1; make hard to keep sync'd
  - ***Breaks if routers split packet***
- Random skip
  - Sender randomly skips a segment
  - Good receiver will ask for lost packet again (Sanity check)
  - Attacker won't be able to distinguish, will ACK
  - Costs receiver 1RT of performance