Lecture 6:

The Transport Layer

Katerina Argyraki, EPFL

Welcome back to Computer Networks.
This week we will continue with the transport layer, but we will focus on the famous Transmission Control Protocol or TCP, which is arguably one of the most successful technologies of our era.

Outline (from last lecture)

- Interaction with application layer
  - UDP
  - TCP
- Reliable data delivery
  - Imaginary protocol
  - UDP & TCP at the next lecture

In the last lecture, we discussed two services provided by the transport layer:
process-to-process communication (sockets etc) and reliable data delivery.

For the latter, we discussed in general terms, about families of protocols, like Go-back-N and Selective Repeat.

Outline (from last lecture)

- Interaction with application layer
  - UDP
  - TCP
- Reliable data delivery
  - Imaginary protocol
  - UDP & TCP at the next lecture

In the last lecture, we discussed two services provided by the transport layer:
process-to-process communication (sockets etc) and reliable data delivery.

For the latter, we discussed in general terms, about families of protocols, like Go-back-N and Selective Repeat.

UDP: reliability elements

- UDP does not really offer reliable data delivery
- **Checksums** to detect corruption

UDP provides only one aspect of reliable data delivery: detection of corruption. For that, it uses checksums: the sender includes in every segment a checksum (as part of the UDP header), and the receiver uses the checksum to ensure that — with a high probability — the segment was not corrupted.

Let's…

TCP: reliability elements

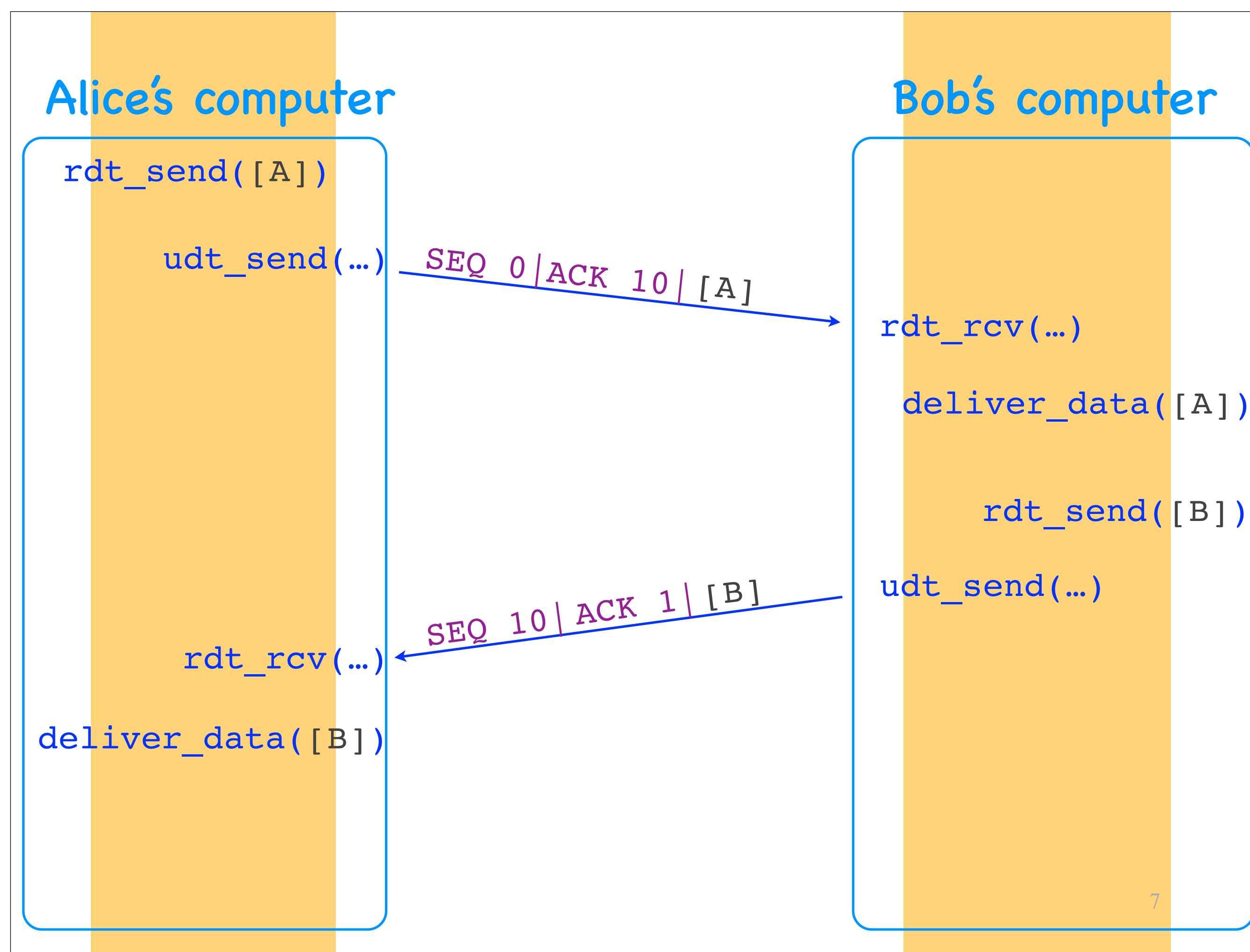
- Checksums to detect corruption
- ACKs to signal successful reception
- SEQs to disambiguate segments
- Timeouts to detect loss
- Retransmissions to recover from corruption+loss

TCP provides all the aspects of reliable data delivery that we discussed last week: …
and it does so in its own special way.

TCP: reliability elements

- Checksums to detect corruption
- ACKs to signal successful reception
- SEQs to disambiguate segments
- Timeouts to detect loss
- Retransmissions to recover from corruption+loss

First, we will focus on how TCP uses ACKs and SEQs.



(From now on, when I say "Alice", I mean "Alice's transport layer" and same for Bob.)

Consider the following scenario:

- The process running on Alice's computer sends one data byte (the character "A") to a process running on Bob's computer.
- Alice creates a segment that carries this data byte, which also carries SEQ 0.
- Bob receives the segment, extracts the data byte, and delivers it to the target process.
- Bob creates a segment that acknowledges receiving the data byte, which carries ACK 1.
- Alice receives the segment.

The first thing to note is that Bob's ACK number does not signal what he received, but what he is expecting: he received data byte 0, and he is expecting data byte 1, this is why he is sending ACK 1.

So: TCP uses cumulative ACKs, each one signalling the next data byte expected by the party sending the segment.

In most typical scenarios, Bob's process also has data to send to Alice's process. E.g., if Alice's process is a web client and Bob's process is a web server, Alice's process sends an HTTP GET request, and Bob's process responds with a file—so, both processes send data.

This means that:

- After receiving Alice's data, Bob's process also sends data (in our example, character "B") to Alice's process.
- Hence, Bob's segment does not only carry ACK 1, which acknowledges that he successfully received data byte 0, it also carries a data byte and a SEQ.
- When Alice receives Bob's segment, she extracts the data byte and delivers it to the target process.
- Moreover, Alice's segment does not only carry SEQ 0, it also carries an ACK, which is the next data byte that she is expecting from Bob.

So: All TCP segments carry a SEQ and an ACK. Even if it does not seem necessary. In our example here, the segment from Alice to Bob carries ACK 10, even if Alice does not need to acknowledge receiving any data from Bob.

Where did the number 10 come from? In this example, it is arbitrary — we will see better examples later. 10 happens to be the number of the data byte that Alice is expecting next from Bob.

SEQs & ACKs

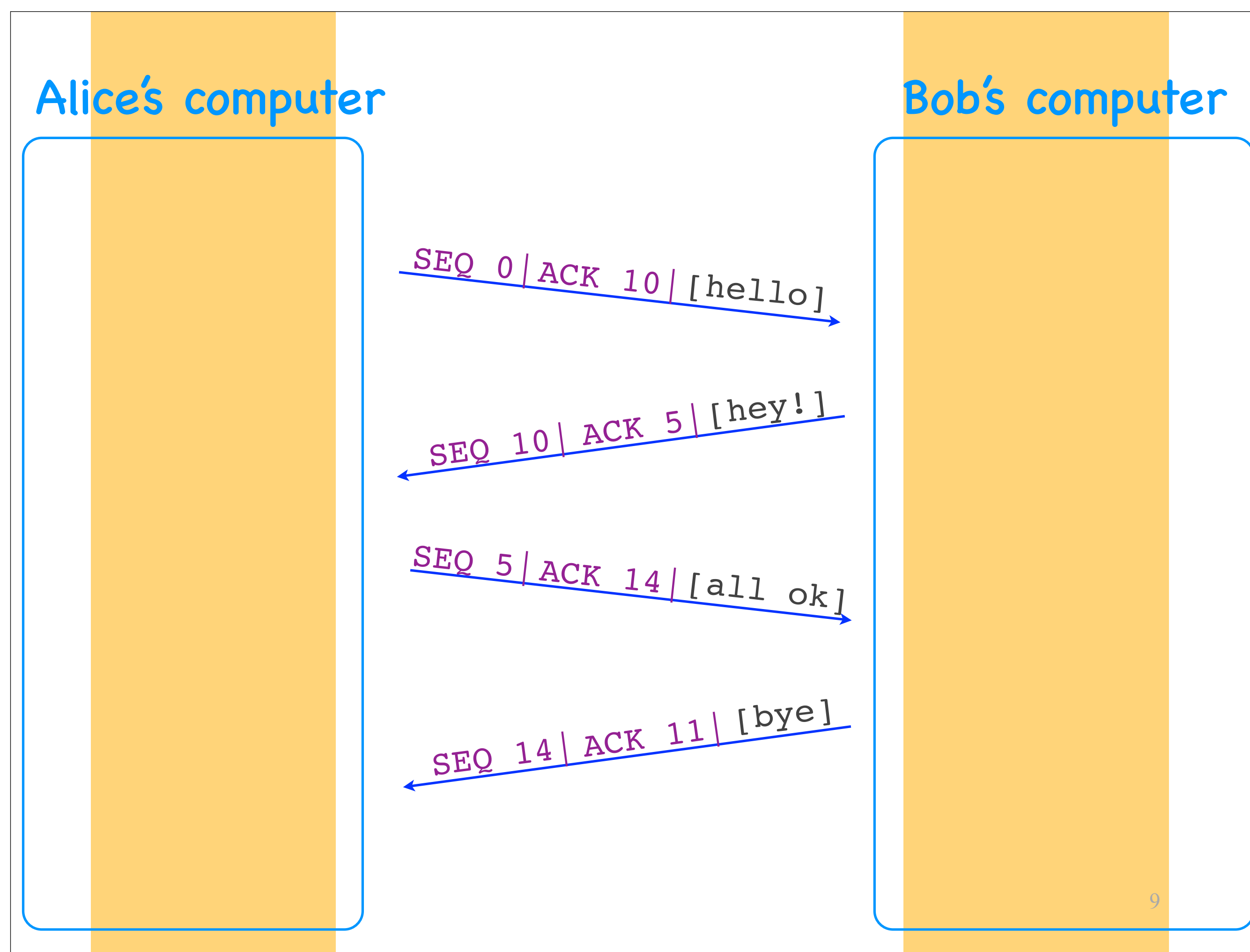
- Data bytes are implicitly numbered
- SEQ: # of the first data byte
- ACK: # of the next data byte that is expected (cumulative)
- Both always present, even if it appears unnecessary

In TCP, data bytes are implicitly numbered. Why do I say "implicitly"? Because it is not the case that each data byte is labelled with an actual number.

Rather, each TCP segment, inside the TCP header, carries a SEQ. This is the number of the first data byte carried in this segment's body.

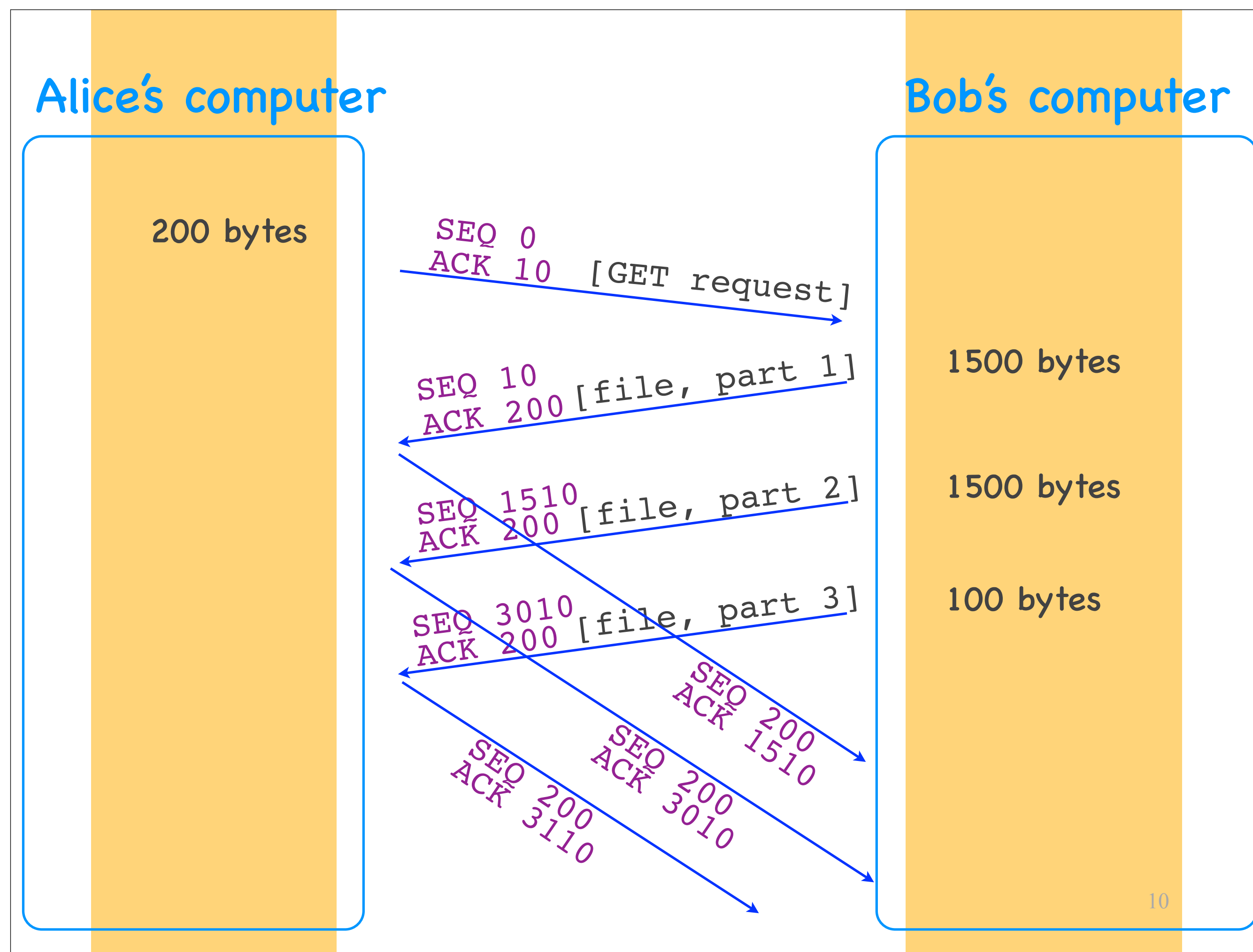
Moreover, each TCP segment, inside the TCP header, carries an ACK, which is the number of the next data byte that the sender of the segment is expecting from the other side. TCP ACKs are cumulative.

