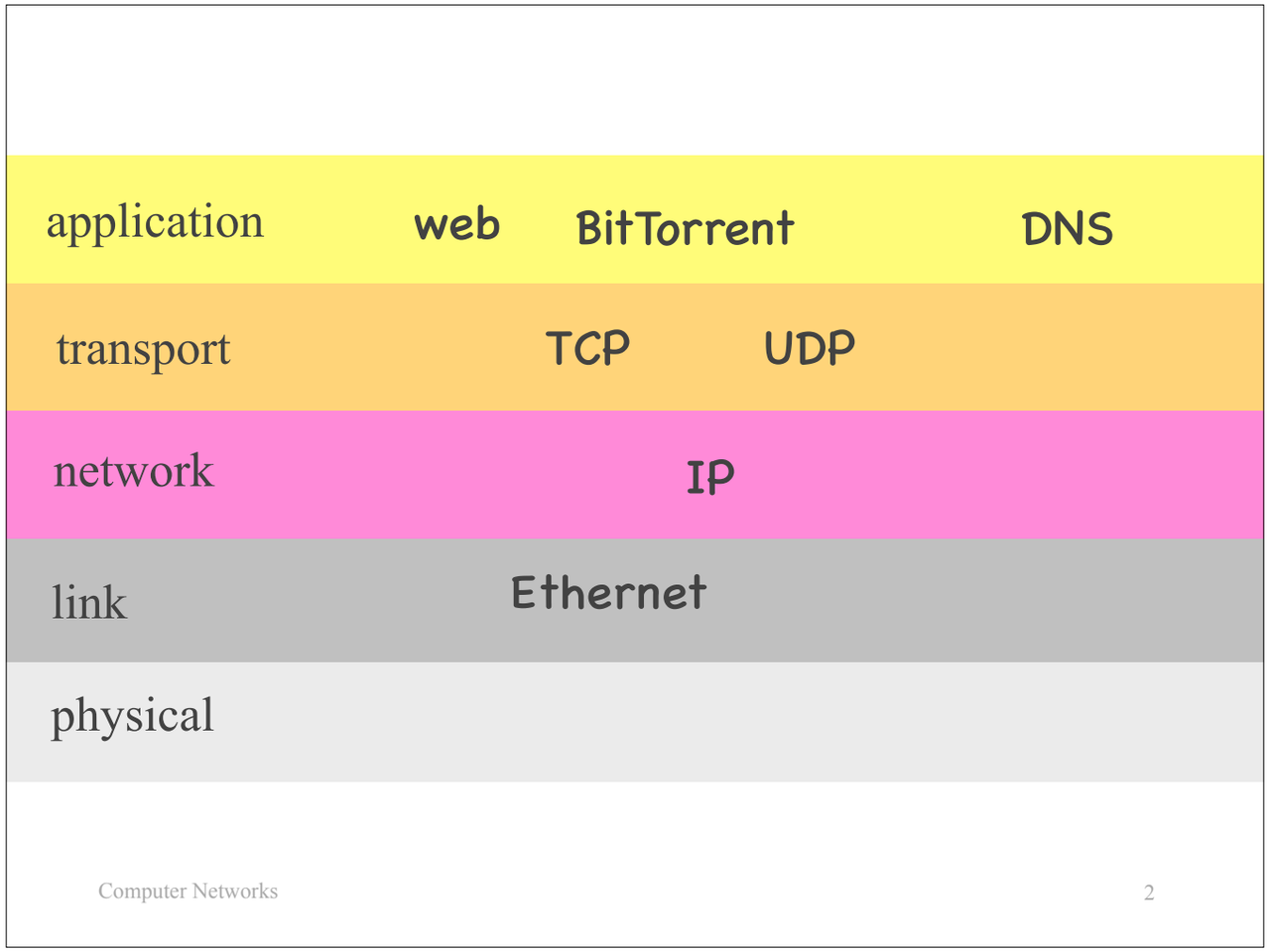
Lecture 5:

The Transport Layer

Katerina Argyraki, EPFL

Welcome to the 5th lecture of Computer Networks, where we will start exploring the transport layer.



Remember that the transport layer is sandwiched between the application layer and the network layer.
So, even though we will focus on the transport layer, we will need to involve the application and network layers in our discussion.

Outline

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

We will discuss two specific aspects of the transport layer:

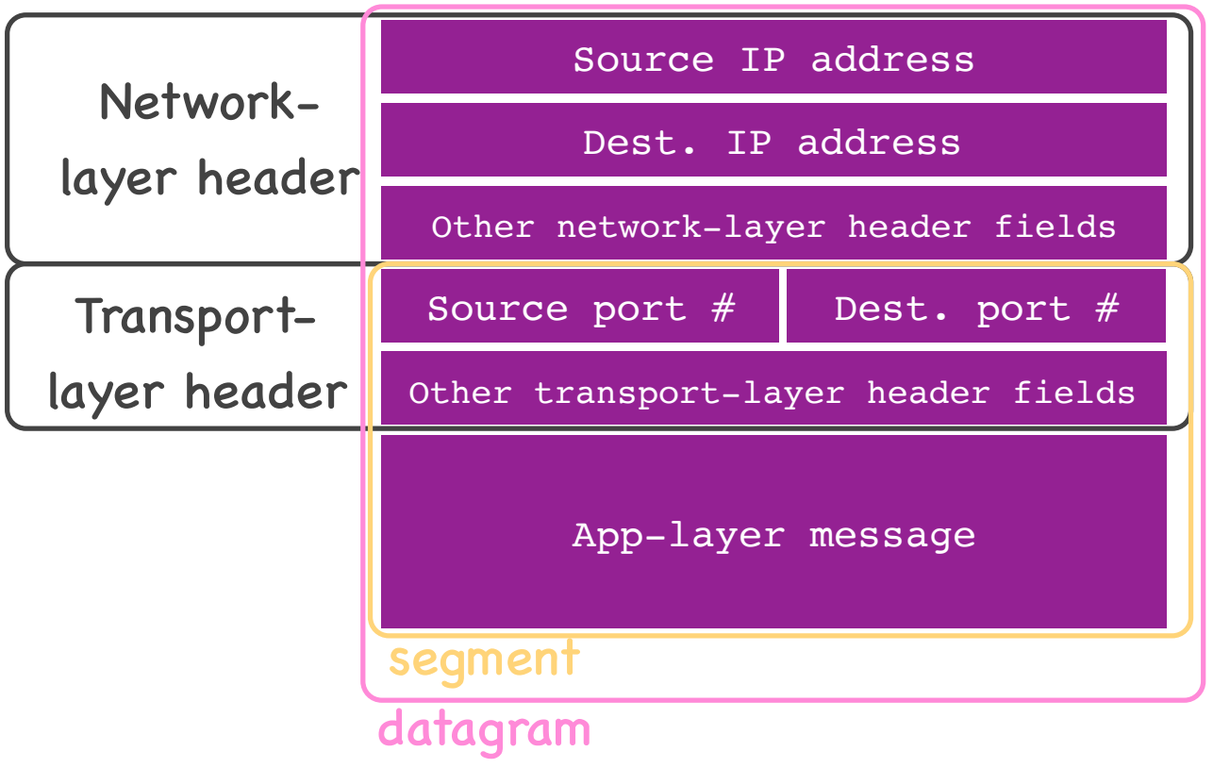
The first one is how the transport layer interacts with the application layer.
We will see how UDP and how TCP does it.

The second aspect is a specific service that the transport layer typically offers to the application layer: reliable data delivery.
We will NOT examine how TCP offers reliable data delivery in this lecture.
For educational purposes, we will construct an imaginary protocol, step by step.

Outline

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

Let’s start with how the transport layer interact with the application layer.



Let's look at the structure of a typical Internet packet.
You have gotten a sense of this structure through the labs and the Wireshark took,
but let's look at it more carefully now.

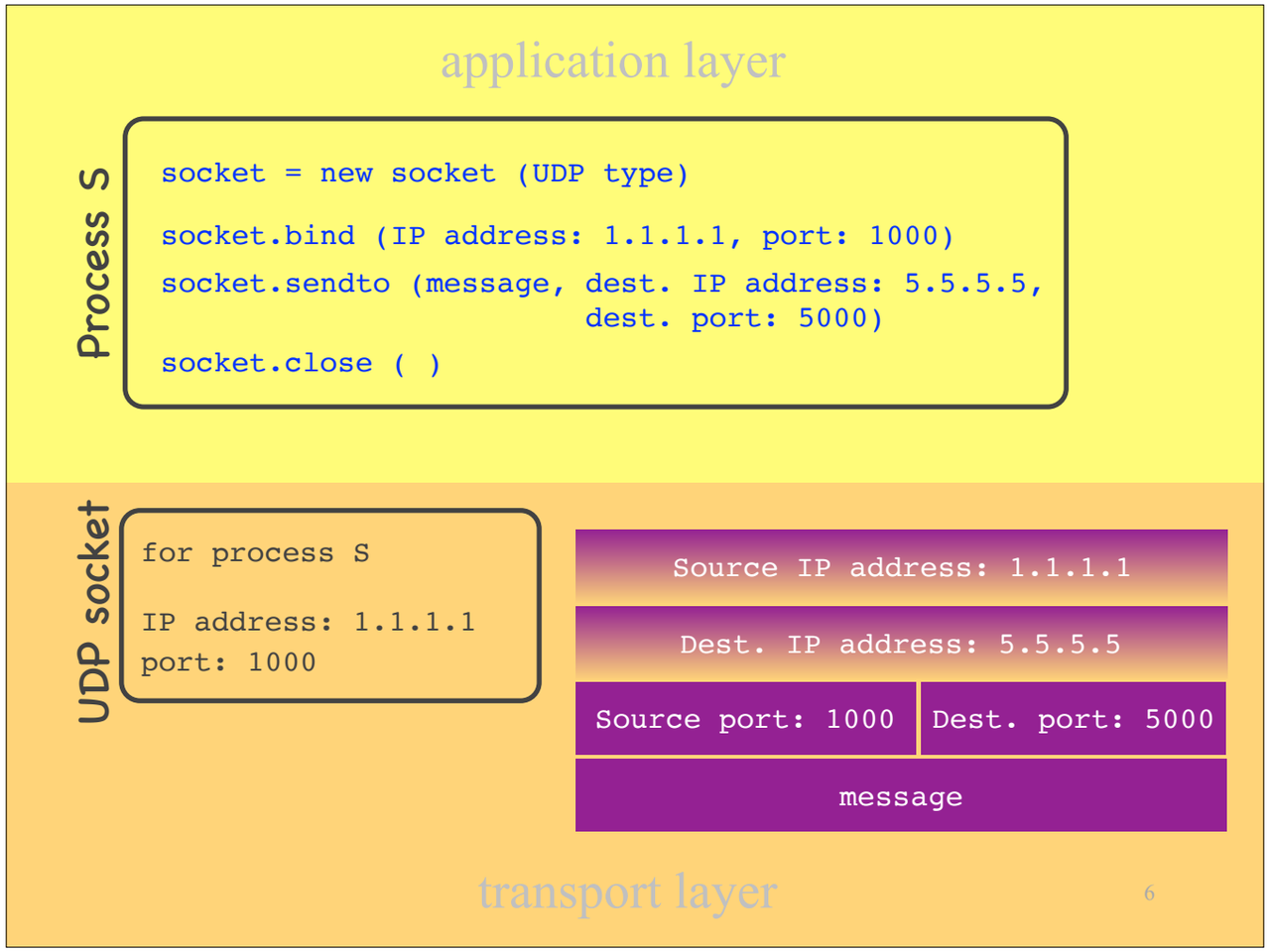
First a process creates a message,
then the transport layer adds a transport-layer header,
and then the network layer adds a network-layer header.

Of the various header fields that the transport and network layer add,
today we will focus on the source and destination IP addresses and the source and destination port numbers.
These fields together specify a packet's source and destination end-systems and processes.

Let's also introduce two new terms:

- A "segment" is the part of a packet that includes the app-layer message and the transport-layer header.
- A "datagram" is the part of a packet that includes the app-layer message, the transport-layer header, and the network-layer header.

So, a packet typically includes a datagram, which typically includes a segment.