Both SEQs and ACKs are always present in the TCP header, even if they appear unnecessary.



Here's another example, where Alice's and Bob's processes exchange small words.

- Bob's first segment carries ACK 5 because Alice sent 5 data bytes, numbered 0 (the SEQ carried by Alice's first segment) to 4. This means that Bob is now expecting data byte 5.
- Alice's second segment carries ACK 14 because Bob sent 4 data bytes, numbered 10 (the SEQ carried by Bob's first segment) to 13. This means that Alice is now expecting data byte 14.
- Bob's second segment carries ACK 11 because Alice sent 6 bytes, numberer 5 to 10. This means that Bob is now expecting data byte 11.
- We can also guess that Alice's first segment carries ACK 10, because that is the SEQ on the next segment that Bob sends.



In this example, Alice's process is a web client, and Bob's process a web server. So, Alice's process sends an HTTP GET request of 200 bytes, and Bob's process responds with the corresponding file, divided in 3 segments.

Parenthesis: TCP cannot create segments of an arbitrary size. There is an upper bound, called Maximum Segment Size (MSS), determined by the underlying network properties. That's typically 1500 bytes, but may vary per network.

In this example:

- Alice's first segment carries a SEQ and an ACK.
- Bob's first segment also carries a SEQ and an ACK. The ACK is 200, because Bob received Alice's 200 bytes, numbered from 0 to 199, and is now expecting data byte 200.
- Bob's next two segments carry the same ACK number, because Alice did not send any new data, so Bob is still expecting data byte 200.
- Alice responds to each of Bob's segments with a segment that carries no data. Of course, each of these segments carries a SEQ and an ACK. The SEQ is 200, because that's the number of the first data byte that it *would* carry if it did carry data.

So: A TCP segment that carries no data, carries a SEQ that is equal to the number of the first data byte that the segment would carry if it did carry data.

Simple things to remember

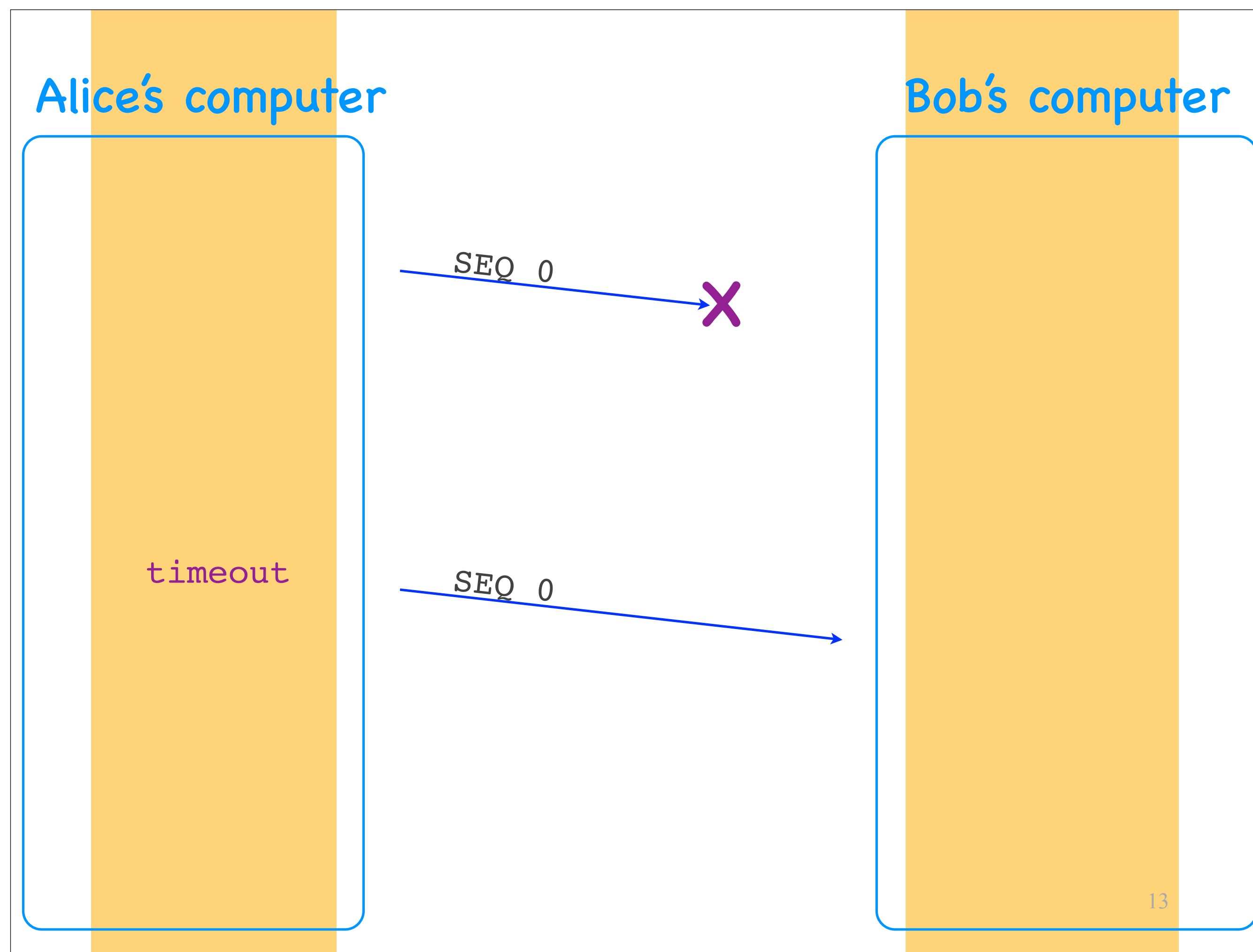
- A TCP connection may carry bidirectional communication
- A segment may or may not carry data (but it always carries a SEQ)
- There exists a maximum segment size (MSS), dictated by network properties

A few things to remember about TCP SEQs and ACKs: ...

TCP: reliability elements

- Checksums to detect corruption
- ACKs to signal successful reception
- SEQs to disambiguate segments
- **Timeouts** to detect loss
- **Retransmissions** to recover from corruption+loss

Now we will focus on how TCP uses timeouts and retransmissions.



When TCP times out, it retransmits *one* segment, which is the one with the oldest unacknowledged SEQ number.

How long should the timeout be?

-> How long should the timeout be?

The timeout should be a bit longer than the round-trip time (RTT) between the sender and the receiver.

Why? Because the sender should be fairly certain that a piece of data has been lost before timing out and retransmitting it. It shouldn't be the case that the segment carrying the data, or the segment acknowledging the data, has been delayed.

-> How should one compute the RTT?

Timeout calculation

- EstimatedRTT =
  0.875 EstimatedRTT + 0.125 SampleRTT
- DevRTT = function(RTT variance)
- Timeout = EstimatedRTT + 4 DevRTT

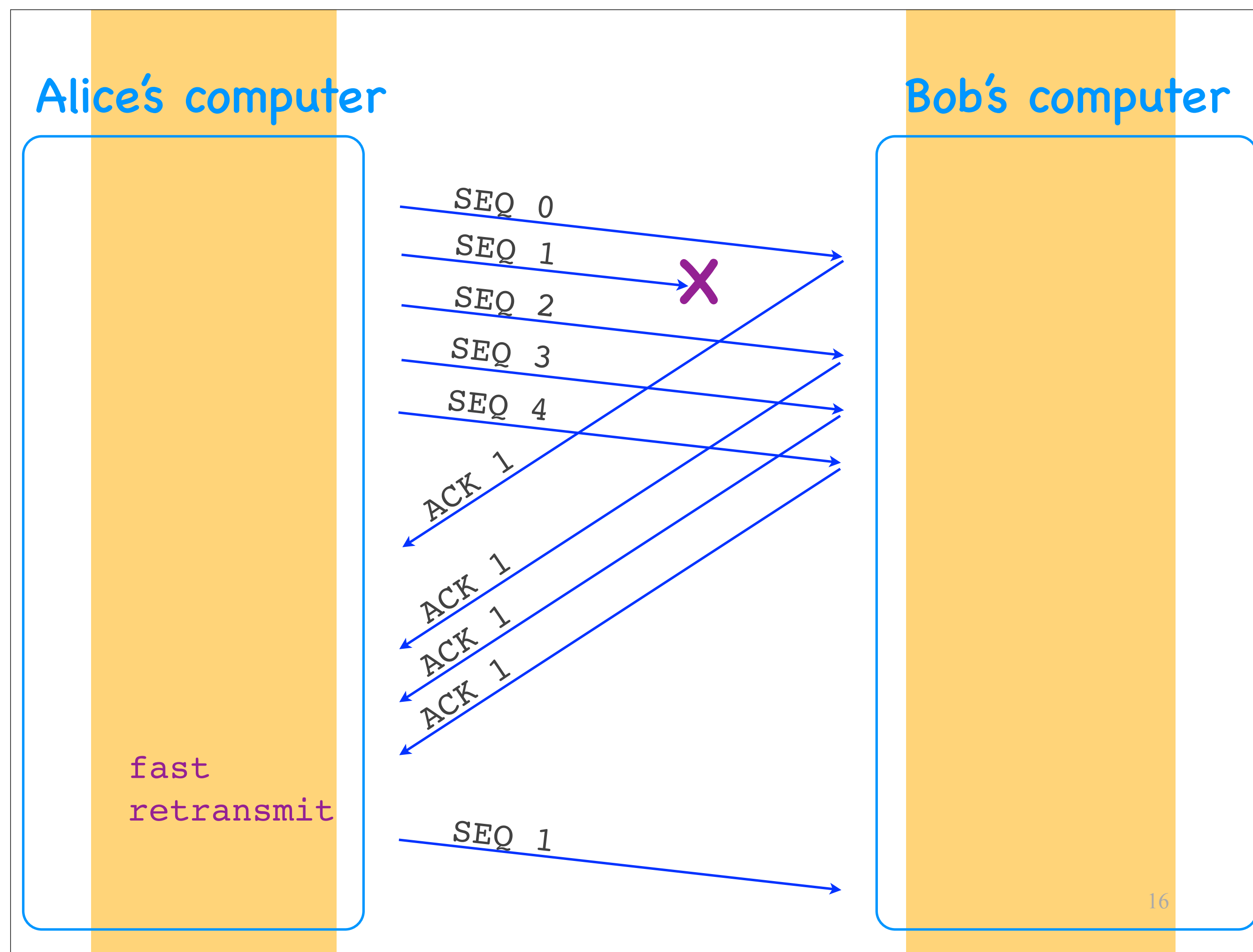
Empirical, conservative prediction of RTT

RTT estimation relies on a sliding window: the current estimate depends on past estimates and a new sample, and each term is appropriately weighed.

Why is this done this way? Well, think of how Google maps would predict how long it takes to go from point A to point B. It would take into account how long it took the last car to go from A to B, but it would not take into account only that one data point, because that car may have encountered some extraordinary temporary situation.

The timeout depends on the estimated RTT and a function of the RTT variance. Why take the variance into account? Because the more the RTT varies over time, the more conservative the sender needs to be when setting the timeout, so that it does not timeout prematurely.

The point is that TCP timeout calculation is empirical. It uses weights and numbers that have turned out to work well in practice. The numbers on this slide are from your book and they correspond to a particular implementation, but other implementations might use different numbers.



Suppose Alice sends 5 segments, each carrying 1 data byte.
Suppose Alice's 2nd segment is lost.

When Bob receives Alice's first segment, he responds with a segment that carries ACK 1, because that's the next byte he is expecting from Alice.

-> When Bob receives Alice's 3rd, 4th, and 5th segment, what ACK number does he respond with?

It's still ACK 1, because TCP ACKs are cumulative. Bob cannot respond with ACK 3, because that would mean that he has successfully received all data bytes from 0 to 2, and he is now expecting 3, which is not the case.

So, he keeps responding with ACK 1.

When Alice receives all the duplicate ACKs, she can guess that the 2nd segment was lost.
Hence, she does not need to wait for a timeout, she can immediately retransmit.

We call this retransmission (that occurs in response not to a timeout but duplicate ACKs), a "fast retransmit".
When it happens, Alice retransmits one segment, which is the one with the oldest unacknowledged SEQ.

Two retransmission triggers

- Timeout => retransmission of oldest unacknowledged segment
- 3 duplicate ACKs => fast retransmit of oldest unacknowledged segment
  - avoid unnecessary wait for timeout
  - 1 duplicate ACK not enough <= network may have reordered a data segment or duplicated an ACK

So: In TCP, there are two events that trigger retransmission: …
Both of them result in 1 retransmission of the oldest unacknowledged segment.

TCP: reliability elements

- Checksums to detect corruption
- ACKs to signal successful reception
- SEQs to disambiguate segments
- Timeouts to detect loss
- Retransmissions to recover from corruption+loss

This completes our discussion of TCP's reliability elements.

Is TCP Go-back-N or SR?

- Go-back-N: cumulative ACKs,
  retransmits multiple segments
- SR: selective ACKs,
  retransmits 1 segment on timeout
- TCP: cumulative ACKs,
  retransmits 1 segment => Go-back-N/SR mix

Last question before we move on: does TCP belong to the Go-back-N or to the Selective Repeat family?

TCP elements

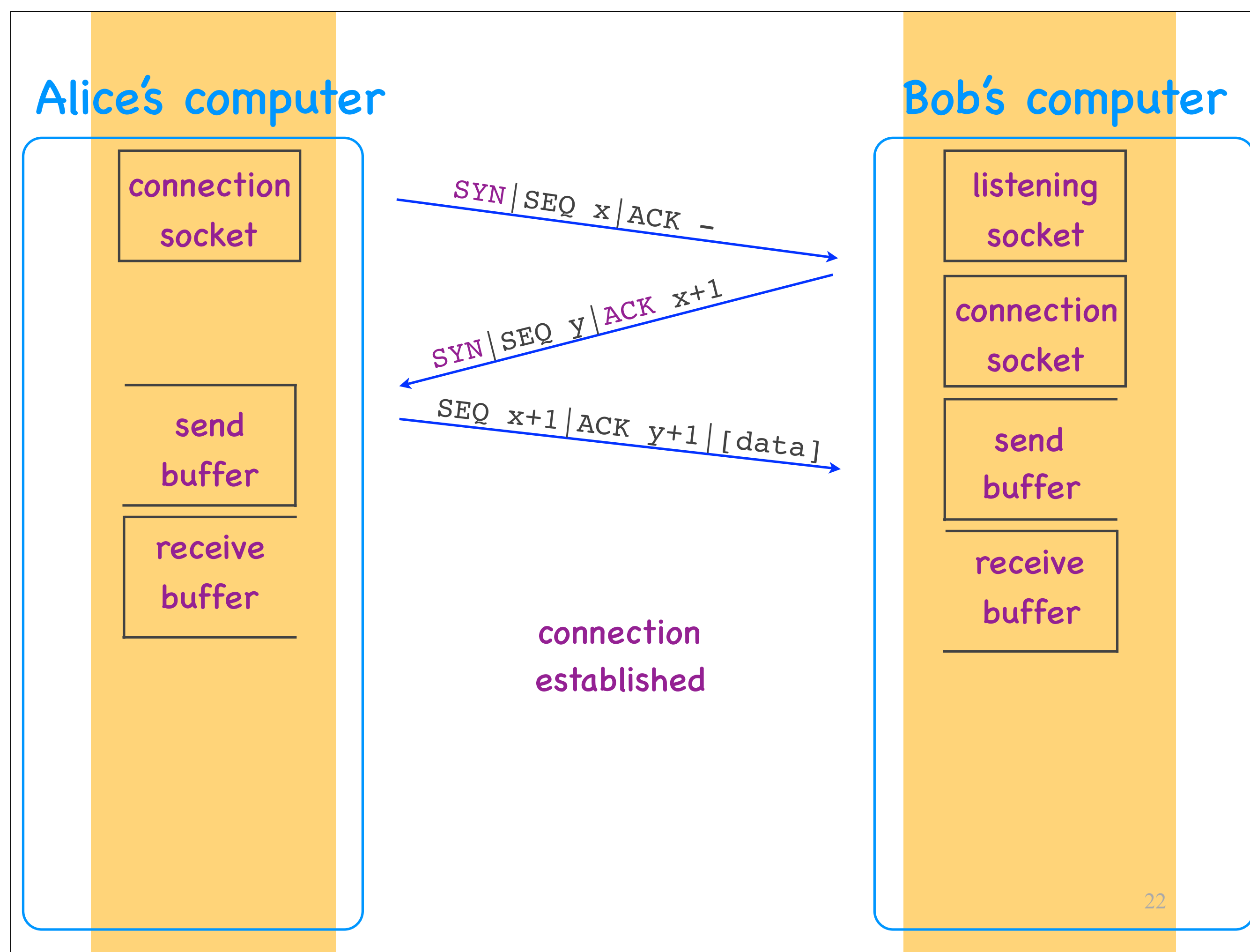
- Connection setup and teardown
- Connection hijacking
- Connection setup (SYN) flooding
- Flow control
- Congestion control

TCP does not provide only reliable data delivery, it has other interesting elements.

TCP elements

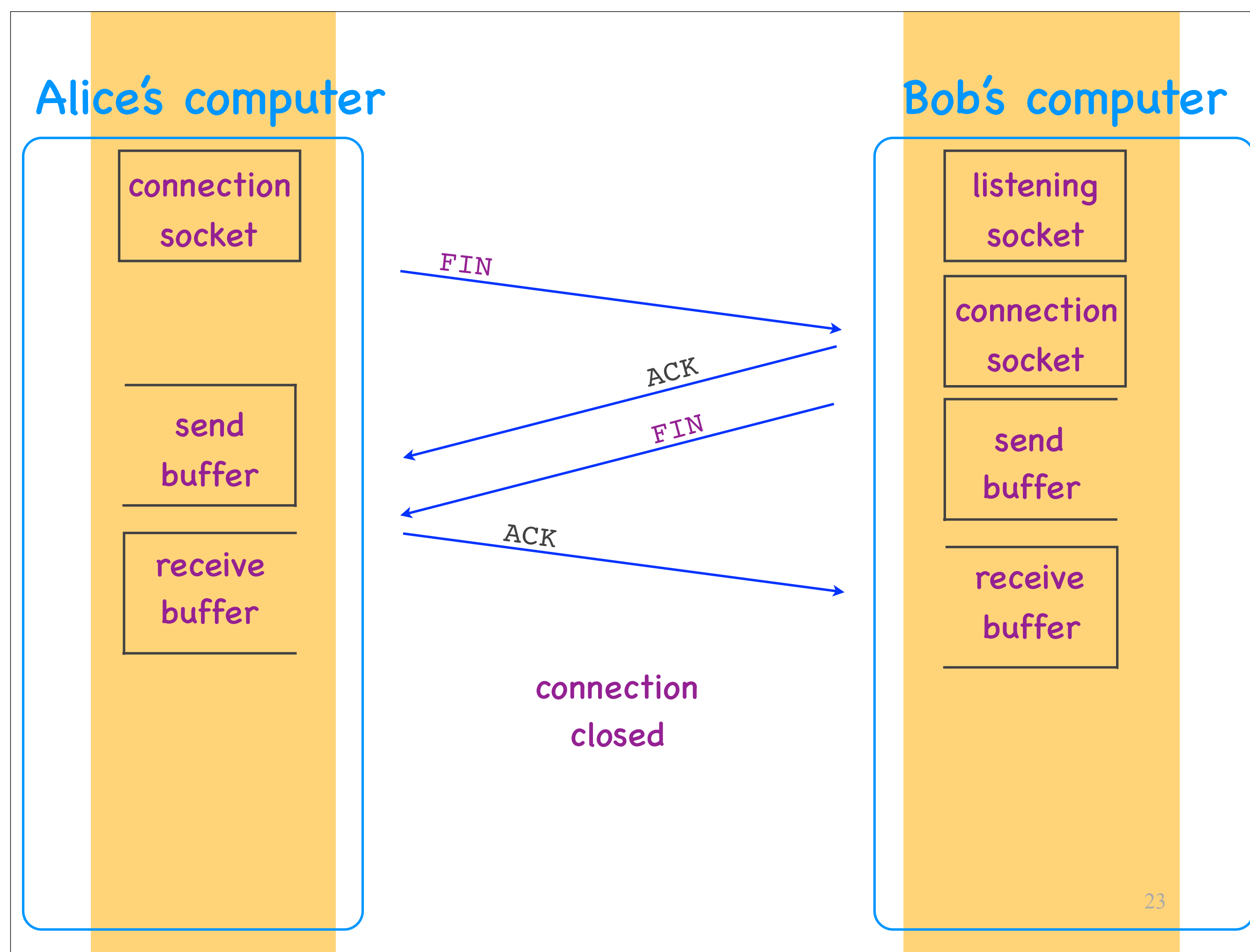
- Connection setup and teardown
- Connection hijacking
- Connection setup (SYN) flooding
- Flow control
- Congestion control

Let's look at TCP connection setup and teardown.



This is an example of a TCP connection setup:

- Bob's process has opened a listening socket.
- Alice's process creates a connection socket to communicate with Bob's process. As a result, Alice sends a connection setup request to Bob. That's a segment that carries what is called the SYN flag (which is basically a particular bit in the TCP header). It also carries a SEQ and an ACK (though the value of the ACK is irrelevant).
- When Bob receives the connection setup request, it passes it to Bob's process. If the process accepts the request, that creates a connection socket for communicating with Alice's process. In response, Bob sends a segment that also carries the SYN flag. It also carries a SEQ (that has nothing to do with Alice's SEQ) and an ACK (which is Alice's SEQ + 1, i.e., it acknowledges that Bob received the connection setup request and is now expecting the first data byte).
- When Alice receives Bob's response, she allocates a send and a receive buffer for this particular TCP connection. Then she sends another segment that may carry data.
- When Bob receive's Alice's 2nd segment, he also allocates a send and a receive buffer for this particular TCP connection. At this point, we say that Alice and Bob have established a TCP connection.



- When Alice's process does not need to send any more data to Bob's process, Alice sends a segment to Bob that carries what is called a FIN flag (which is another bit in the TCP header).
- When Bob receives it, he acknowledges it like any other segment, and deletes his receive buffer, because Alice will not be sending any more data on this connection.
- Alice also deletes her send buffer.
- When Bob's process does not need to send any more data to Alice's process, Bob sends a segment to Alice that carries the FIN flag.
- When Alice receives it, she acknowledges it like any other segment, and deletes her receive buffer.
- Bob also deletes his send buffer.
- At this point, Alice and Bob can release all the resources that were dedicated to this connection, including the connection sockets, and we say that Alice and Bob have closed their TCP connection.
- Beware, the listening socket is still there, because it is not bound to a particular TCP connection, it's listening for new connection-setup requests.

Connection setup

- 3-way handshake
  - "TCP client": end-system initiating the handshake
  - "TCP server": the other end-system
- First 2 segments carry a SYN flag
  - 1-bit field in the TCP header
- "TCP connection" = resources (sockets, buffers…) allocated for communication

So:

A TCP connection setup occurs though a 3-way handshake. We use the term "TCP client" to refer to the entity that initiates the handshake, and "TCP server" to the other entity.

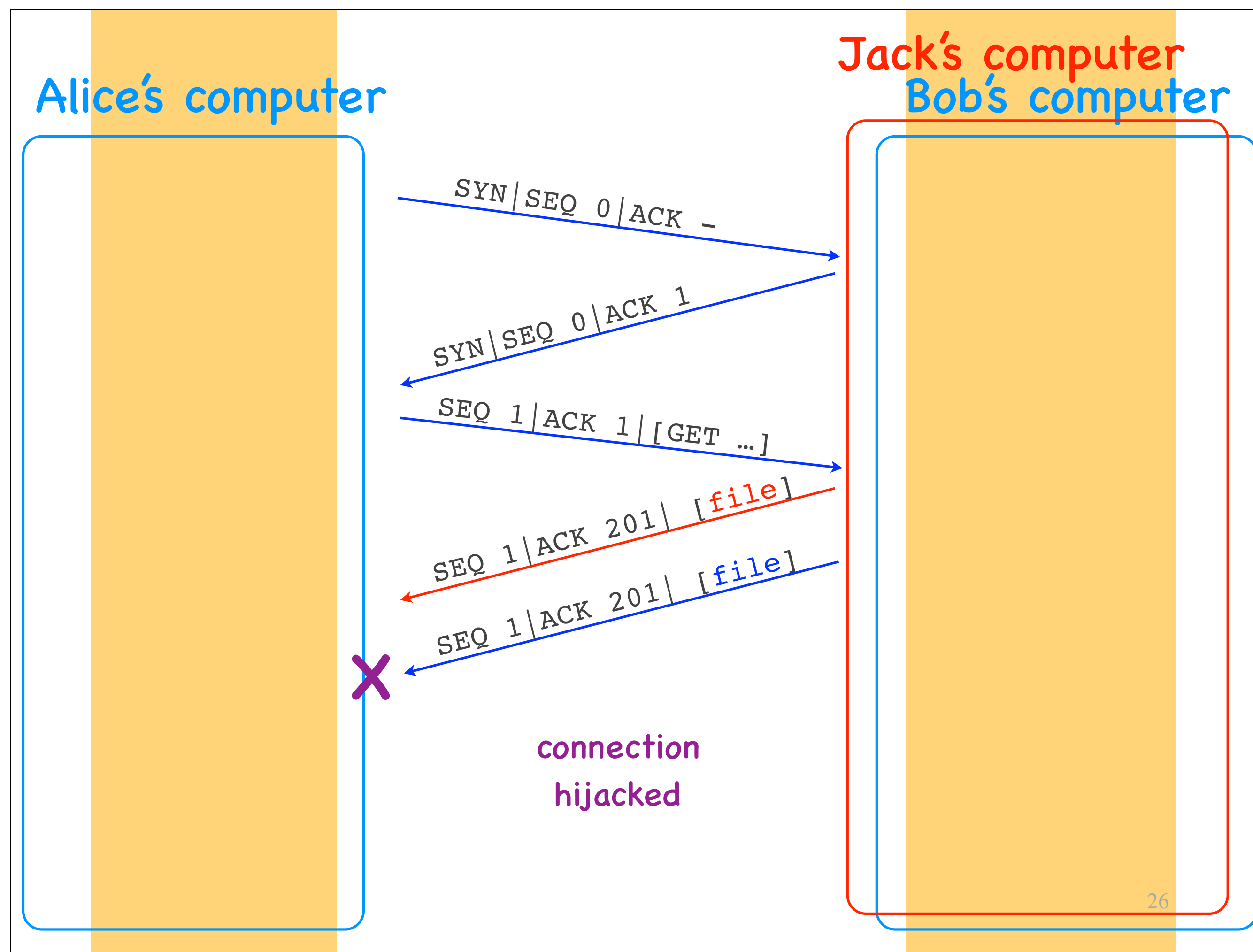
The first 2 segments carry a SYN flag, which is a 1-bit field in the TCP header.

When we say that two end-systems have "established a TCP connection," we mean that they have allocated all the resources, like sockets and buffers, that are necessary for their communication.

TCP elements

- Connection setup and teardown
- Connection hijacking
- Connection setup (SYN) flooding
- Flow control
- Congestion control

Now let's look at an attack that targets TCP connections.



Consider the scenario where Alice's web client and Bob's web server have established a TCP connection. With her 2nd segment (the one that completes the 3-way handshake), Alice sends an HTTP GET request for a file.

Now consider an attacker, Jack, who is NOT on the path between Alice and Bob, i.e., he cannot intercept their communication. Suppose Jack somehow knows that Alice has just requested a file from Bob and wants to impersonate Bob and send the wrong file to Alice.

To do that, Jack constructs a segment that looks identical to the legitimate one sent by Bob, with the only difference that the file it contains is the wrong one. Jack sends the segment to Alice.

Suppose Alice receives first Jack's segment, then Bob's. Because the two segments have identical TCP headers, Alice thinks Bob's is a duplicate and drops it.