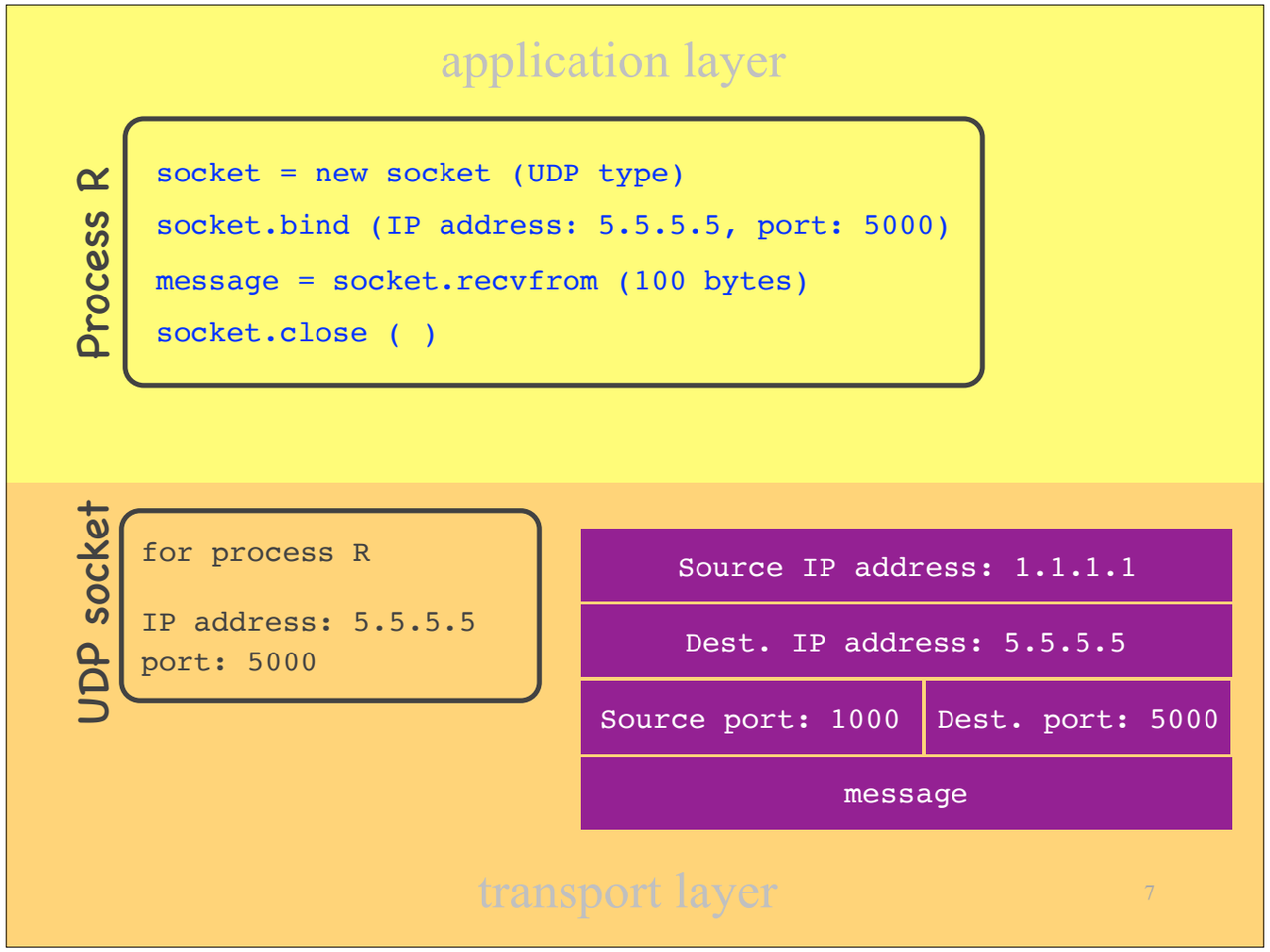
Suppose a process S wants to use UDP as its transport-layer protocol to send a message to a remote process:

- First, the process asks the transport layer to open a UDP socket. In response, the transport layer creates a UDP socket and associates it with this process. So, a “socket” is simply a data structure that lives in the transport layer and is associated with a particular local process.

- Second, the process asks the transport layer to bind the socket to a particular (local) IP address and port number. In response, the transport layer adds this information to the socket.

- At this point, the process is ready to send a message through this socket. To do this, it calls a function and provides as arguments a pointer or reference to the message, the destination IP address, and the destination port number. In response, the transport layer starts putting together a packet: the message that the process is sending, the destination IP address and port number that the process passed as arguments through the sendto function, and the source IP address and port number that are associated with this socket.
  - To be precise, the transport layer creates only the transport-layer header (which contains the source and destination port numbers, not the source and destination IP addresses), but it still keeps track of the source and destination IP addresses, because it needs to provide them to the network layer, which will create the network-layer header.

- If the process does not need the socket any more, it asks the transport layer to close it. In response, the transport layer deletes the socket.



Now consider the receiving end: A process R wants to use UDP to receive a message from a remote process:

- First, the process asks the transport layer to open a UDP socket. In response, the transport layer creates a UDP socket and associates it with this process.

- Second, the process asks the transport layer to bind the socket to a particular (local) IP address and port number. In response, the transport layer adds this information to the socket.

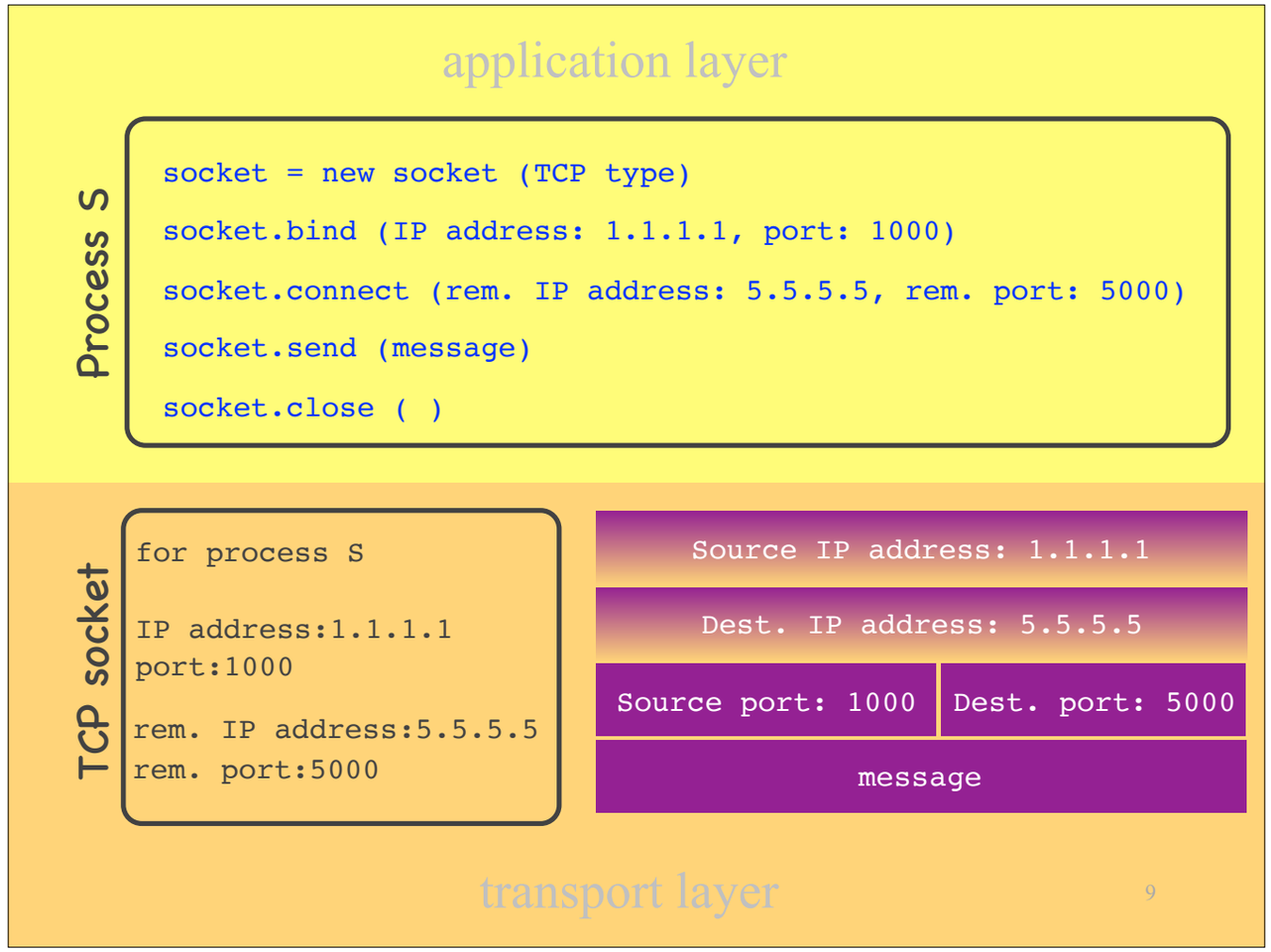
- At this point, the process is ready to receive a message through this socket. To do this, it calls a function and provides as argument the number of bytes that it wants to receive.

- If a packet arrives from the network layer, the transport layer reads the headers and identifies which of the many processes that are running in the local application layer this packet is meant for. In this particular example, the transport layer will read from the packet's headers that the destination IP address is 5.5.5.5 and the destination port number is 5000. Then it will ask: "Do I have a socket with this IP address and port number? Yes, I do, for process R", so it will pass 100 bytes from the message to this process.

- If the process does not need the socket any more, it asks the transport layer to close it. In response, the transport layer deletes the socket.

UDP sockets

- Each UDP socket has a unique (IP address, port #) tuple
- A process may use the same UDP socket to communicate with many remote processes



Now suppose a process S wants to use TCP as its transport-layer protocol to send a message to another process.

The first two steps are similar to the UDP scenario:

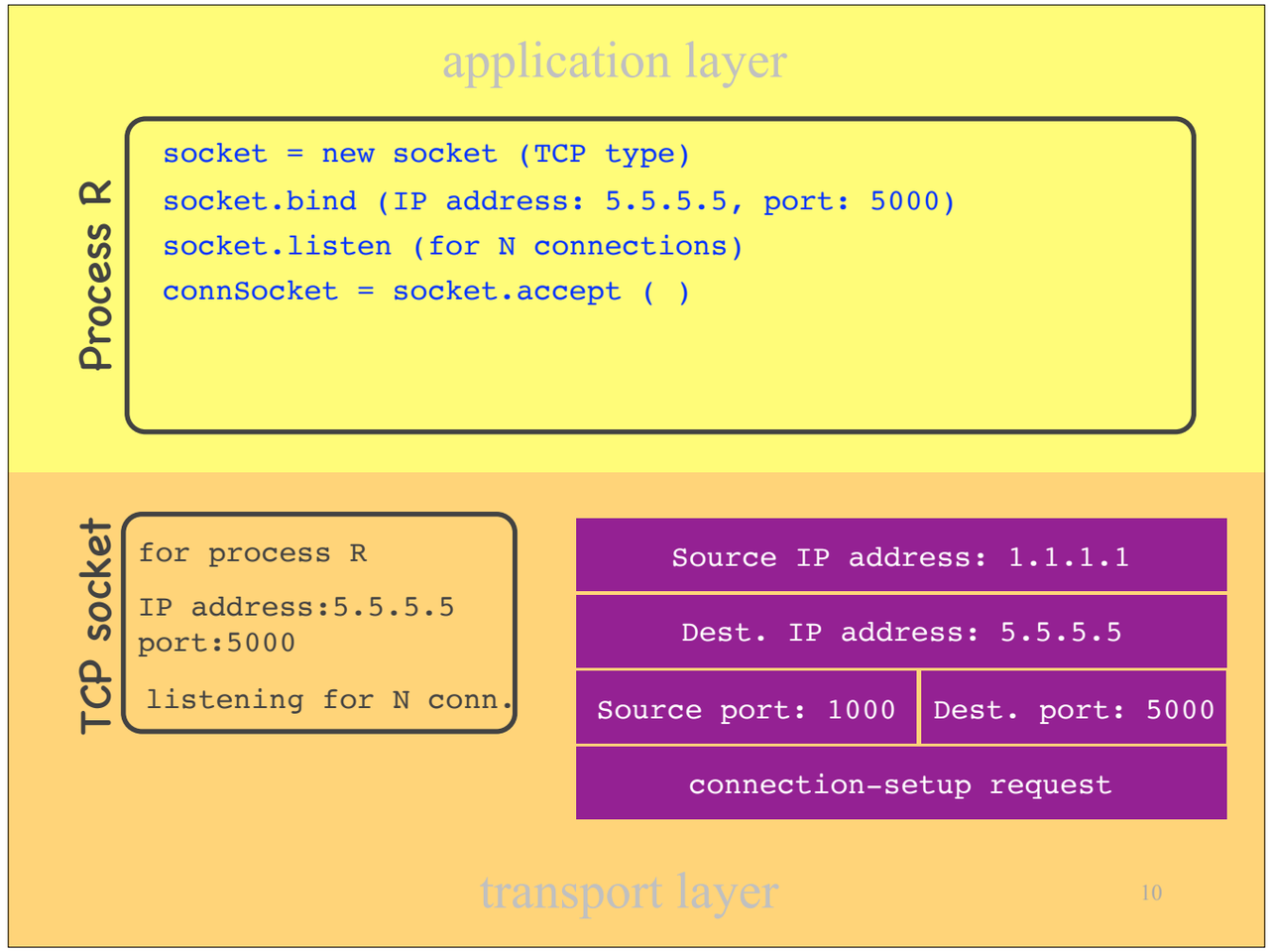
- First, the process asks the transport layer to open a TCP socket. In response, the transport layer creates a TCP socket and associates it with this process.
- Second, the process asks the transport layer to bind the socket to a particular (local) IP address and port number. In response, the transport layer adds this information to the socket.