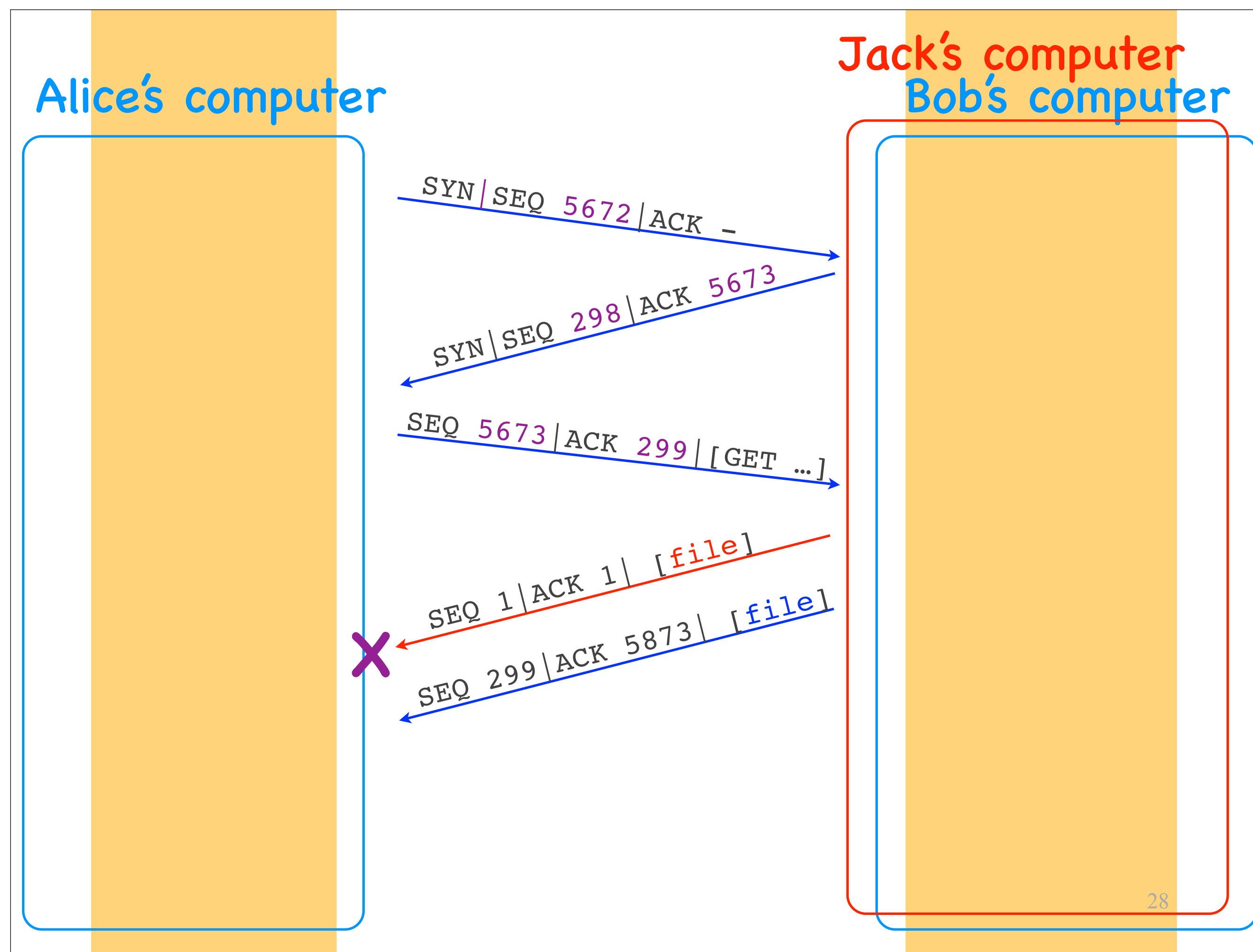
At this point, Jack has successfully hijacked Alice and Bob's TCP connection.

The reason this can be done is that — as we have described things so far — TCP headers are predictable => easy to fake. If Jack knows the size of Alice's HTTP GET request, he can guess what the ACK number on Bob's 2nd segment will be. If Jack can also guess Bob's initial sequence number, then he can reconstruct exactly the TCP headers of Bob's 2nd segment.

How to prevent connection hijacking?

-> How can one prevent connection hijacking?

One way is to make the TCP segments unpredictable, hard to guess by a third party that cannot see them.



When Alice and Bob setup their connection, they choose random sequence numbers that would be hard for Jack to guess (again, as long as he is not on the path and cannot intercept and read Alice/Bob's communication).

So, now, when Jack sends his fake segment, that has the wrong SEQ, and Alice rejects it, while she accepts the true segment sent by Bob.

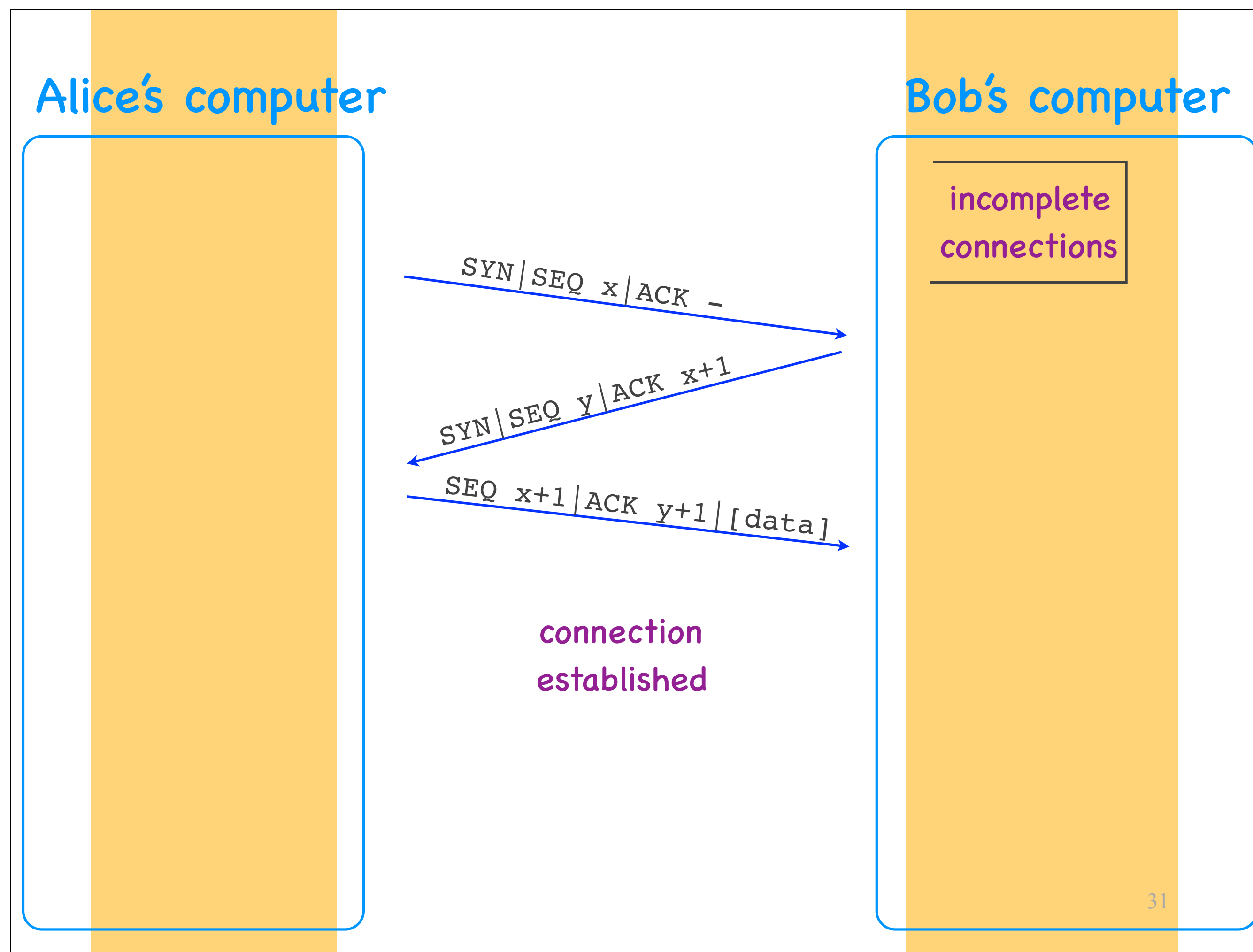
Connection hijacking

- Attacker impersonates TCP server (or client)
  - sends segment that appears to be coming from the impersonated end-system
- Approach: fake valid segment
  - if the TCP header predictable
- Solution: make TCP header (SEQs) unpredictable

TCP elements

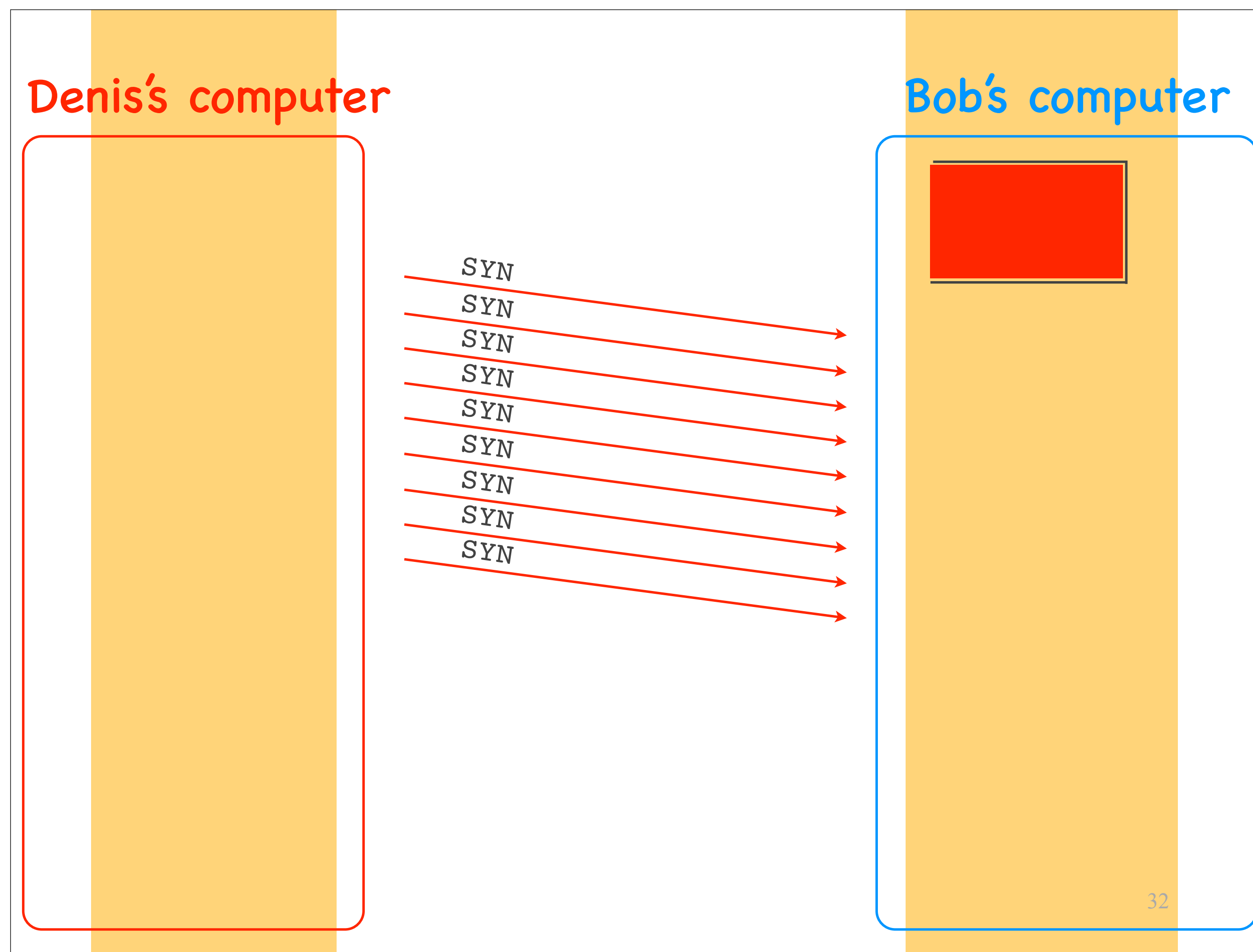
- Connection setup and teardown
- Connection hijacking
- **Connection setup (SYN) flooding**
- Flow control
- Congestion control

Let’s look at another attack that targets TCP connections.

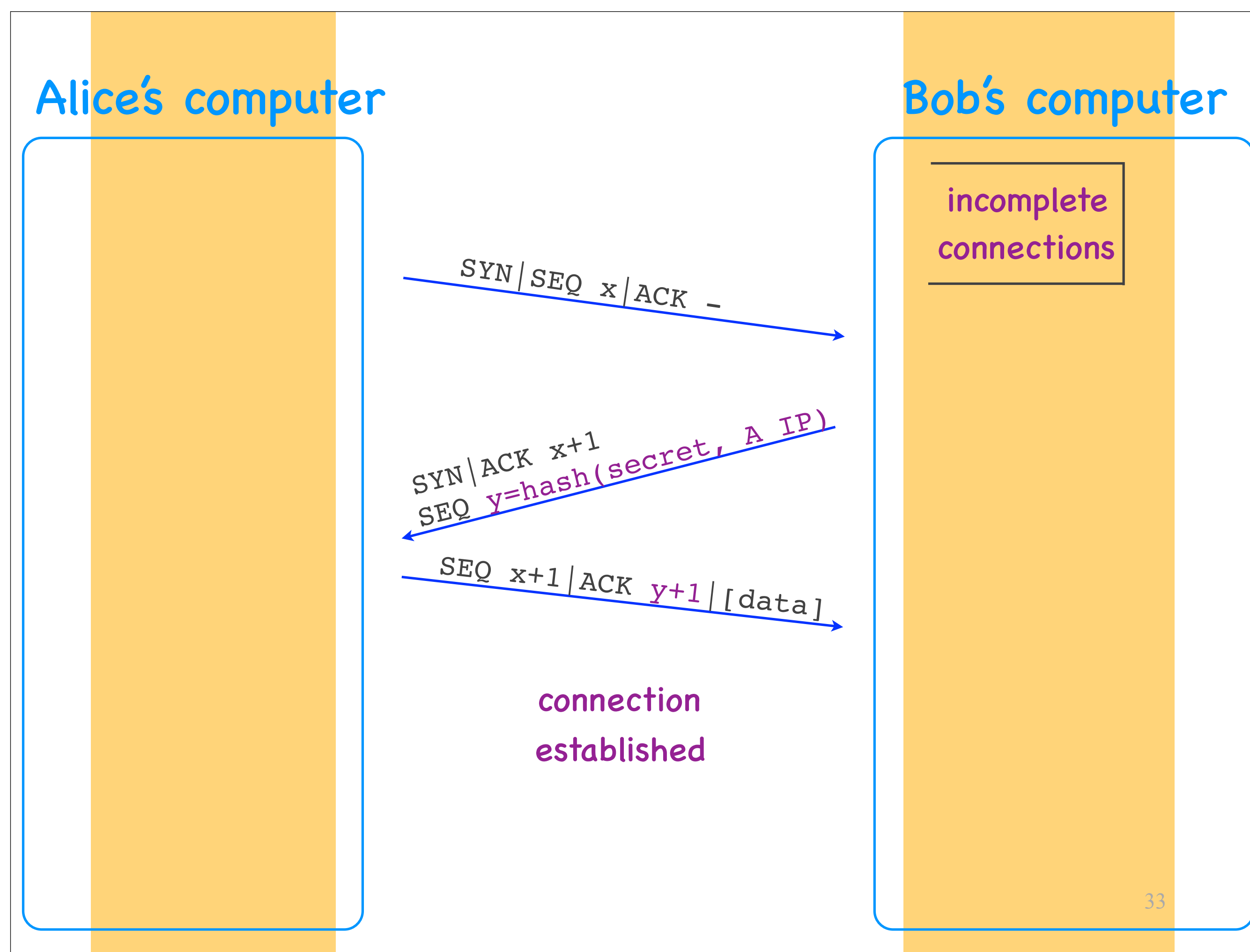


Any end-system that uses TCP maintains a piece of memory — a buffer — for storing information on incomplete connections.