The next steps differ. Unlike UDP, a TCP sender establishes a connection to a TCP receiver:

- The process asks the transport layer to connect the socket to a particular remote IP address and port number. In response, the transport layer adds this information to the socket and sends a TCP connection setup packet to the remote process.

- If the connect function returns successfully, the process is ready to send a message through this socket. For this, it calls a function called “send” and provides as argument a pointer or reference to the message. In response, the transport layer starts putting together a packet with all the necessary information, including source and destination IP addresses and port numbers.
  - Unlike with UDP, the process does not provide the destination IP address and destination port number as arguments to the send function, because the socket is already connected to a specific remote IP address and port number.

- If the process does not need the socket any more, it asks the transport layer to close it. In response, the transport layer deletes the socket.



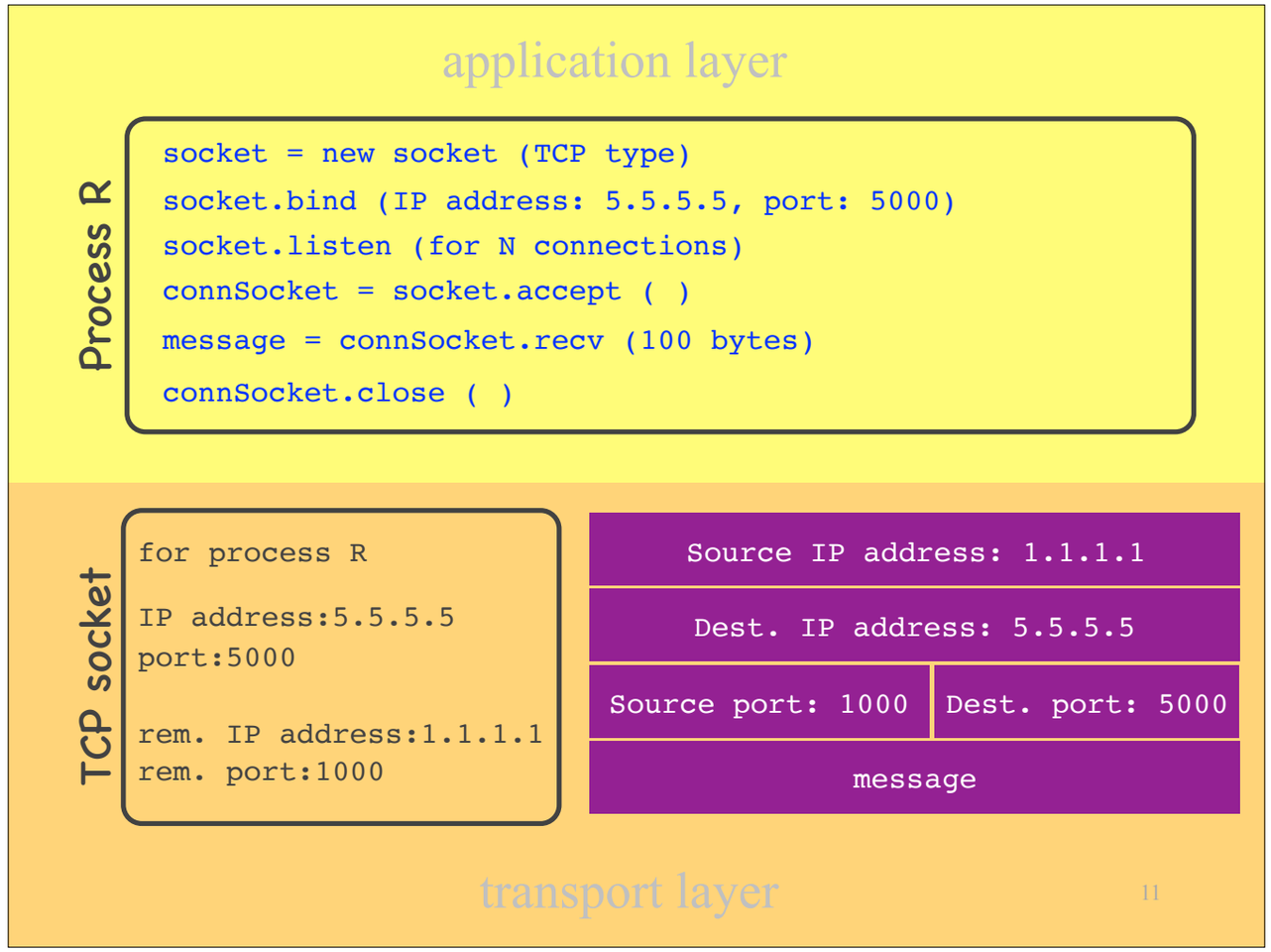
Now consider the receiving end: A process R wants to use TCP to receive a message from another process:

Again, the first two steps are similar to the UDP scenario:

- First, the process asks the transport layer to open a TCP socket. In response, the transport layer creates a TCP socket and associates it with this process.
- Second, the process asks the transport layer to bind the socket to a particular (local) IP address and port number. In response, the transport layer adds this information to the socket.

The next steps differs:

- The process tells the transport layer that it wants to use this socket to listen for TCP connection-setup requests from remote processes.
- When a new TCP connection-setup request arrives, the transport layer reads the destination IP address and destination port number and sees that there exists a TCP socket listening for TCP connection-setup requests at this IP address and port number. Hence, it passes the request to process R.
- If the process chooses to accept the request, it says so to the transport layer. In response, the transport layer…



…creates a new TCP “connection socket,” which is connected to the specific remote IP address and port number.

- At this point, the process is ready to receive a message through this socket. For this, it calls a function called “recv” and provides as argument the number of bytes that it wants to receive.

- If a packet arrives from the network layer, the transport layer reads the headers and identifies which of all the processes running at the local application layer this packet is meant for. In this particular example, the transport layer will read from the packet’s headers that the source IP address is 1.1.1.1, the destination IP address is 5.5.5.5, the source port number is 1000, and the destination port number is 5000. Then it will ask: “Do I have a socket with these IP addresses and port numbers? Yes, I do, for process R”, so it will pass 100 bytes from the message to this process.

- If the process does not need the socket any more, it asks the transport layer to close it. In response, the transport layer deletes the socket.

TCP sockets

- Listening & connection sockets
- Each connection socket has a unique (local IP, local port, remote IP, remote port) tuple
- A process must use a different TCP connection socket per remote process

Interaction with application layer

- Multiplexing
  - upon receiving a new message from a process, create new packets
  - identify the correct IP addresses and ports
- Demultiplexing
  - many processes running in app layer
  - upon receiving a new packet from the network, identify the correct dest. process

So, this is how the transport layer interacts with the application layer:

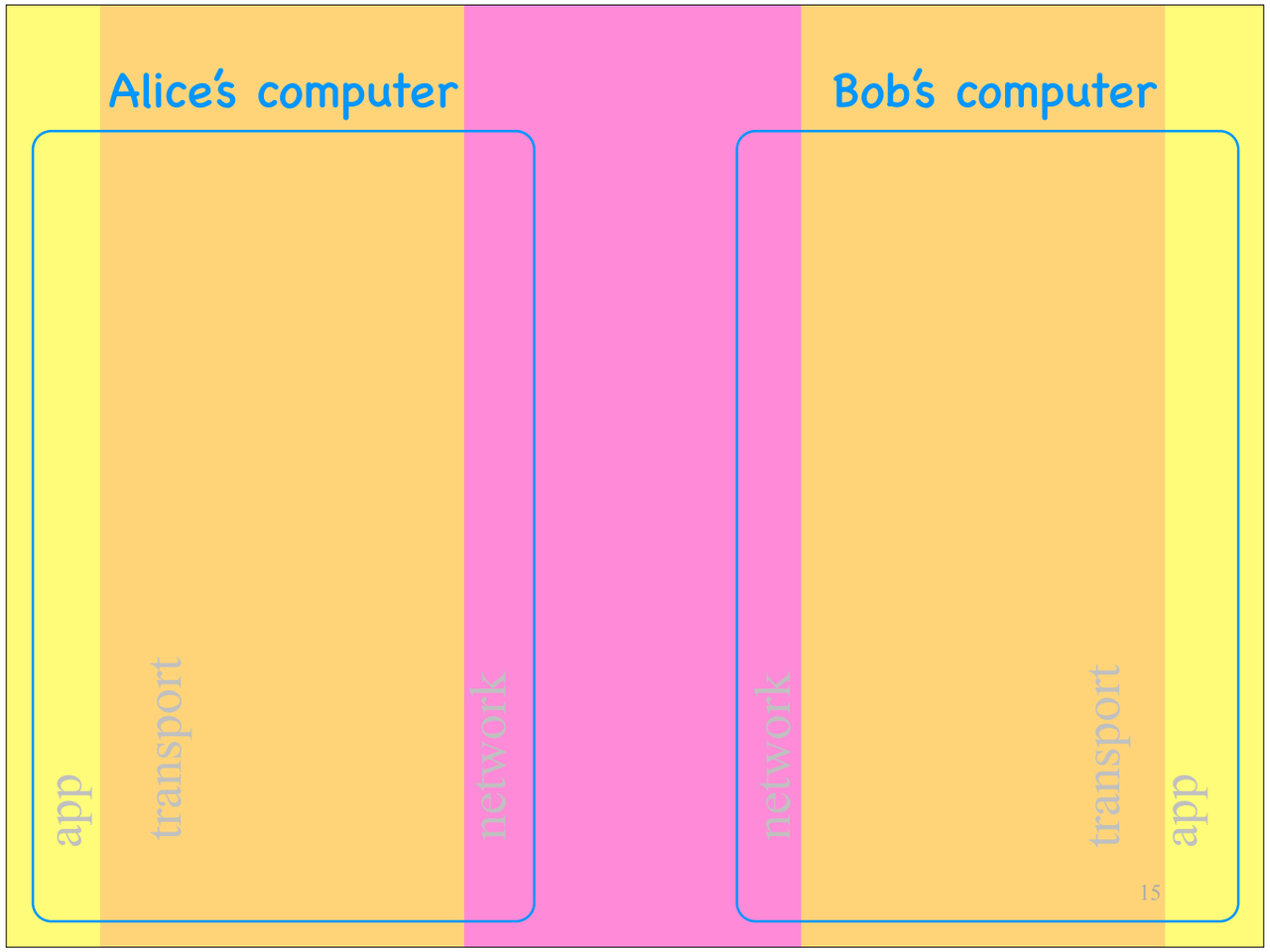
- At the sender side, we have “multiplexing”: the transport layer receives messages from many processes, through many sockets, and it “multiplexes” them, i.e, encapsulates them into segments which are sent over the same network.
- At the receiver side, we have “demultiplexing”: upon receiving a segment from the network layer, the transport layer identifies the correct destination process, decapsulates the enclosed message, and passes it up.

Outline

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

Now let's turn to reliable data delivery.

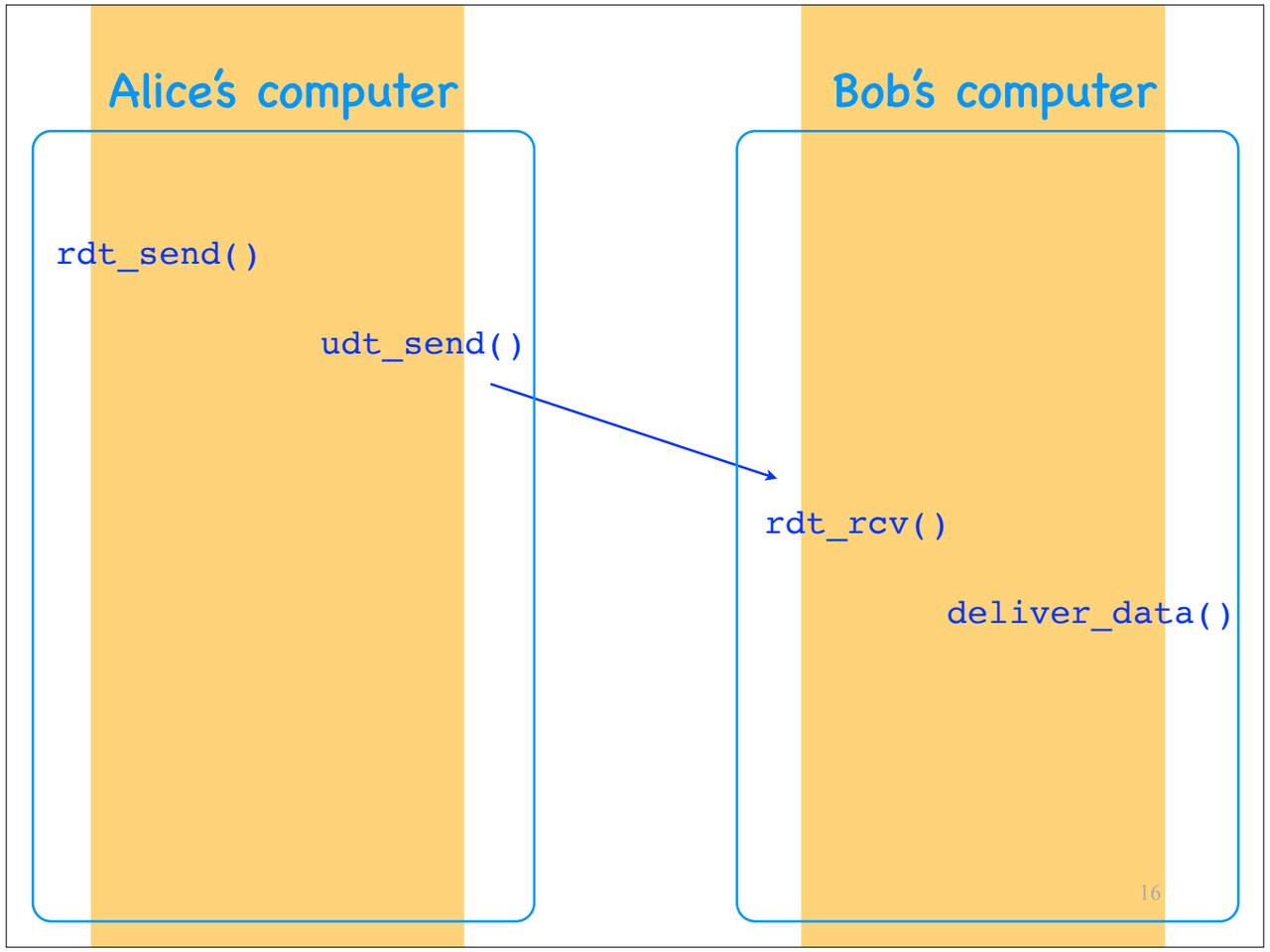
We will first build our own imaginary protocol for reliable data delivery,
which does not correspond to any actual transport-layer technology (it's not TCP),
and we will examine TCP in the next lecture.