For instance, when Bob receives Alice's SYN segment, he responds but also stores a copy of the SYN segment in this buffer. When Alice sends her 2nd segment (that completes the 3-way handshake), Bob checks the buffer, retrieves the copy of Alice's SYN segment and that way he is certain that he previously agreed to connect to Alice.



The incomplete-connection buffer is a convenient target for an attacker: he can send a lot of SYN segments (without ever completing any connection), thereby causing Bob's incomplete-connection buffer to overflow. If that happens, then Bob must drop any new connection setup request => cannot accept any new TCP connections.

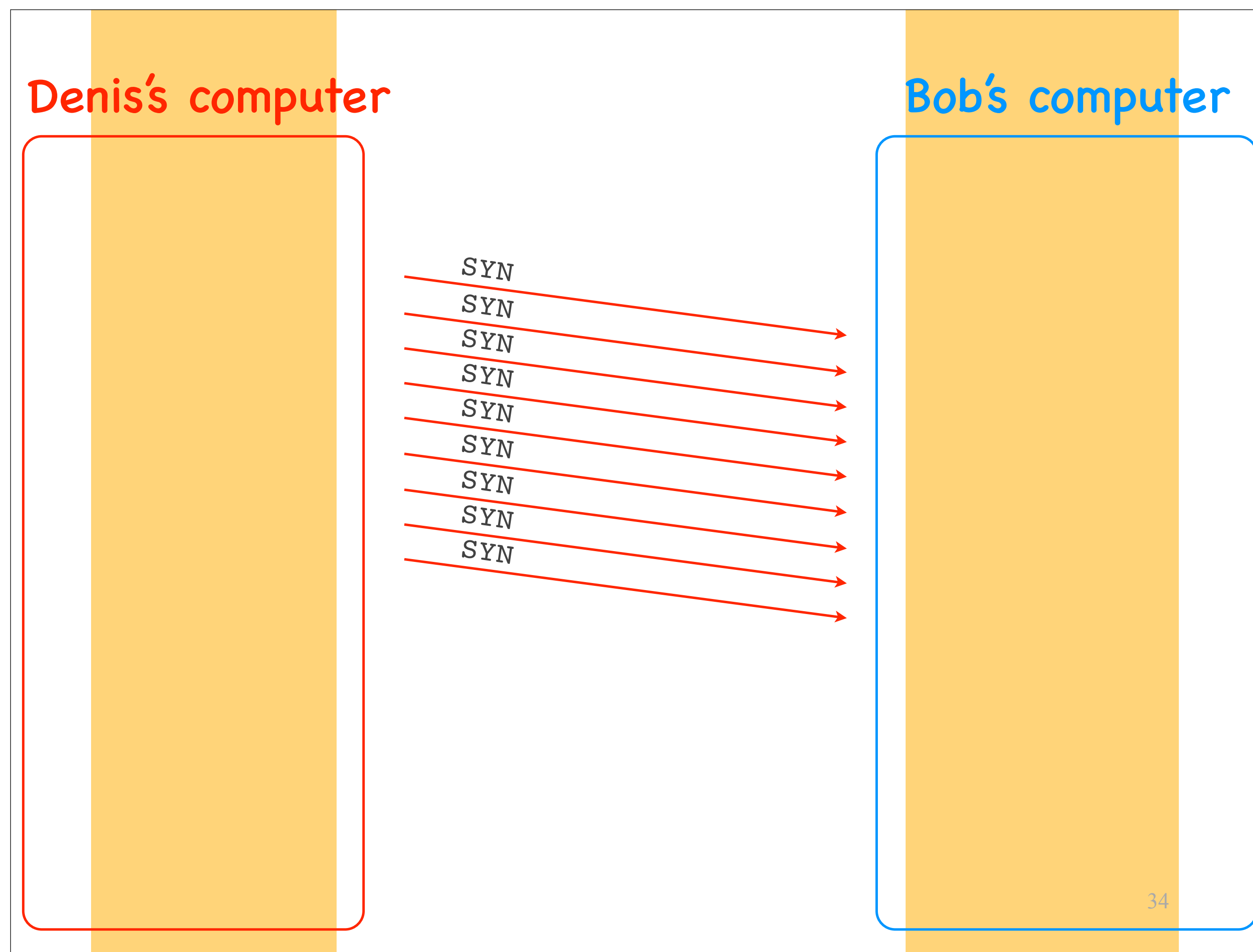


One way to prevent this attack is to remove the incomplete-connection buffer from the picture.

Instead, Bob embeds in his initial sequence number enough information that, when Alice sends her 2nd segment, Bob can determine whether he previously agreed to this connection, just by looking at the ACK number on Alice's 2nd segment.

In more detail:

- Bob keeps a secret known only to him.
- He sets his initial sequence number y to hash(secret, Alice's IP).
- When Bob receives Ali he's 2nd segment, he reads the ACK number (y+1), subtracts 1, and checks whether it is equal to hash(secret, Alice's IP).
- If yes, then, with a very high probability, Bob previously agreed to connecting to Alice. Otherwise, it is very unlikely that Alice could have guessed the correct ACK number to put on her 2nd segment.



Of course, Denis can still send a lot of SYN segments to Bob. However, now, Bob does not rely on the incomplete-connection buffer to keep track of which connections he agrees to. Hence, Denis cannot prevent Bob from accepting new TCP connections by overflowing this buffer.

To clarify: The solution I just described does not prevent Denis — or any malicious entity — from establishing a connection with Bob. It prevents Denis from launching a specific type of attack: disrupt Bob's ability to accept new connections by overflowing his incomplete-connections buffer. Why is this type of attack important enough to discuss (when there are so many other bad things that Denis could do)? Because it is extremely simple and cheap to launch. Overflowing an incomplete-connection buffer is typically easier than overwhelming Bob's CPU or bandwidth.

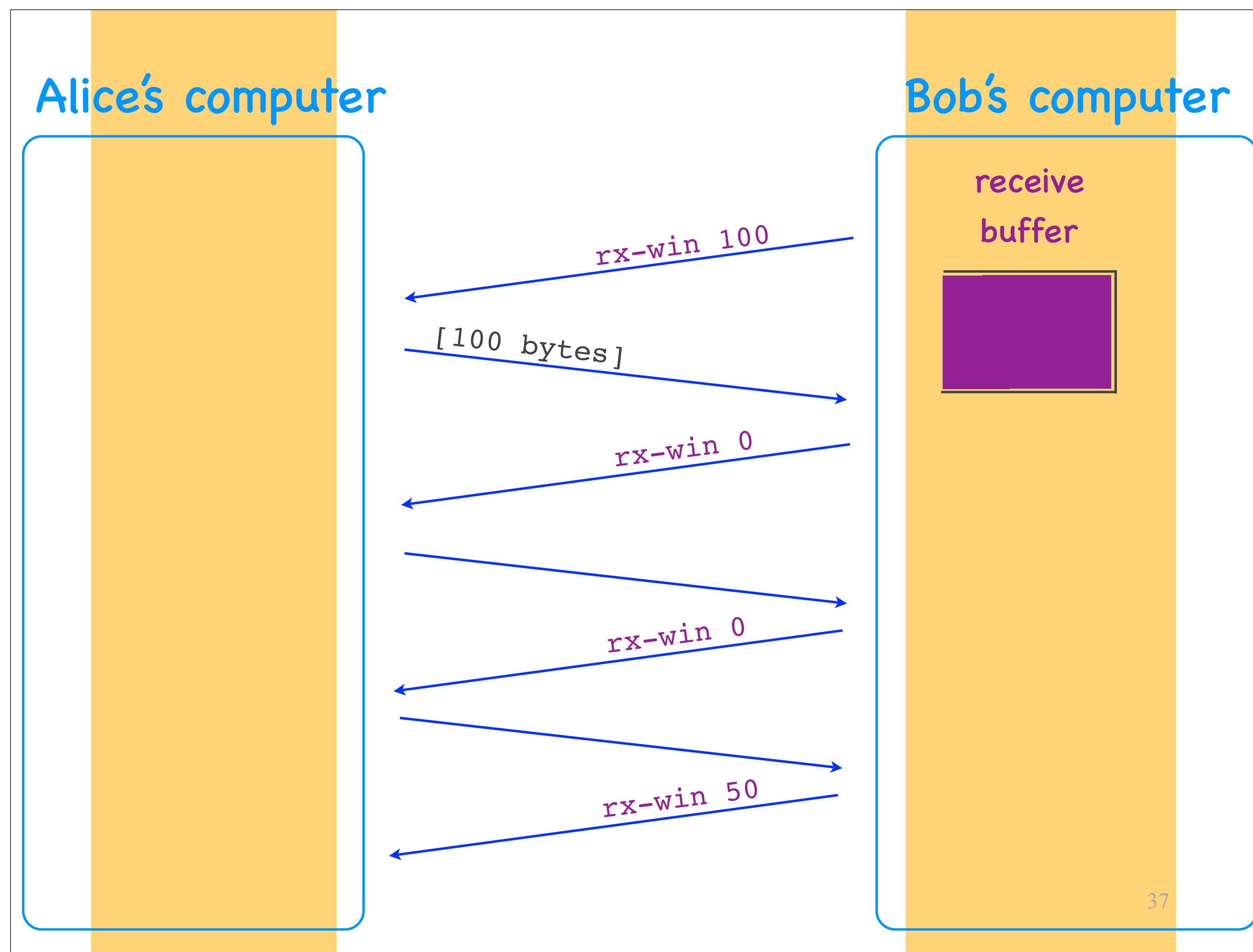
SYN flooding

- Attacker exhausts buffer for incomplete connections
  - sends lots of connection setup requests
- Problem: one small resource affects all TCP communication
- Solution: remove the resource
  - pass the state to the TCP client

TCP elements

- Connection setup and teardown
- Connection hijacking
- Connection setup (SYN) flooding
- Flow control
- Congestion control

On top of reliable data delivery, TCP provides “flow control.”



The goal of flow control is to protect a receiver from being overwhelmed by a sender.

To achieve it, the receiver communicates to the sender up to how many bytes the sender can send to it. This number is equal to the free space in the receiver's receive buffer (allocated for this particular connection).

This information is stored in a TCP header field called "receiver window".

In this example, Bob has 100 bytes left in his receive buffer, so he tells Alice that she can send him up to 100 bytes.

It is possible that the receiver window "closes", which means that the receiver tells the sender that it cannot send any more bytes (for the moment). If this happens, the sender needs to periodically probe the receiver, i.e., send a segment that carries no data or new ACK, just to elicit another segment in response, such that the receiver can communicate the new receiver-window value.

Flow control

- Goal: not overwhelm the receiver
  - not send at a rate that the receiver cannot handle
- How: "receiver window"
  - spare room in receiver's rx buffer
  - receiver communicates it to sender as TCP header field

TCP elements

- Connection setup and teardown
- Connection hijacking
- Connection setup (SYN) flooding
- Flow control
- Congestion control

The more complicated sibling of flow control is congestion control.

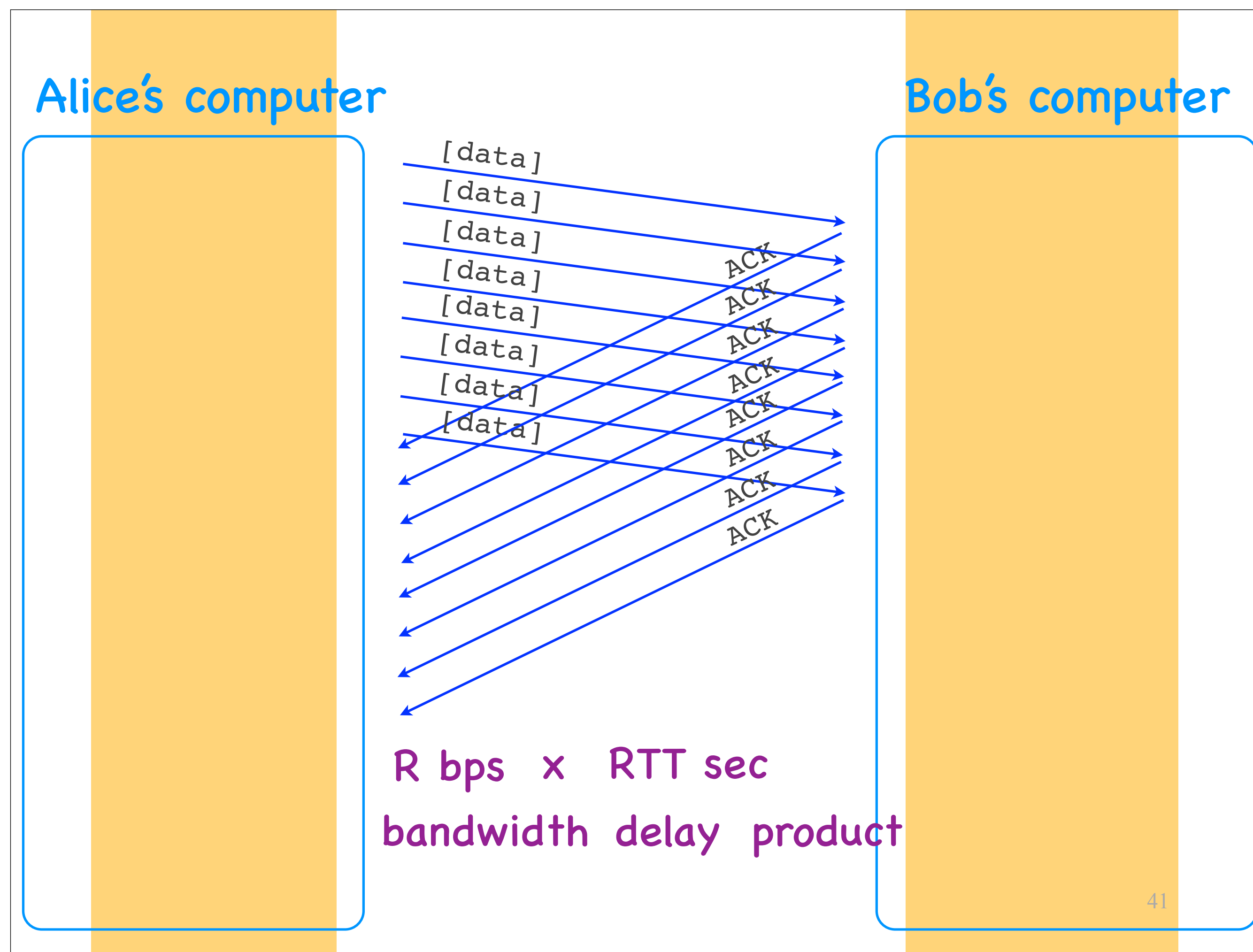
Congestion control

- Goal: not overwhelm the network
  - not send at a rate that the would create network congestion
- How: "congestion window"
  - number of unacknowledged bytes that the sender can transmit without creating congestion
  - sender estimates it on its own

The goal of flow control was to protect the receiver from being overwhelmed; the goal of congestion control is to protect the network from being overwhelmed, from being congested.

To achieve this, the sender uses various techniques to estimate a "congestion window" = the max number of unacknowledged bytes it should send so as to avoid contributing to congestion.

-> Why is congestion control more complicated than flow control? For flow control, the receiver tells the sender how many bytes the sender may send so as not to overwhelm the receiver. Why can't we do congestion control in a similar way? Why can't someone tell the sender how many bytes it may send so as not to overwhelm the network?



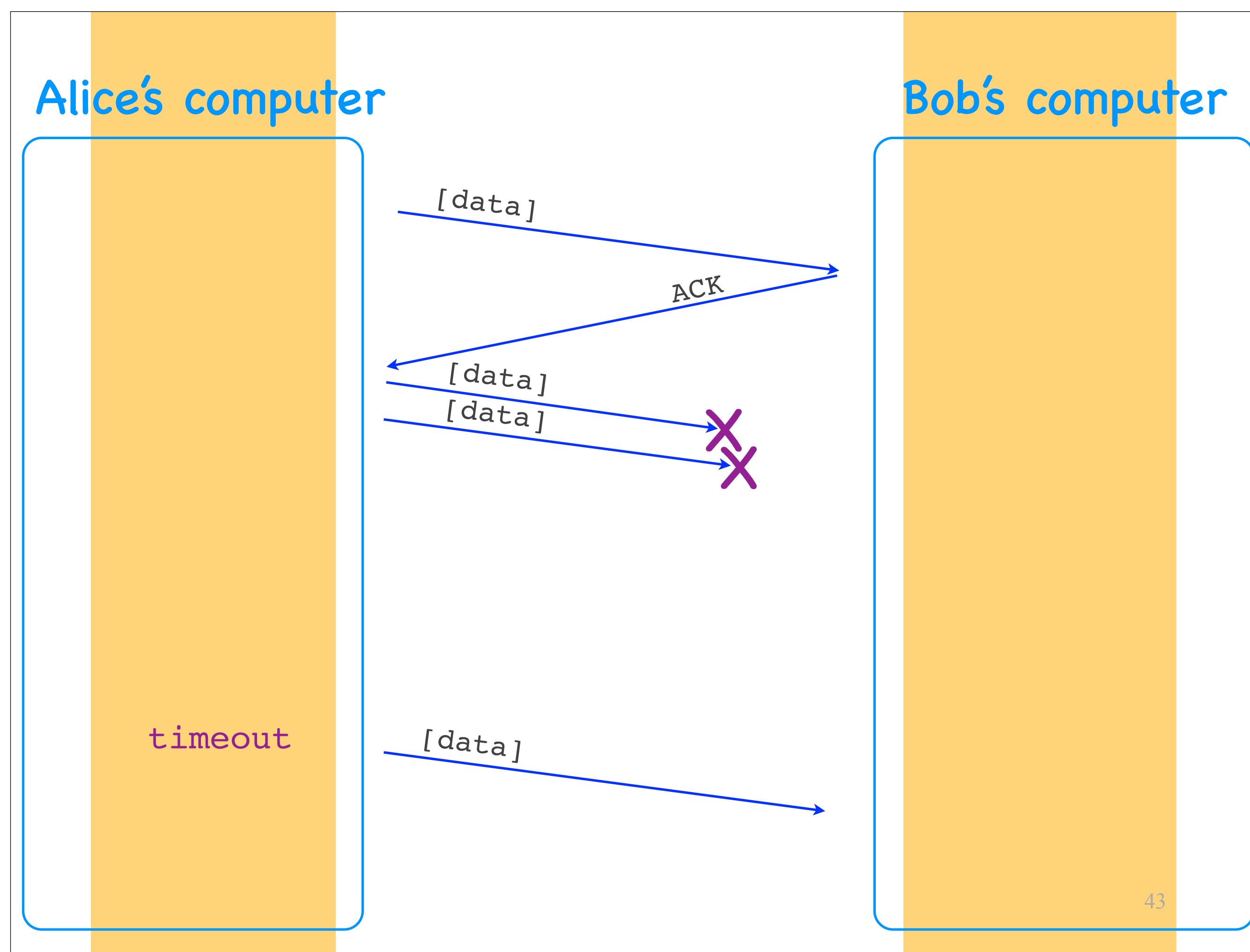
Let's see how TCP congestion control works.

First of all, what is the maximum congestion window size that makes sense?

Alice can keep sending unacknowledged bytes until she receives an ACK for the first segment she sent out.
In this case, Alice transmits data at some rate R (which is her throughput to Bob) and for a period of RTT (the round-trip time to Bob).
Hence, she transmits a total of R x RTT bits.
This is called the bandwidth-delay product, and it is the maximum amount of traffic that Alice can transmit before receiving an ACK.

Bandwidth-delay product

- Max amount of traffic that the sender can transmit until it gets the first ACK
- = the maximum congestion window size that makes sense



A TCP sender does not always use the maximum sender window size possible.

To avoid congestion, it makes sense to be conservative:

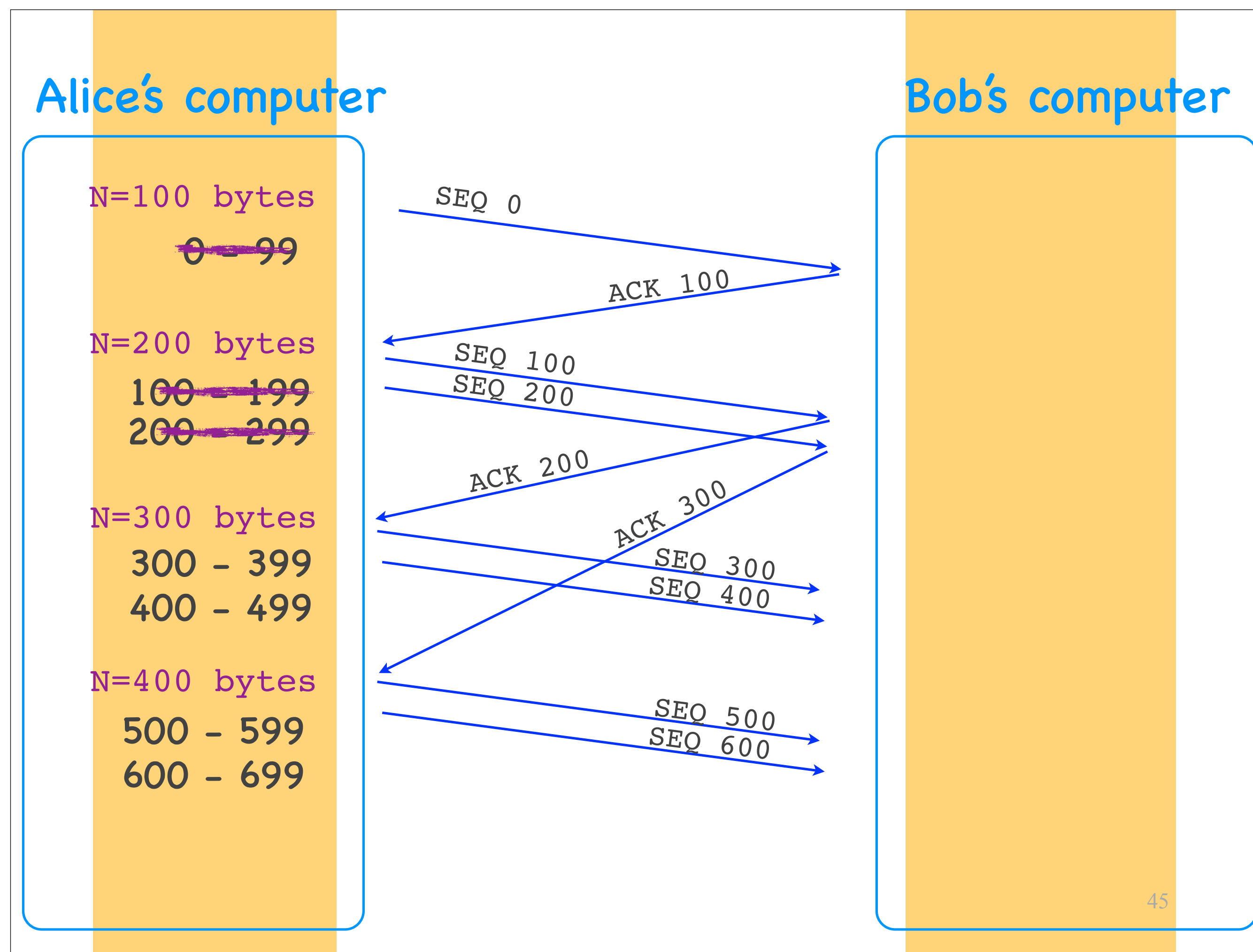
- Alice starts out cautiously: she first sends out a single segment.
- If she gets an ACK for the data she sent out, she becomes more optimistic and sends out more segments.
- If she times out waiting for an ACK, she retransmits, but she also goes back to sending a single segment.

Self-clocking

- Sender guesses the "right" congestion window based on the ACKs
- ACK = no congestion, increase window
- No ACK = congestion, decrease window

This behavior of TCP is called self-clocking.
It refers to the fact that TCP, on the sender side, on its own determines the right congestion window based on the ACKs it gets from the receiver.
Every time the sender gets an ACK, this indicates that there was no congestion,
so it increases the window.
Every time the sender does not get an ACK, this indicates that there was congestion, so it decreases the window.

What algorithm exactly does TCP use to increase and decrease the window size?



Alice has established a TCP connection to Bob and wants to send him data.
The maximum segment size (MSS) of the network between them is 100 bytes.

Let's first consider the case where there is no packet loss.

In this case:

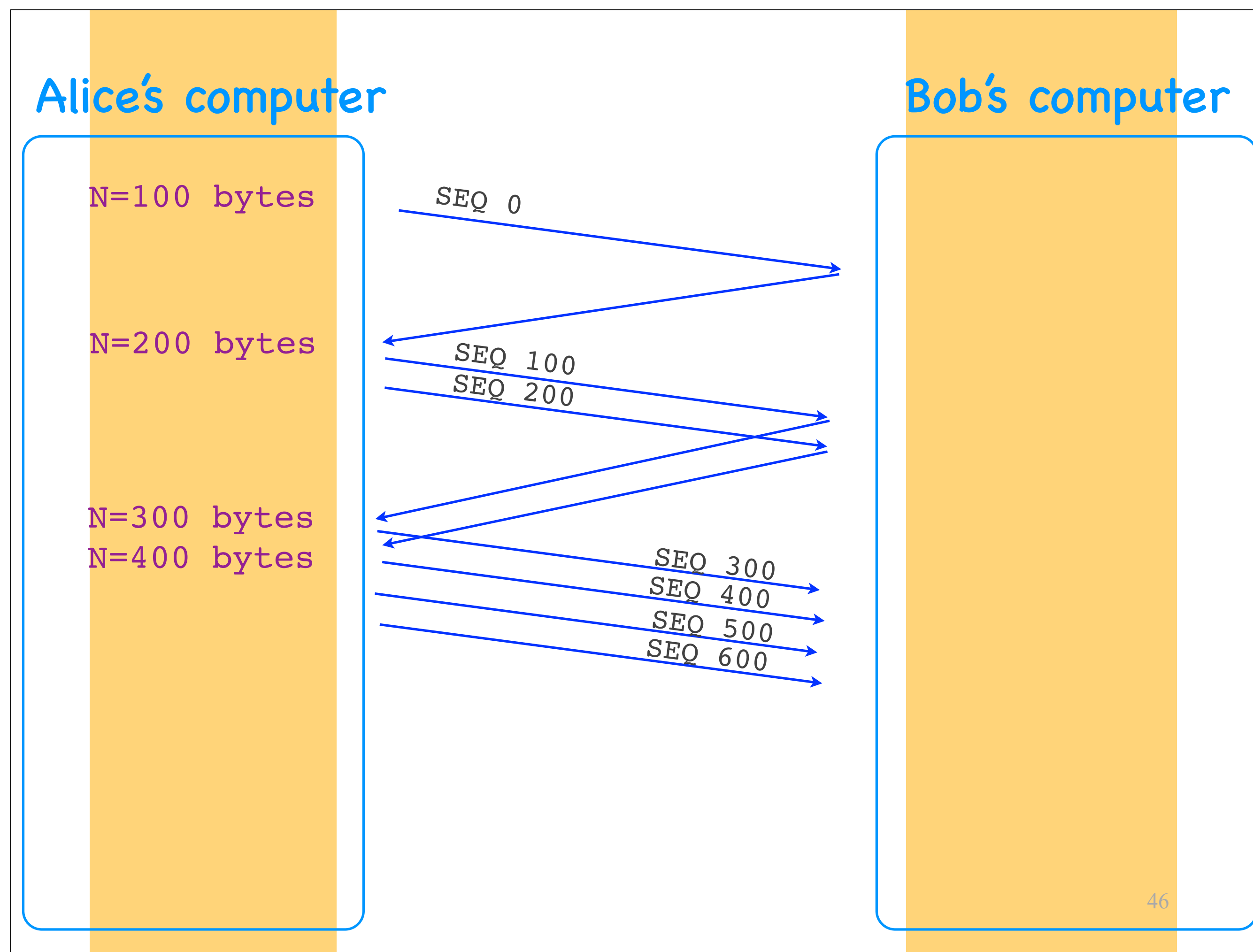
Alice sets her congestion window to 1 MSS.
So, she sends 1 segment carrying 100 bytes (SEQ 0).