In this picture, I have rotated the layers by 90 degrees, so that I can depict two end-systems, a sender and a receiver.
On the left we have Alice's app layer, which talks to Alice's transport layer, which talks to the network, which talks to Bob's transport layer, which talks to Bob's app layer.

The network layer of the Internet is unreliable: it may corrupt or drop packets.

Let's see how we can build a transport layer that provides reliable data transfer on top of an unreliable network. I.e., we want a transport layer that provides to the application layer the illusion of a reliable network.



Consider Alice's and Bob's computers. Suppose a process in Alice's application layer wants to send a message to a process in Bob's application layer. Suppose the message fits in a single segment.

Here is the minimum that happens:

- Alice's app layer calls a function, called rdt_send (reliable-data-transfer_send), to pass the message to the transport layer.
- Alice's transport layer calls another function, called udt_send (unreliable-data-transfer_send), to pass the segment to the network layer.
- The segment traverses the network from Alice to Bob.
- Bob's network layer calls rdt_rcv (reliable-data-transfer_receive) to pass the segment to the transport layer.
- And the transport layer calls deliver_data to pass the message to the app layer.

This transport layer that I just described does nothing but pass data between the application and network layers. It does nothing useful, it might as well not be there.

Now let's start adding useful functionality.

First, the transport layer can detect that a segment was corrupted.
This can be done by using a…

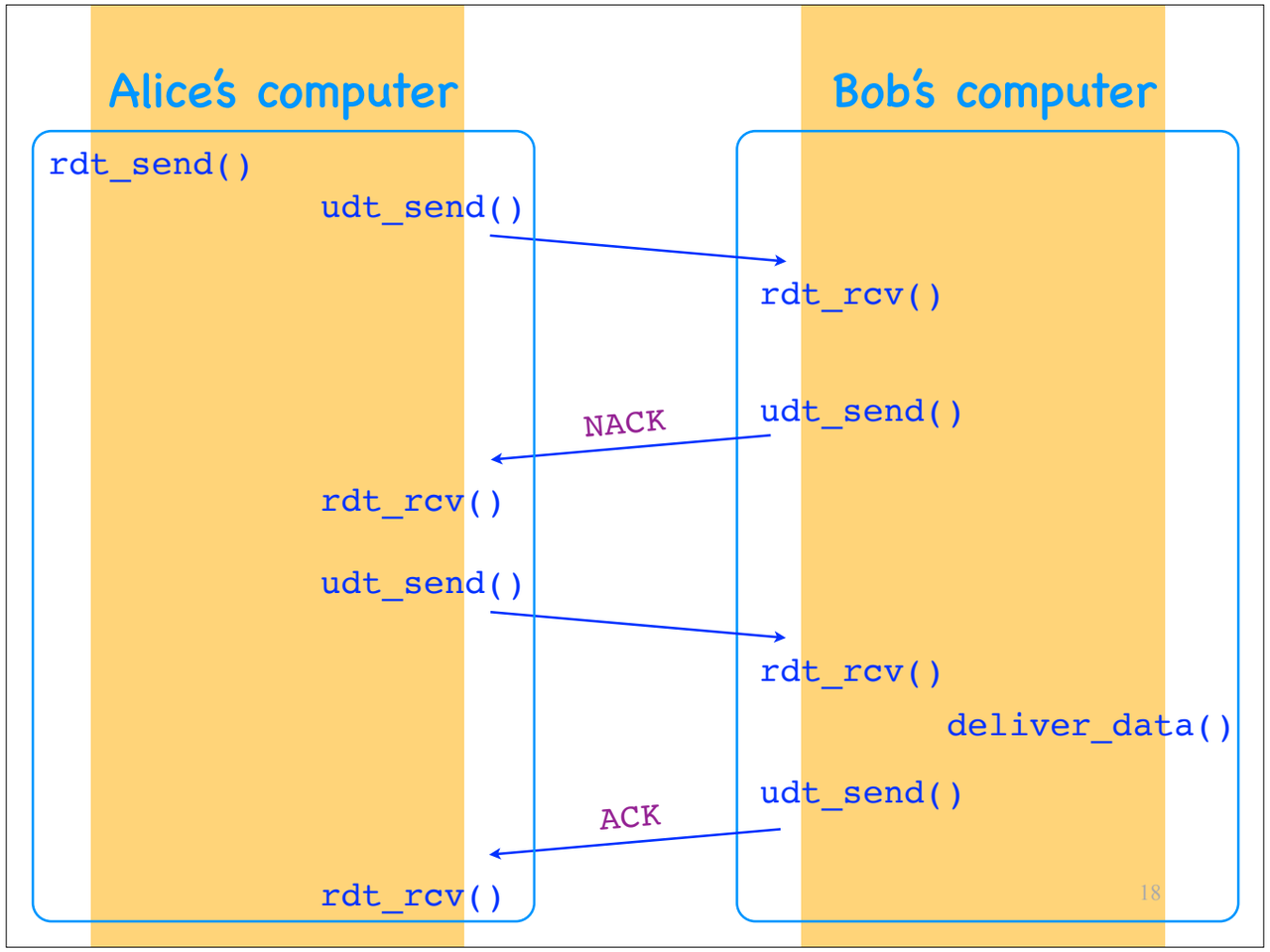
Checksum

- Redundant information
  - e.g., the binary sum of all data bytes
- Sender adds checksum C to each segment
  - transport-layer header field
- Receiver uses it to detect data corruption
  - receiver recomputes checksum C'
  - if C'!= C, segment was corrupted

... checksum,
which is redundant information that depends on the data,
e.g., the sum of all the data bytes.

The sender adds a checksum to each segment.
The receiver recomputes the checksum and determines whether the segment was corrupted or not.

Certain checksums make it possible to not only detect corruption but also correct limited forms of corruption, but we will not discuss this scenario today.



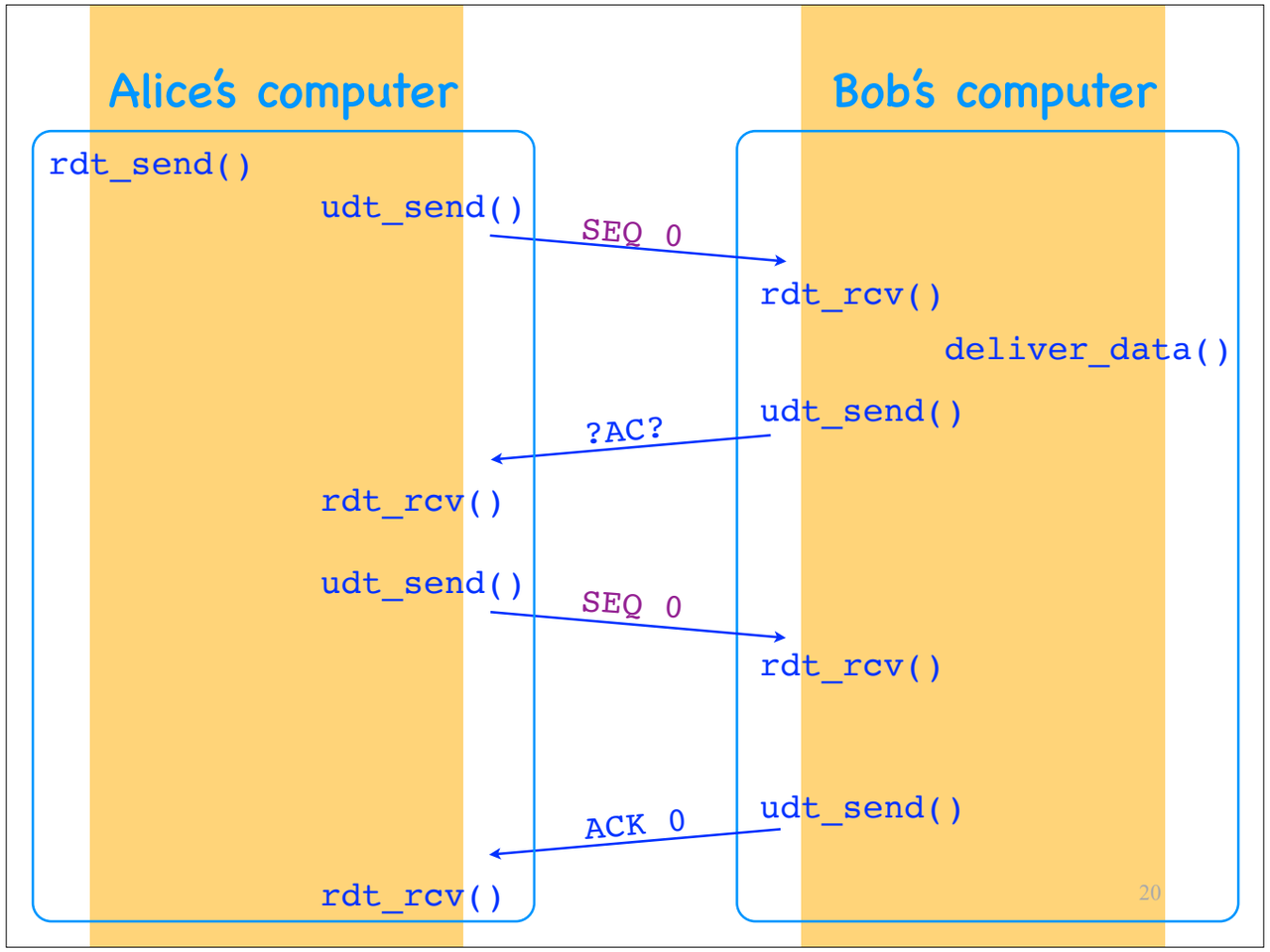
Suppose that Alice sends a segment to Bob, and Bob's transport layer determines (using the checksum) that the segment was corrupted.
What can it do next?

It can send back to Alice's transport layer a negative ACK (NACK).
In response, Alice's transport layer retransmits the same segment.
If the retransmitted segment arrives at its destination uncorrupted, Bob's transport layer extracts the data, delivers it to the application layer, and sends back a positive ACK.

Acknowledgment

- Feedback from receiver to sender
- Receiver adds ACK to each segment
  - transport-layer header field
- Sender uses it to detect and overcome data corruption
  - if sender gets negative ACK, it retransmits the data

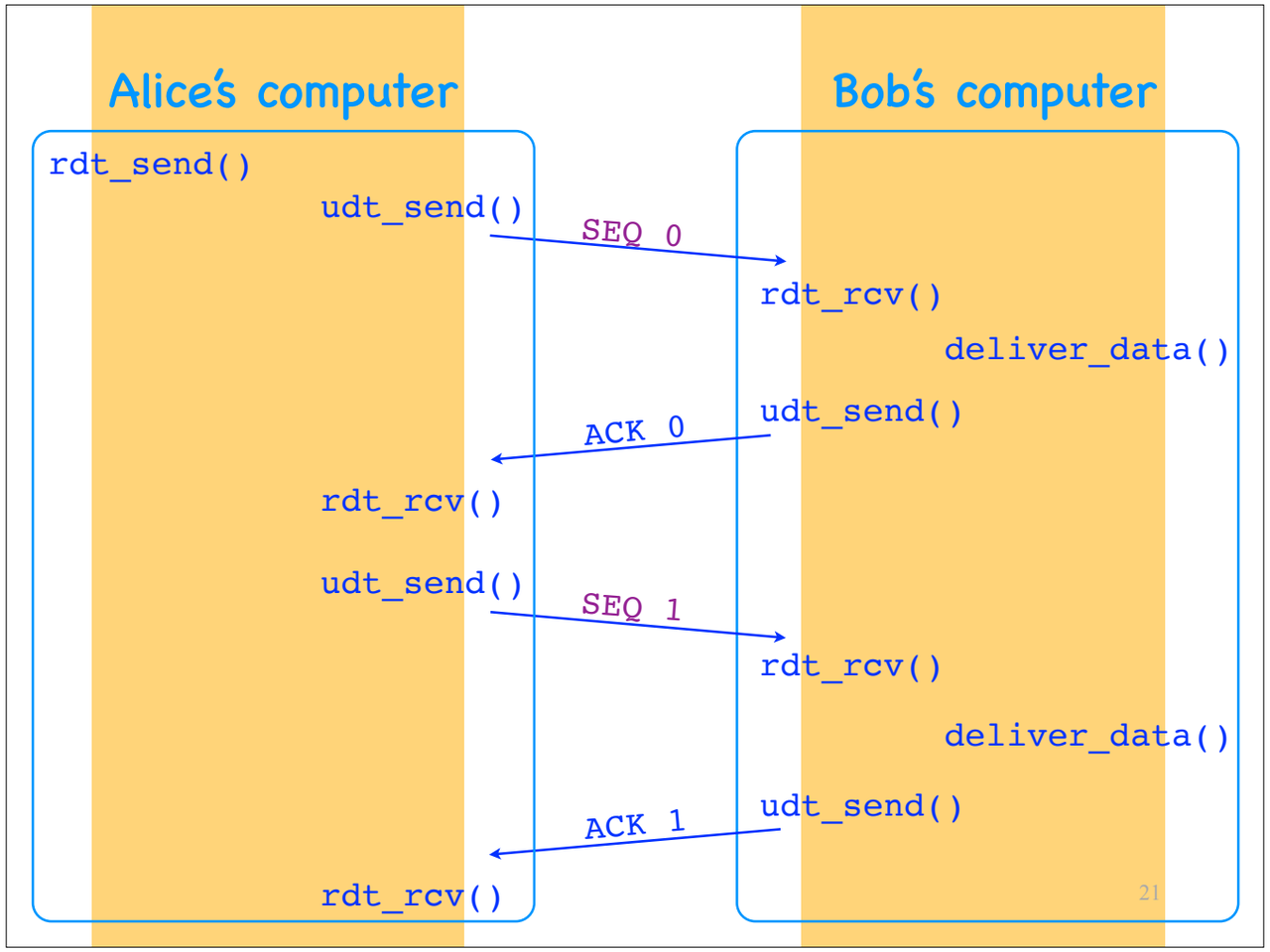
So: An important element of reliable data delivery is the combination of acknowledgments (ACKs) and retransmissions…



Suppose that Alice sends a segment to Bob, and Bob's transport layer sends back a positive ACK. However, the ACK gets corrupted, and Alice's transport layer cannot tell if it's a positive or a negative ACK. What must Alice's transport layer do?

It must be conservative and retransmit the segment.
However, Bob's transport layer (which has already successfully received this segment) may now think that Alice's transport layer is transmitting a new segment, whereas she is *re*transmitting the old one.

To resolve this issue, we need sequence numbers:
- When Alice's transport layer sends a segment, it adds a sequence number to it.
- If it retransmits the same segment, that segment has the same sequence number,
so Bob knows that Alice is retransmitting the same segment.



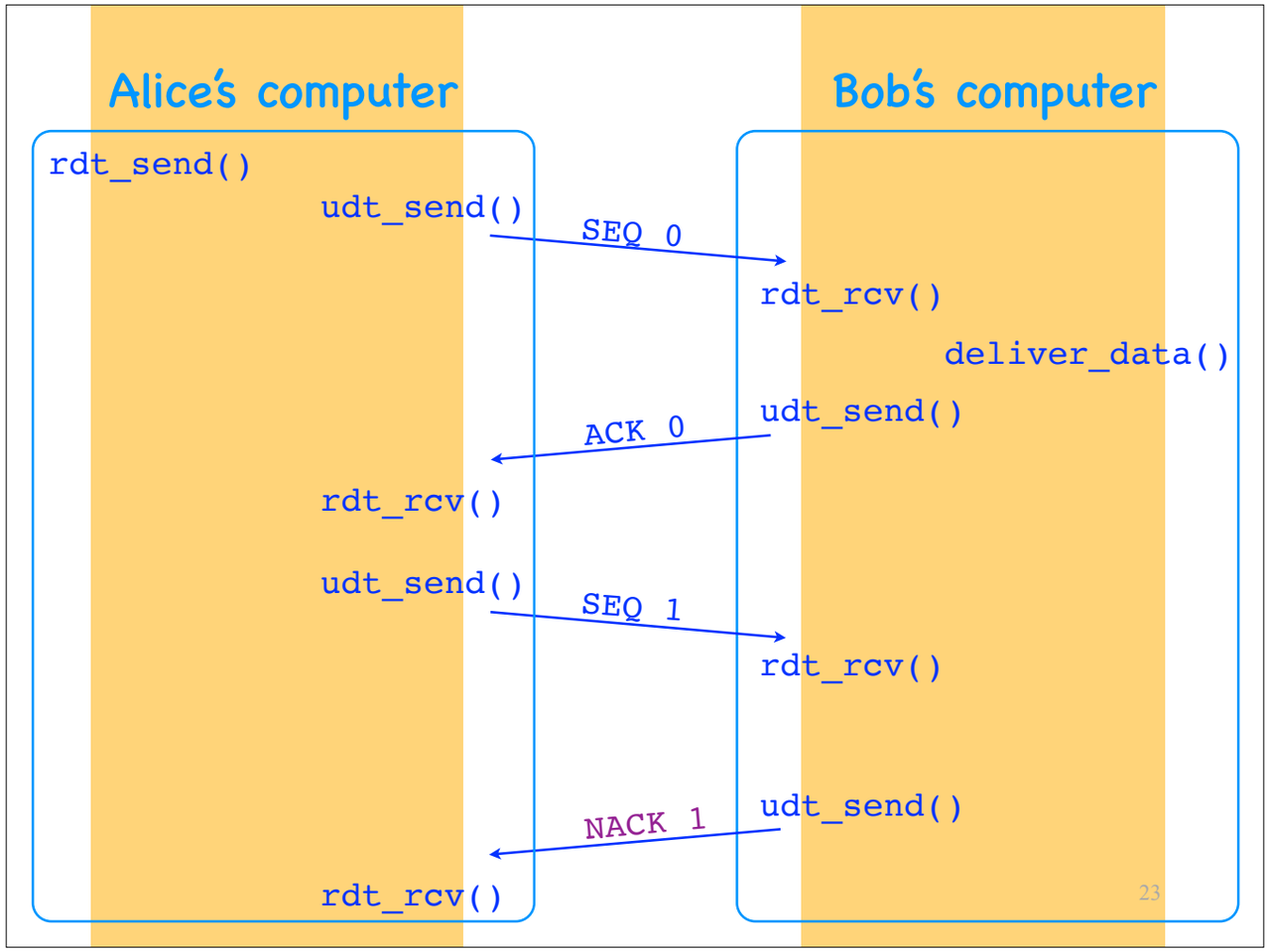
- If Alice's transport layer transmits a new segment, that has a different sequence number, so Bob's transport layer knows that Alice is transmitting a new segment.

-> How many sequence numbers do we need?

Sequence number

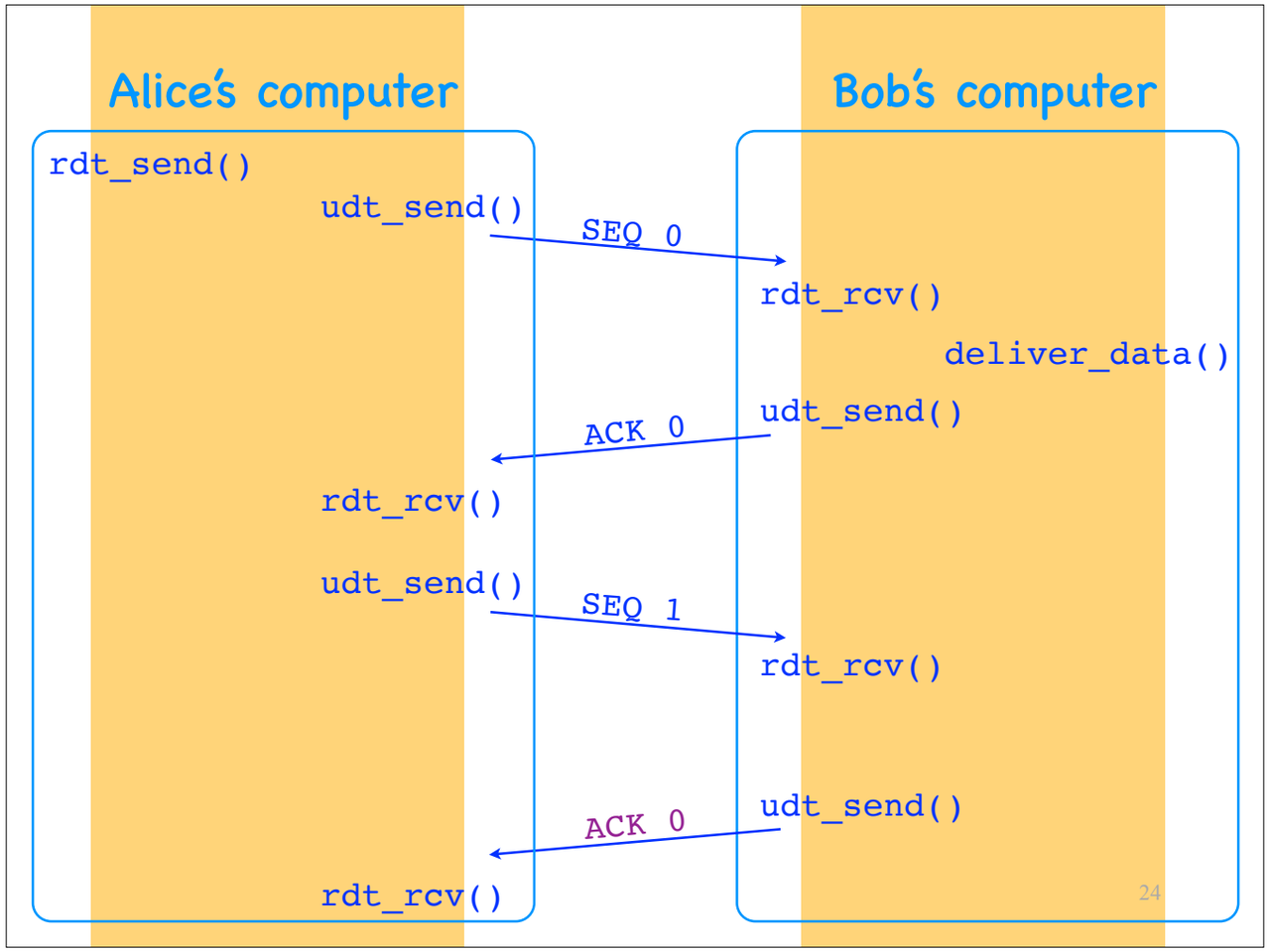
- An identifier for data
- Sender adds SEQ to each segment
  - transport-layer header field
- Receiver uses it to disambiguate data