When she receives ACK 100,
which means that Bob received bytes 0-99,
she increases her congestion window by 1 MSS,
which means that she can now send up to 200 un-ACK-ed bytes.

When she receives ACK 200,
which means that Bob received bytes 100-199,
she again increases her congestion window by 1 MSS,
which means that she can now send up to 300 un-ACK-ed bytes.
She has already sent 100 un-ACK-ed bytes (SEQ 200),
so she sends 200 more (SEQs 300 and 400).

When Alice receives ACK 300,

which means that Bob received bytes 200-299,
she again increases her congestion window by 1 MSS,
which means that she can now send 400 un-ACK-ed bytes.
She has already sent 200 un-ACK-ed bytes (SEQs 300 and 400),
so she sends 200 more (SEQs 500 and 600).



I will squeeze things a bit to give you a better sense of how things happen over time.

In summary:
- Alice starts out cautiously, by sending 1 segment.
- Every time she receives a new ACK number, she increases her congestion window by 1 MSS.

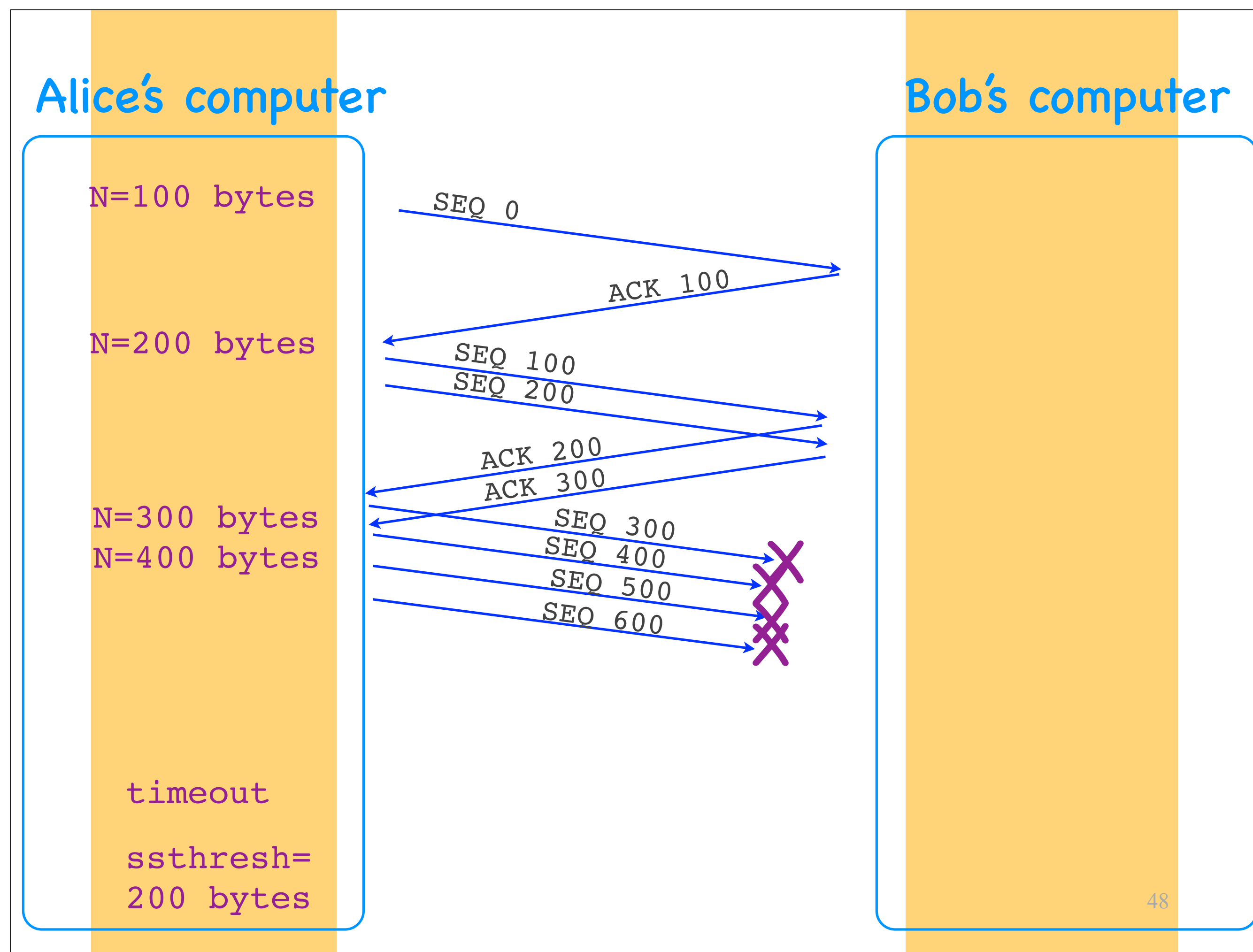
As a result, Alice **doubles** the size of her congestion window every RTT: after 1 RTT the window size is 200 bytes; after 2RTTs, it is 400 bytes.

Differently said, Alice increases the size of her congestion window exponentially as a function of time.

Increase window size

- Exponentially
  - by 1 MSS for every ACKed segment
  - = window doubles every RTT
  - when we do not expect congestion

So: When the sender has no reason to expect congestion, it increases the congestion window size by 1MSS with every ACK-ed segment, and this results in doubling the congestion window every RTT.

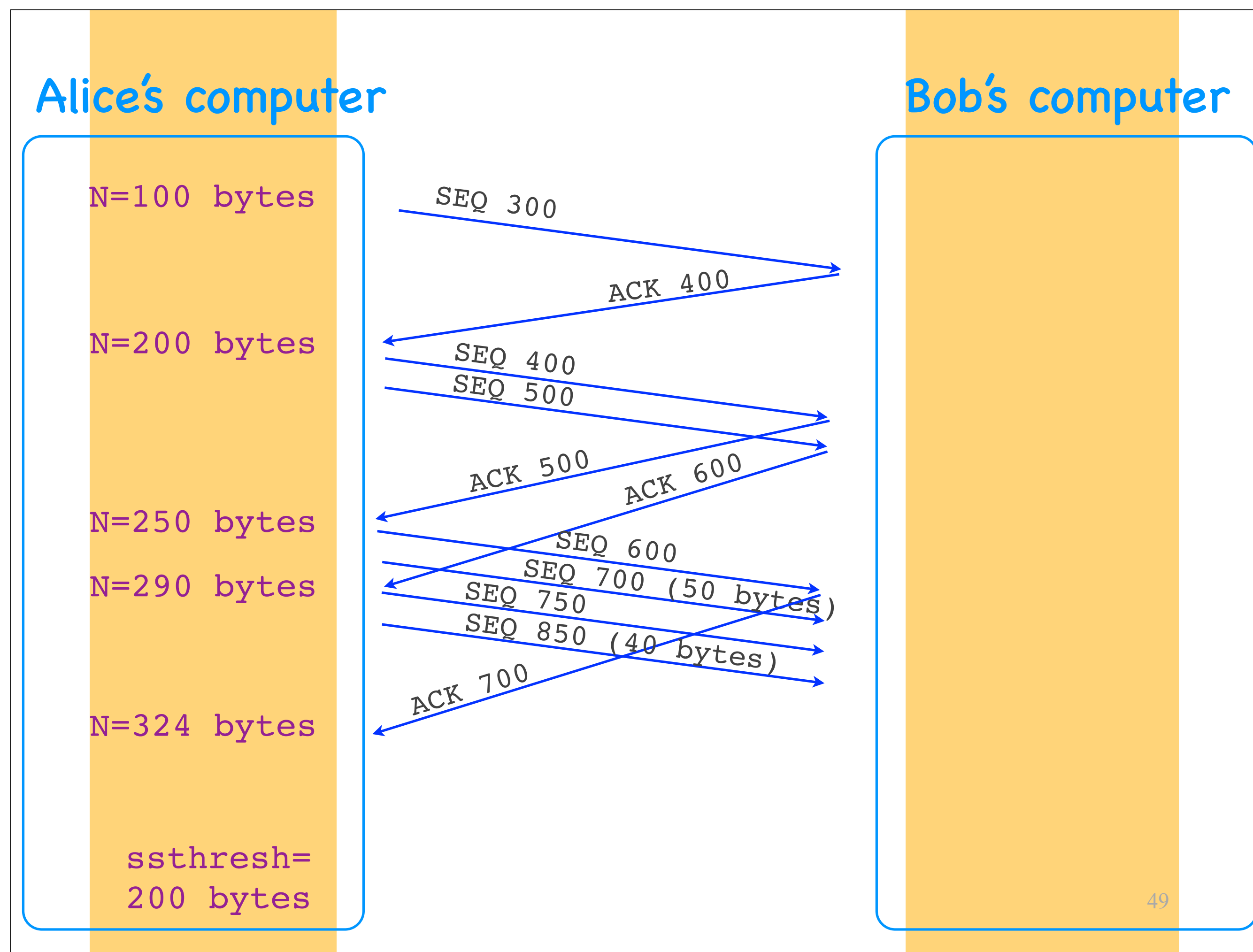


We are at the point where Alice has increased her congestion window to 400 bytes.

Alice sends 4 segments, carrying a total of 400 bytes.
Suppose they are all lost.

In this case:

Alice times out waiting for a new ACK.
She sets her "congestion threshold" (ssthresh) to half the current window size.
She resets her congestion window to 1 MSS.
And she starts from the beginning (see next slide)…



Alice retransmits the oldest un-ACK-ed segment (SEQ 300).

When she receives ACK 400, she increases her congestion window by 1 MSS.
So, she sends 200 bytes (SEQs 400 and 500).

When she receives ACK 500,
according to what we have said so far,
we would expect her to increase her congestion window by 1 MSS (i.e., to 300 bytes) and send 200 more bytes.
But is this a good idea?
No, because last time she increased her congestion window beyond 200 bytes, there was packet loss, and this is likely to happen again.

Instead, Alice becomes more cautious:

When she receives ACK 500, she increases her congestion window to only 250 bytes.
She has already sent 100 un-ACK-ed bytes (SEQ 500),
so she sends 150 more bytes (SEQs 600 and 700).

When she receives ACK 600, she increases her congestion window to only 290 bytes.
She has already sent 150 un-ACK-ed bytes (SEQs 600 and 700),
so she sends 140 more bytes (SEQs 750 and 850).

When she receives ACK 700, she increases her congestion window to 324 bytes. And so on.

You are wondering how exactly these congestion window sizes are chosen, and I will tell you precisely in a moment.

Increase window size

- Exponentially
  - by 1 MSS for every ACKed segment
  - = window doubles every RTT
  - when we do not expect congestion
- Linearly
  - by MSS*MSS/N for every ACKed segment
  - = by 1 MSS every RTT
  - when we expect congestion

So:

TCP has 2 ways of increasing the congestion window:

- When TCP does not expect congestion,
it increases the congestion window by 1 MSS every time a new ACK is received,
which results in doubling the congestion window every RTT.
We call this an exponential increase, because the congestion window increases exponentially over time.

- When TCP does expect congestion,
it increases the congestion window by MSS*MSS/N, where N is the current congestion window size, every time a new ACK is received,
which results in increasing the congestion window by roughly 1 MSS every RTT.
We call this a linear increase, because the congestion window increases linearly over time.

-> Where does this weird formula, MSS*MSS/N, come from?

Goal: increase N by MSS bytes per RTT

Alice sends N unack-ed bytes per RTT

$$= \frac{N}{MSS} \text{ data segments per RTT}$$

She expects $\frac{N}{MSS}$ ACKs per RTT

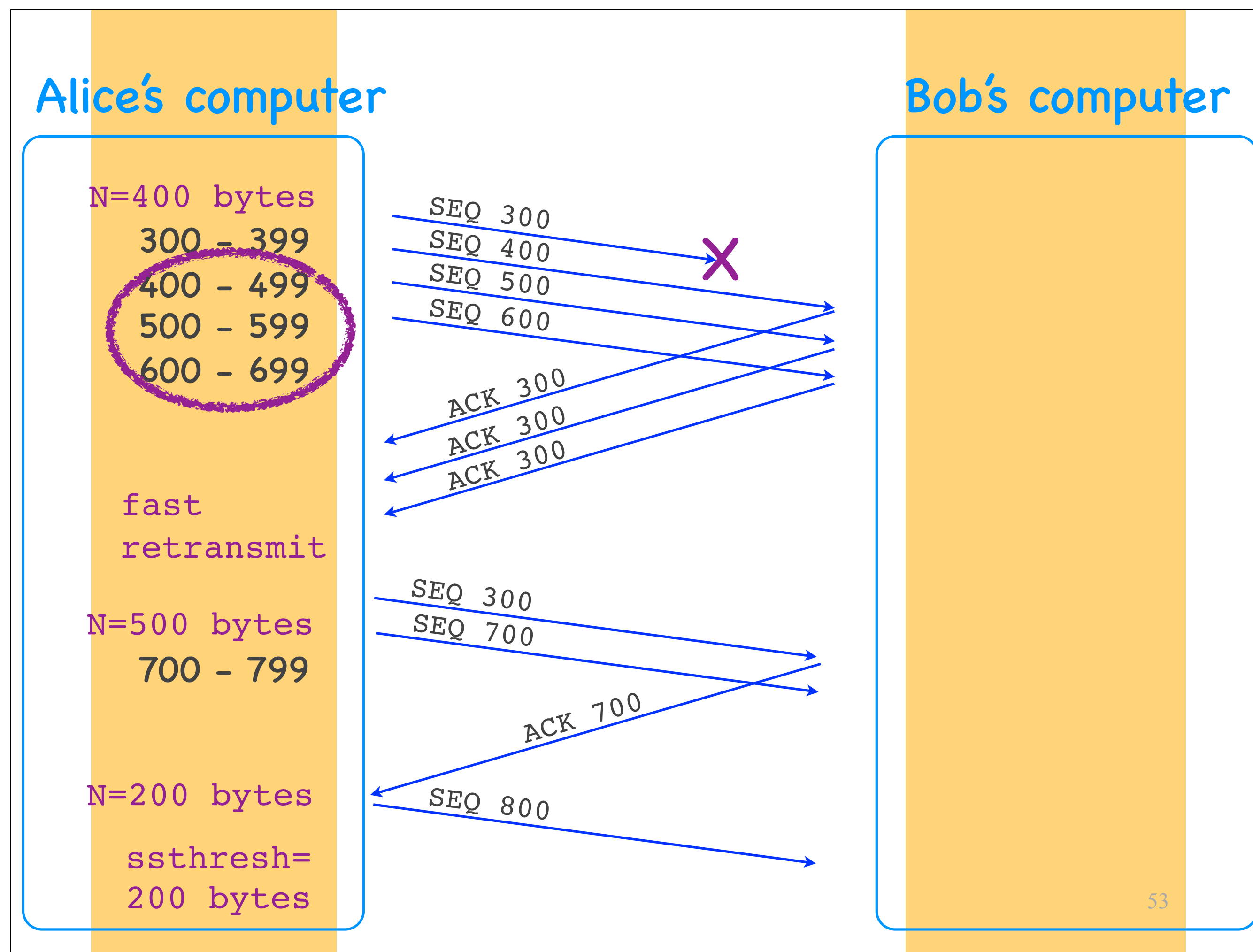
$$\frac{N}{MSS} * \frac{MSS*MSS}{N} \text{ bytes } = MSS \text{ bytes}$$

Basic algorithm (Tahoe)

- Set window to 1 MSS,
  increase exponentially
- On timeout, reset window to 1 MSS,
  set ssthresh to last window/2
- On reaching ssthresh,
  transition to linear increase

This is the basic idea behind the TCP “Tahoe” congestion-control algorithm…

In other words: TCP congestion control acts like a child who wants chocolate: it starts by asking for a little bit, then increasingly asks for more, until it approaches the limit where her parents previously refused to give her more; as she approaches the limit more and more she asks more and more cautiously, to see if she could maybe exceed the limit this time.