So: Another important element of reliable data delivery is sequence numbers…

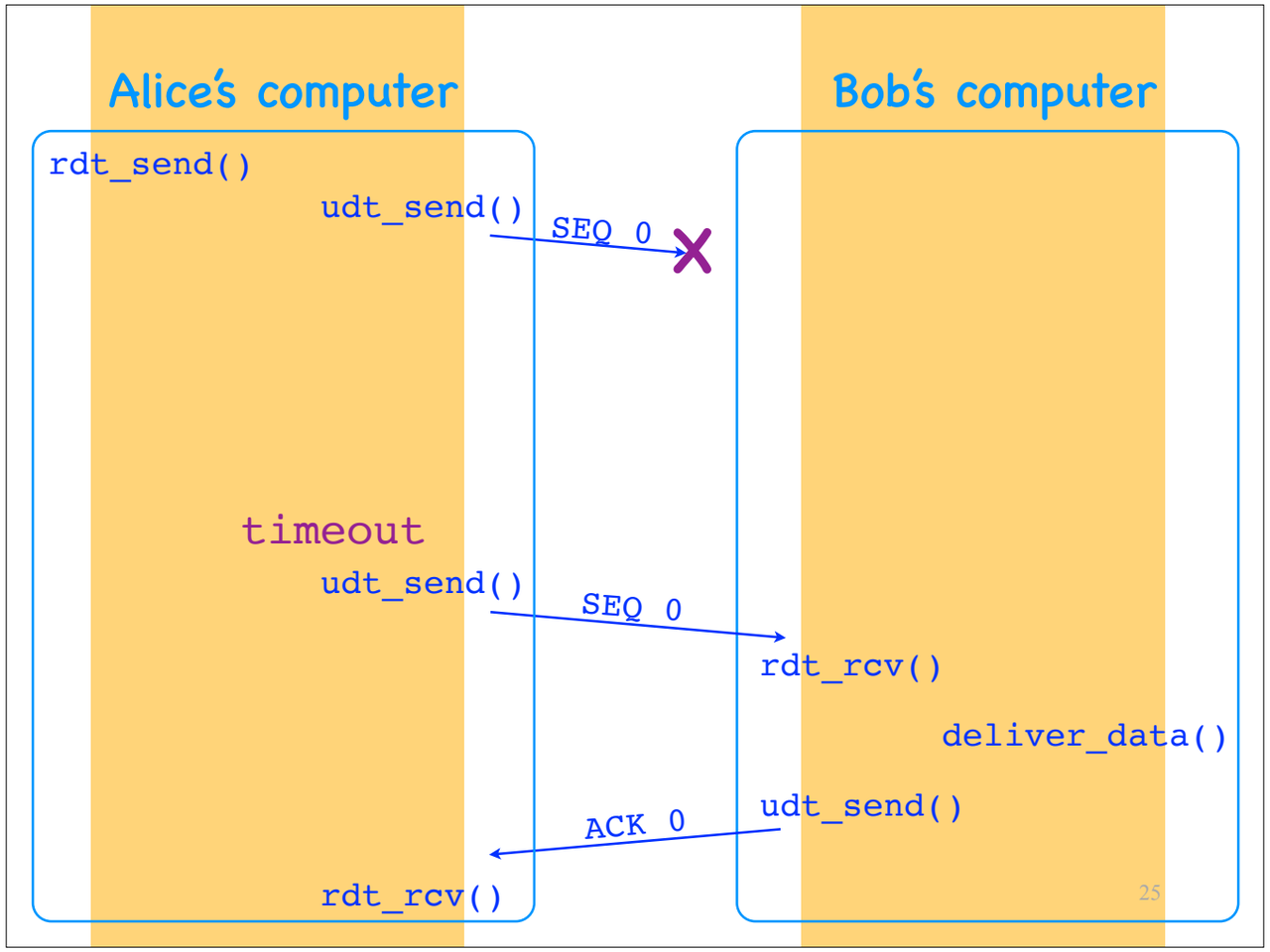


Sequence numbers make it possible to have *implicit* negative ACKs.

For example, suppose Bob's transport layer needs to send a NACK for segment 1.
Instead…

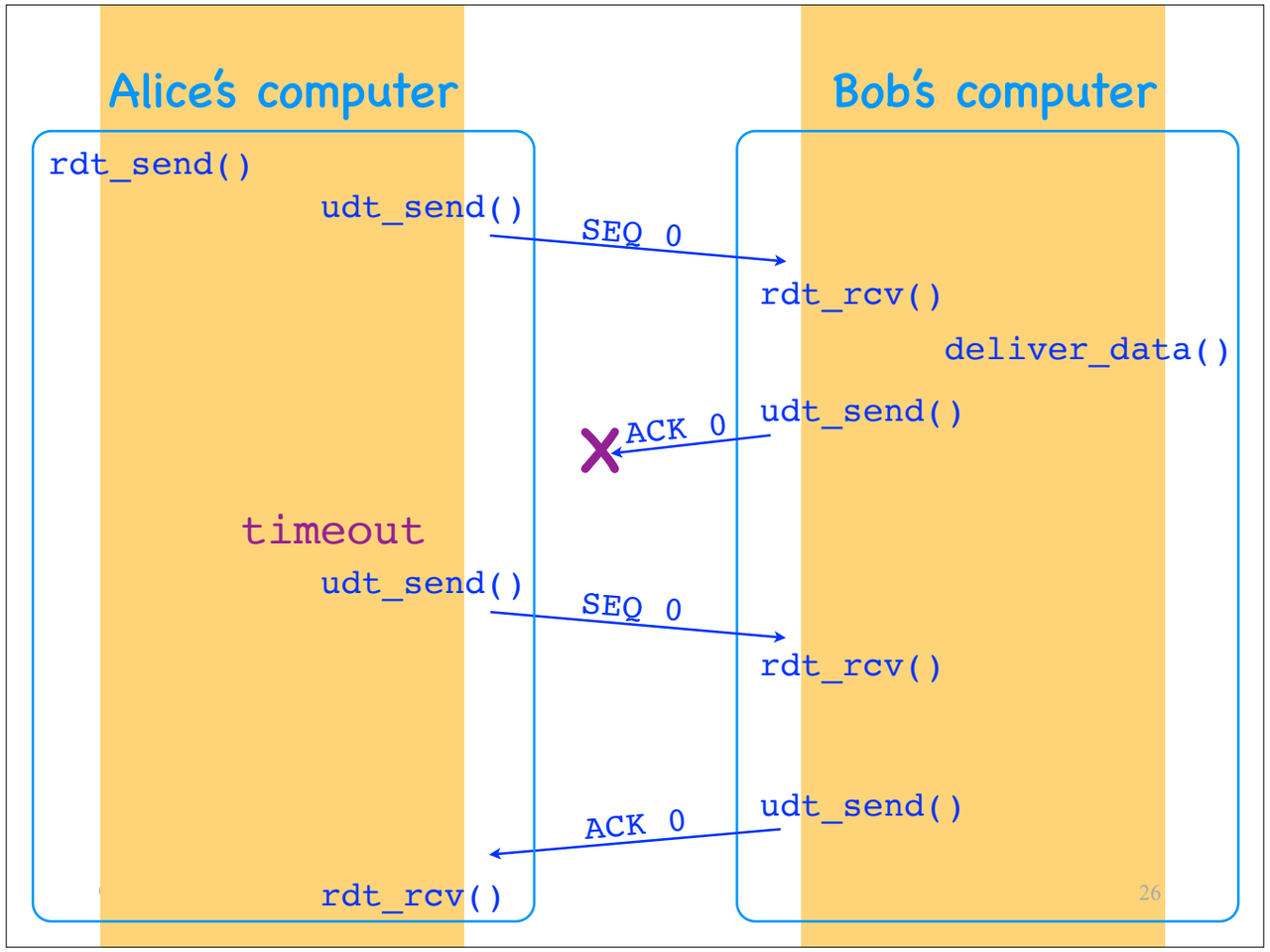


it can send a (second) ACK for segment 0, which means: “The last segment I received correctly from you had SEQ 0.” This is equivalent with Bob sending a NACK for segment 1.



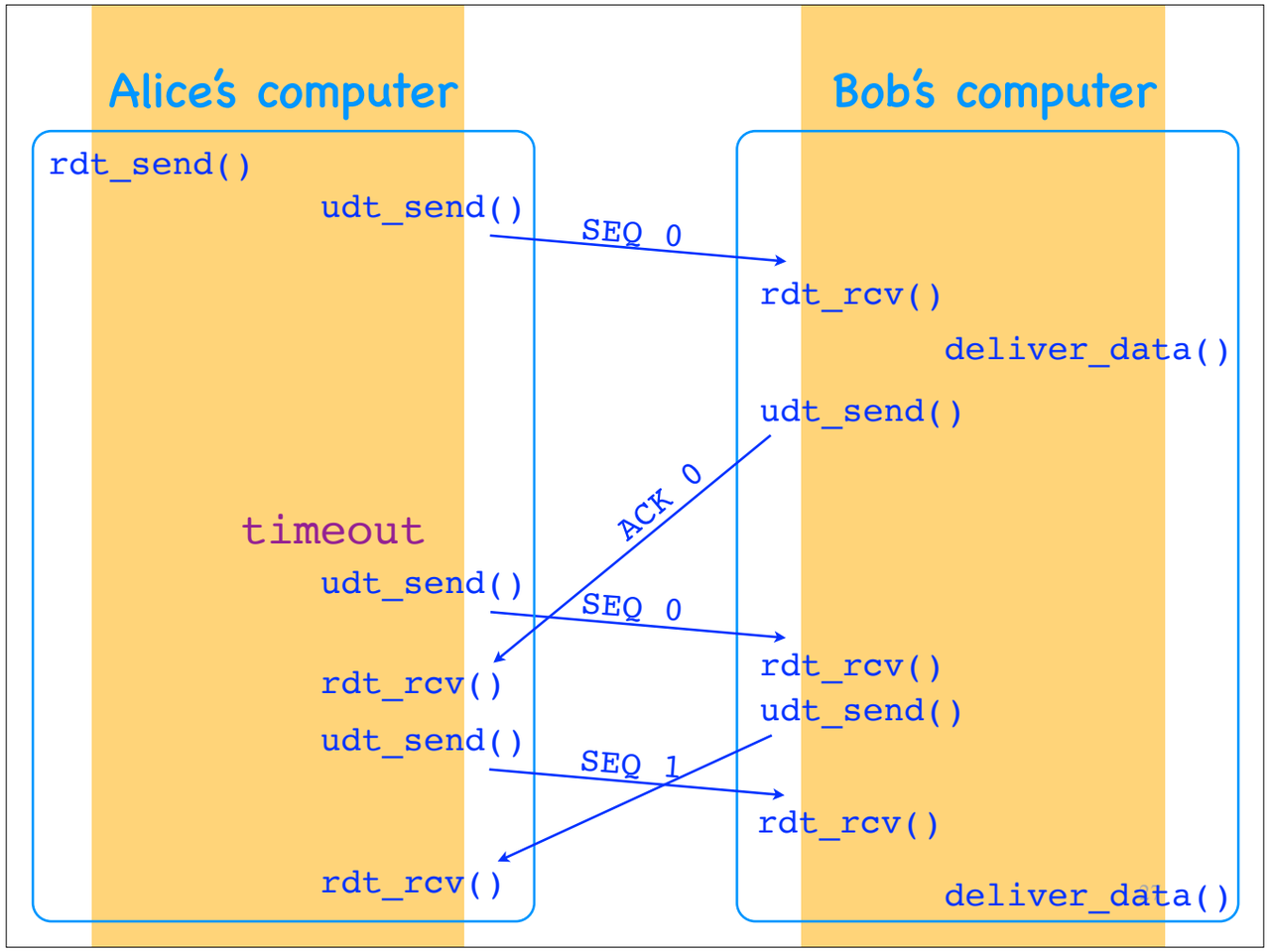
Now suppose Alice sends a segment to Bob, and the segment is lost.
Now Alice's transport layer receives no feedback at all about its segment.

To deal with this scenario, Alice's transport layer uses timeouts:
- Every time it transmits a segment, it starts a timer.
- If the timer expires and Alice's transport layer has not received an ACK for the segment, it assumes the segment lost and retransmits.



What if Alice's segment reaches Bob, but Bob's ACK is lost?
Alice's transport layer still times out and retransmits the segment.

In general, there is no way for Alice's transport layer to know if a timeout is due to a segment really being lost or the ACK for the segment being lost. It must act conservatively and retransmit anyway.



And what if Alice's segment reaches Bob, and Bob's ACK reaches Alice, but (due to some unpredicted network delay) after Alice's transport layer has timed out?

There's no way for Alice's transport layer to avoid useless retransmissions like this one.
However, when Alice's transport layer receives the first (the delayed) ACK 0, it can immediately transmit SEQ 1 — it does not need to wait for Bob's second ACK 0.

In the meantime, when Bob's transport layer receives the retransmitted SEQ 0,
it will ACK 0 again, and that's OK. Alice's transport layer will just ignore the second ACK 0.

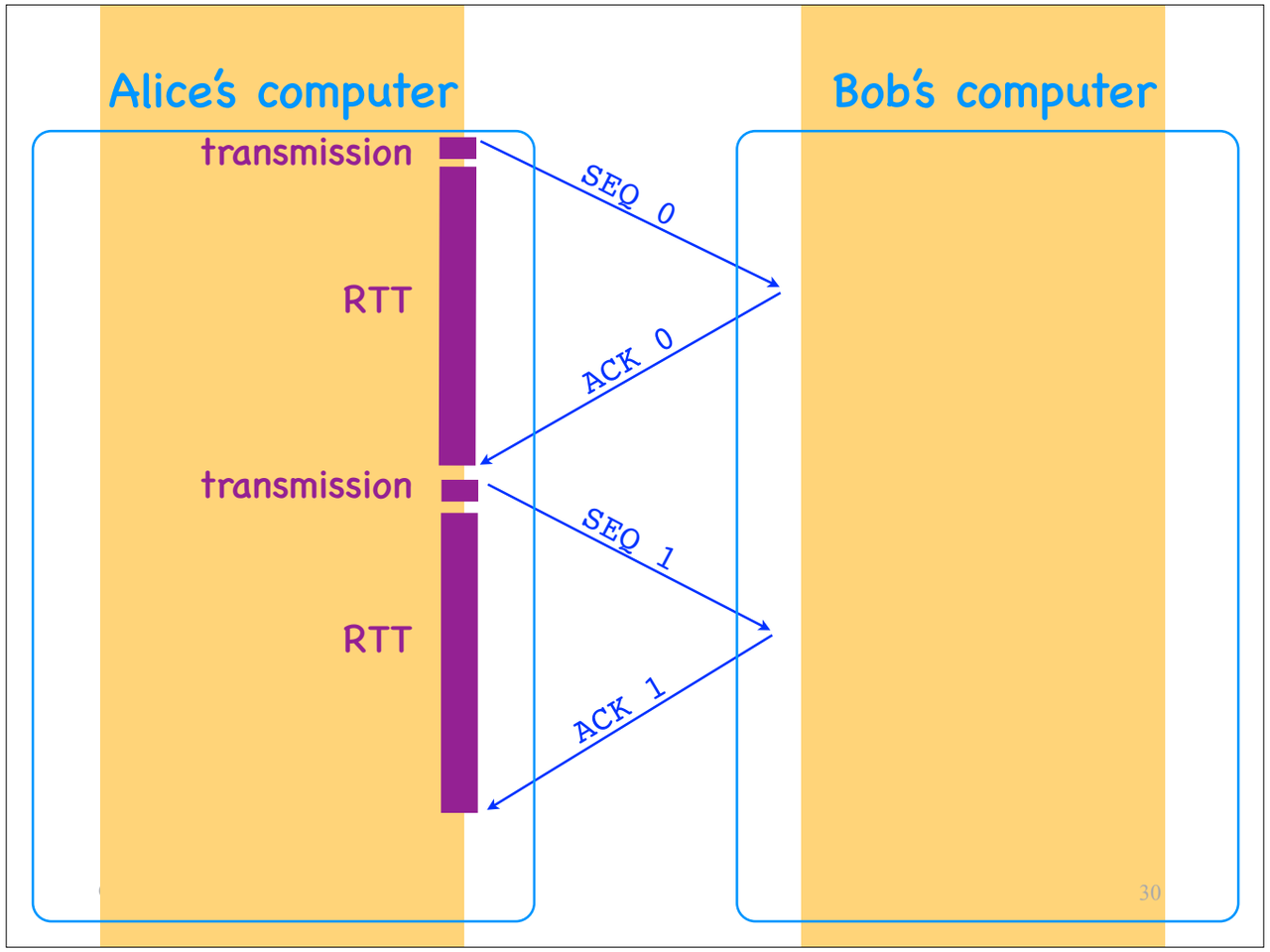
Timeout

- No arrival of an expected ACK
  - a segment was lost or delayed
  - the ACK for a segment was lost or delayed
- Sender uses it to overcome data loss
  - if the sender times out, it retransmits

So: Yet another important element of reliable data delivery is the timeout…

Basic elements

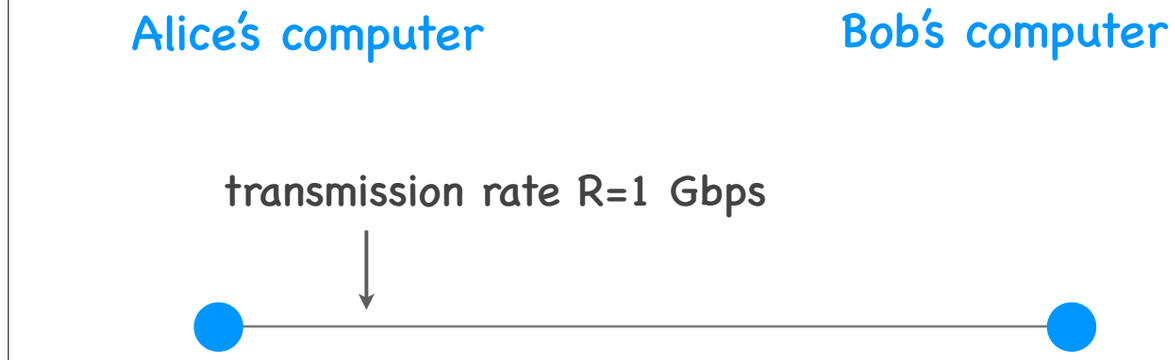
- Checksums
  - detect data corruption
- ACKs + retransmissions + SEQs
  - overcome data corruption
- Timeouts + ACKs + retransmissions + SEQs
  - detect and overcome data loss



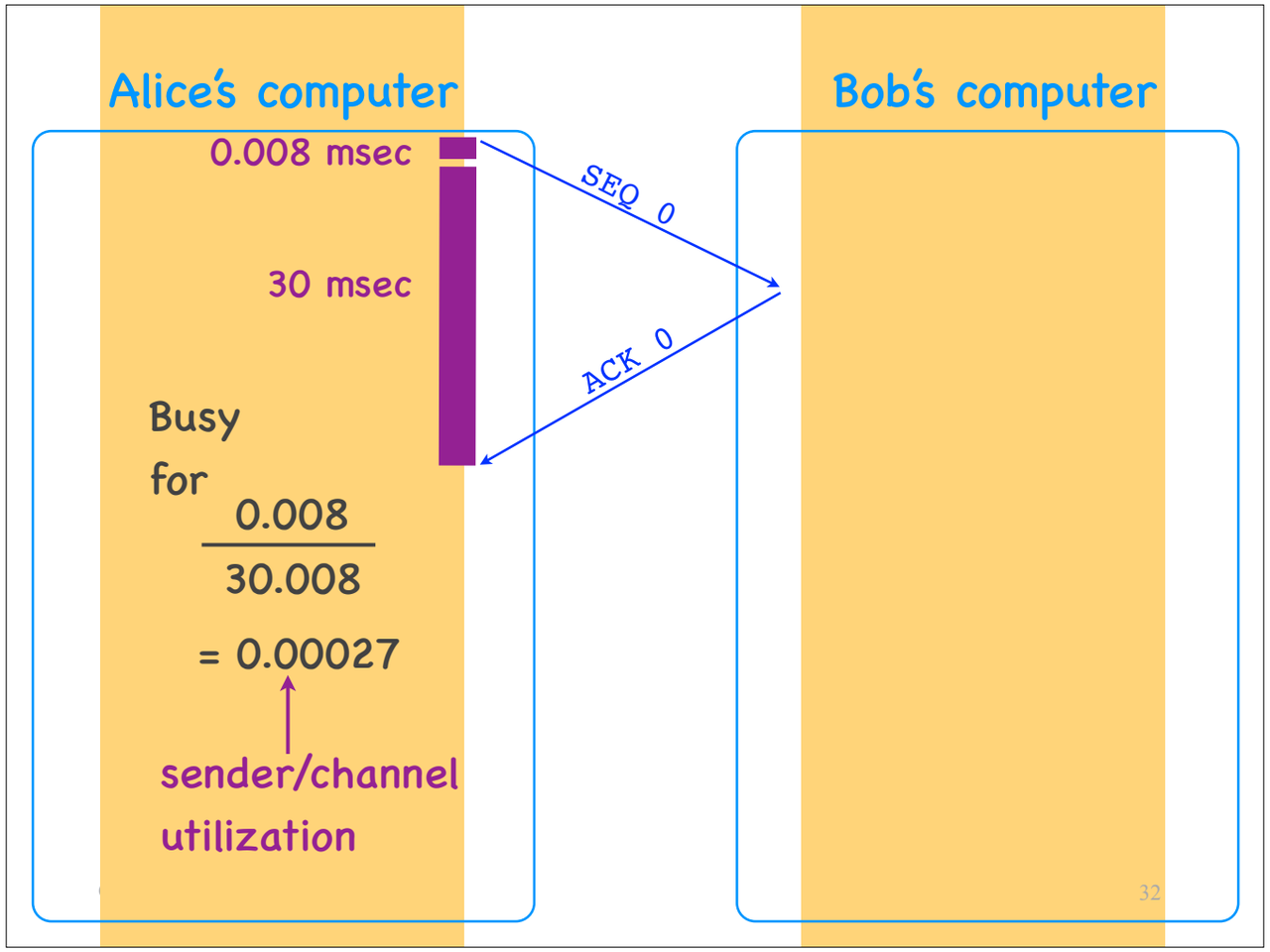
I have described to you a “stop-and-wait” transport-layer protocol,
where the sender always sends only one segment and then waits for an ACK or a timeout.
Here is what communication looks like when no segment is corrupted or lost.

—>What is the problem with this protocol?

Performance and, in particular, throughput.
Alice’s transport layer transmits one segment, then waits for an ACK.
Then she transmits another segment, then waits for an ACK.
And so on.

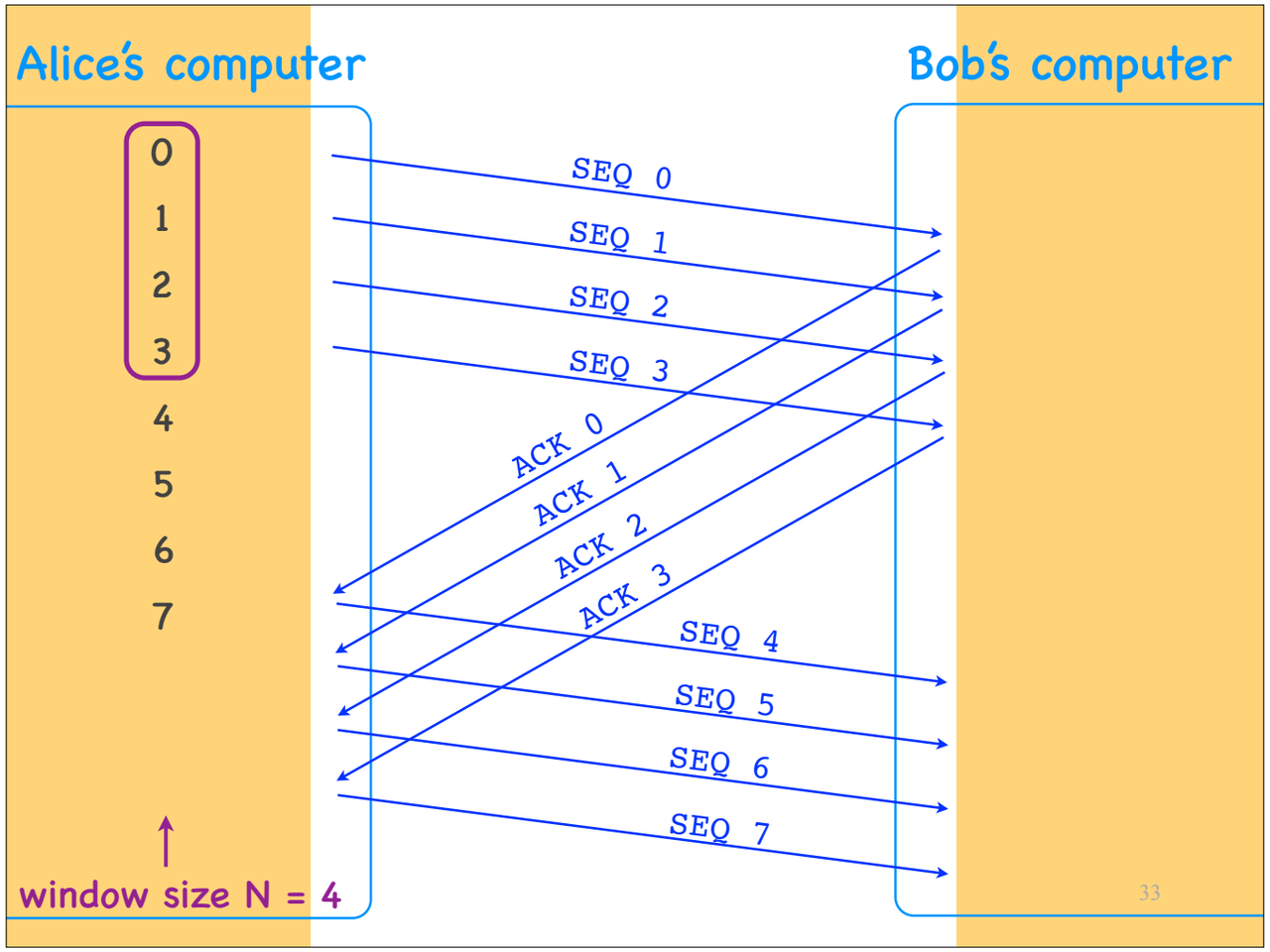


Let's go back, for a moment, to this view of sender and receiver.
Suppose Alice's and Bob's computers are directly connected over one physical link of transmission rate 1 Gbps and propagation delay 15 msec.
Suppose the segment size is 1000 bytes.
Hence, the transmission delay for each segment is 8 usec.



This means that Alice transmits for 8 usec and then does nothing for 30 msec.
This is a long wait.
Another way to put this is that Alice's transport layer is busy only for 0.027% of the time.
We call this number the *sender* or *channel utilization*.
So, the stop-and-wait protocol has poor sender utilization.

—>How can we improve sender utilization?