We are back at the point where Alice has increased her congestion window to 400 bytes.

Alice sends 4 segments, carrying a total of 400 bytes.
Suppose the first one gets lost.

In this case:

Alice receives 3 duplicate ACKs (3x ACK 300).
This makes her suspect that the segment with SEQ 300 was lost,
while the next 3 segments were received
(it's a very good explanation of why Bob would sent 3x ACK 300).

Given this suspicion, Alice does a "fast retransmit": she retransmits the segment she suspects lost (SEQ 300) **without waiting for a timeout**.

Moreover, she sets ssthresh to half her congestion window
and reduces her congestion window.
However, she does not reset it to 1 MSS
(this would be an overreaction, given that only 1 segment was lost).
Instead, she wants to set it to her congestion threshold (which is 200 bytes, in our example), since this was the last congestion window for which Alice did not experience any packet loss.

However, there is a problem:

Imagine, for a moment, that Alice sets her congestion window to 200 bytes,
which means that she can send 200 un-ACK-ed bytes.
She has already sent 400 un-ACK-ed bytes (SEQs 300 — 600).
Hence, she has already exceeded her congestion window,
which means that she can send 0 bytes.

The problem is the following:

- The segments with SEQs 400, 500, and 600 are officially un-ACK-ed.
- However, Alice believes that Bob received them because of his 3 duplicate ACKs. So, Alice does not want to treat these segments as un-ACK-ed.

The solution to this problem is for Alice to “inflate” her congestion window by 3 MSS’s (which is 300 bytes, in our example).
So, Alice sets her congestion window to 500 bytes,
which means that she can now send 500 un-ACK-ed bytes.
She has already sent 400 un-ACK-ed bytes (SEQs 300 — 600),
so she can send 100 more bytes (SEQ 700).

Then, Alice enters a “fast recovery” phase:

Every time she receives a new duplicate ACK (another ACK 300),
she increases her congestion window by 1 MSS.

The moment she receives a new ACK (ACK 700, in our example),
she concludes that Bob finally received the lost segment,
she exits fast recovery,
she sets her window to her congestion threshold (200 bytes, in our example),
and (since her congestion window is now equal to her congestion threshold)
she transitions to linear increase.

Alice's computer

Bob's computer

N=500 bytes

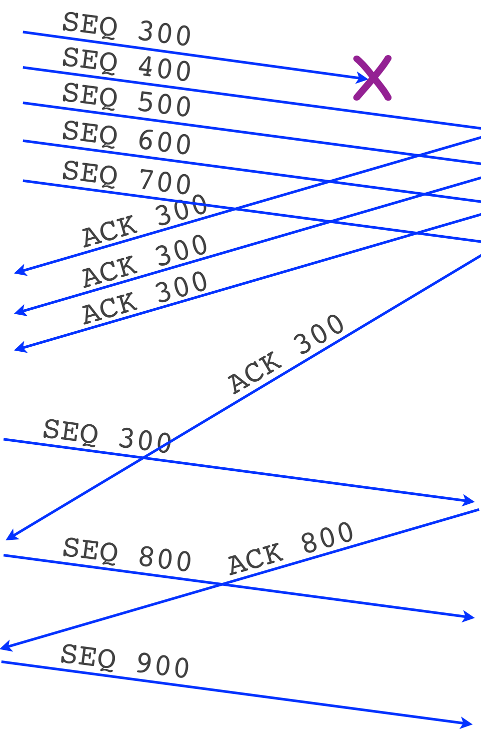
300 - 399

400 - 499

500 - 599

600 - 699

700 - 799



fast retransmit

N=500 bytes

N=600 bytes

N=200 bytes

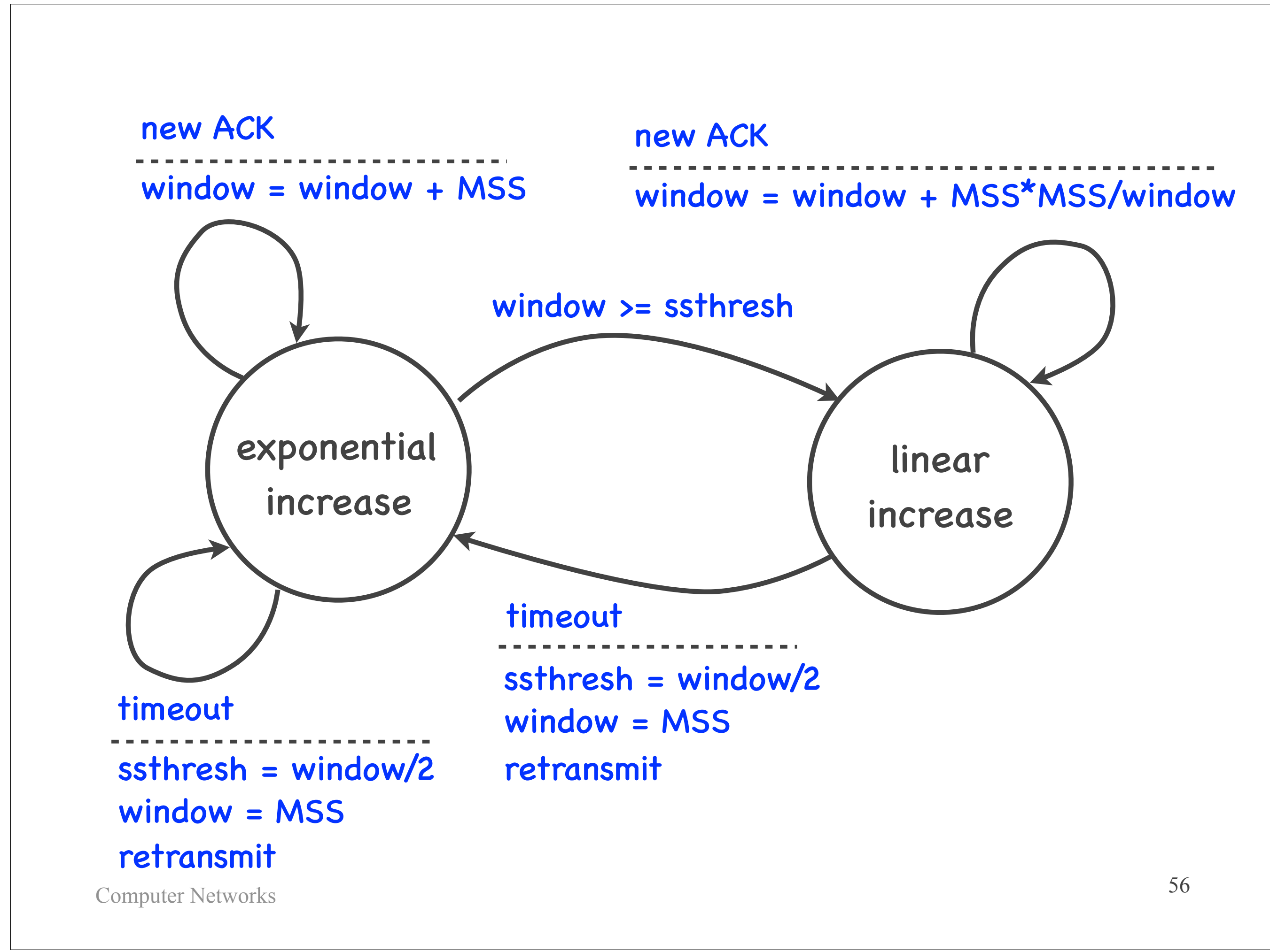
ssthresh= 200 bytes

Basic algorithm (Reno)

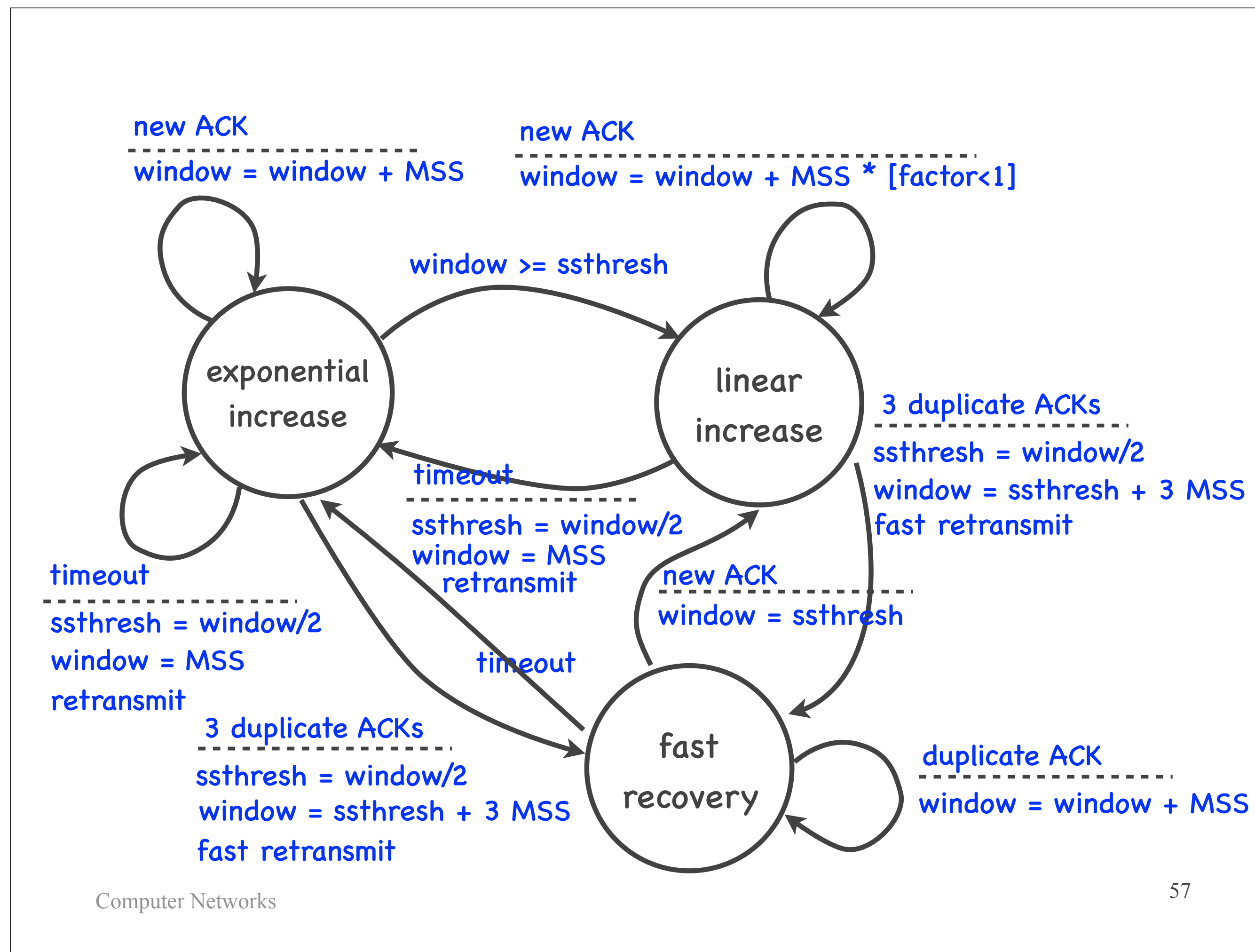
- Set window to 1 MSS, increase exponentially
- On timeout, reset window to 1 MSS,
  set ssthresh to last window/2, retransmit
- On receiving 3 duplicate ACKs,
  set window to ssthresh (+inflation), retransmit
- On reaching ssthresh transition
  to linear increase

This is the basic idea behind the TCP “Reno” congestion-control algorithm, which is more recent than Tahoe, and which is essentially Tahoe + the fast retransmit mechanism and the fast recovery phase.

This is not the latest TCP congestion-control algorithm. There is at least New Reno and Cubic, which are used by state-of-the-art operating systems. We will not examine them in this course, but you know everything you need to to explore them on your own.



The Tahoe state machine.



The Reno state machine.

Clarification: When TCP is on the fast recovery state and there is a timeout, it sets ssthresh to window/2, the window to MSS, and retransmits, before transitioning to exponential increase. But I did not have room to show all this.

TCP terminology

- Exponential increase = slow start
  - it's called slow, because it starts from a small window; but it's not really slow, the window increases exponentially
- Linear increase = congestion avoidance
  - this term does make sense; it means that TCP expects congestion, so it increases the window more cautiously

A word of caution about TCP terminology: …

Flow + congestion control

- Goal: not overwhelm receiver or network
- How: sender window
  - sender learns receiver window from receiver
  - sender computes congestion window on its own
  - Sender window = min{ receiver w, congestion w }

In the end, a TCP sender learns the receiver window (from the receiver), computes a congestion window (on his own), and sets its sender window to the smaller of the two.

-> Why bother with TCP details, like inflation window etc?