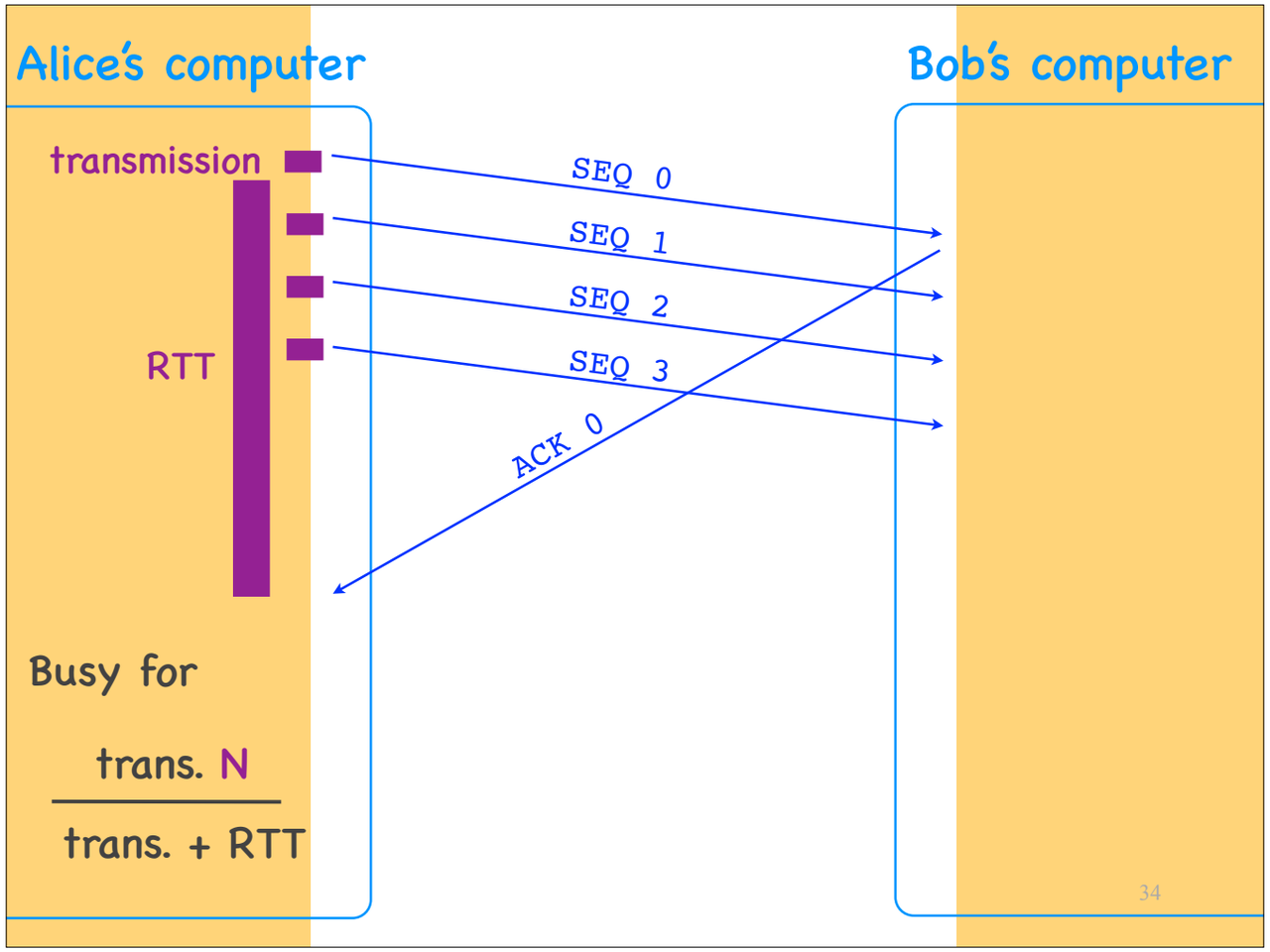


Let's introduce the notion of a sender window.

What I am showing here on the left are sequence numbers that Alice will use,
and this rectangle indicates that she has a window of size 4,
which means that she will send out up to 4 segments without waiting for an ACK.

For example:
- She sends out segments 0 to 3, then stops.
- When she receives Bob's first ACK, she advances her window by 1 SEQ,
which means that she can send out segment 4.
- When she receives Bob's second ACK, she advances her window by 1 more SEQ,
which means that she can send out segment 5.
- And so on.

At any point in time, she has sent no more than 4 unacknowledged segments.
We say that her window size is 4.



Let's see by how much we have improved sender utilization:

Now Alice transmits 4 segments and waits for an ACK,
so, she is busy for transmission delay times 4,
divided by transmission delay + RTT.

So, we have improved sender utilization by a factor N,
which is the sender's window size — in our specific example, 4.

—>How large would you make N?
—>Is there any reason to not increase N arbitrarily?

Sender utilization

- Stop and wait: poor sender utilization
  - the sender does nothing while waiting for the receiver's ACK or the timeout
- Pipelining: better utilization
  - the sender sends up to N un-ACKed segments
  - N = sliding window size

We first put together what we call a “stop and wait” protocol:
The sender sends out a segment, then does nothing until it receives an ACK from the receiver, or it times out.

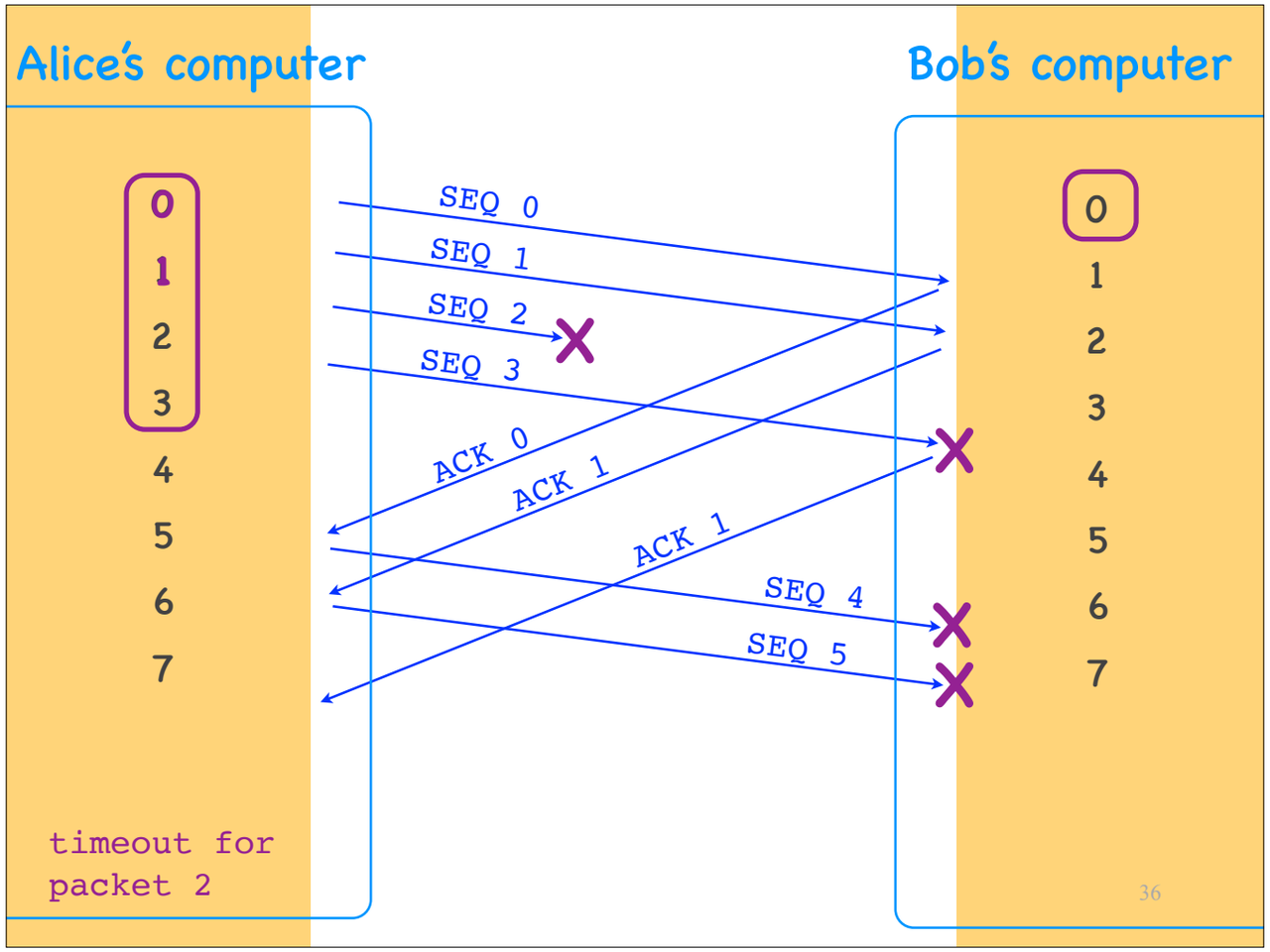
Then we put together what we call a “pipelined” protocol,
where the sender sends out multiple unacknowledged segments.

The advantage of pipelining is that it improves sender utilisation by a factor that is equal to the sender’s window size.

—

But what happens when we use pipelining and segments get corrupted or lost?

There are two protocols that address this scenario:



The first one is called Go-back-N:

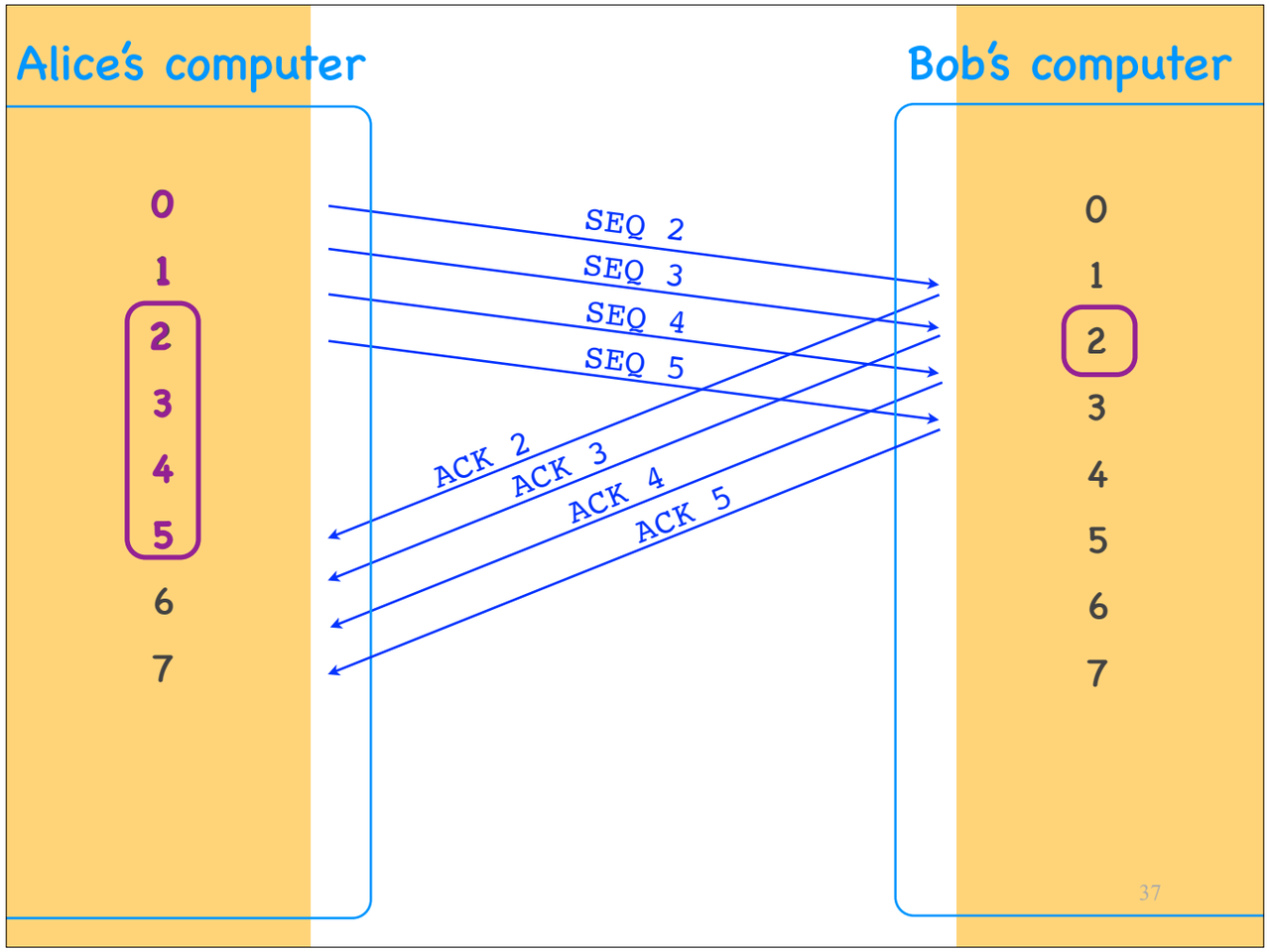
The receiver never accepts any out-of-order segments:
- The receiver also has a window, of size 1.
- This means that the receiver is willing to accept only one SEQ.
- It discards anything else.

For example:

- Alice sends segment 0, Bob receives it, and advances his window by 1 SEQ, which means that he is now ready to receive segment 1.
- Alice sends segment 1, and now Bob is ready to receive segment 2.
- Alice sends segment 2, but it is dropped.
- When she sends segment 3, Bob discards it, because his window is stuck at 2.

  - In the meantime, Bob has ACKed segment 0.
  - When Alice receives ACK 0, she advances her window by 1 SEQ, and sends segment 4, but Bob rejects it, because his window is still stuck at 2.
  - Bob has also ACKed segment 1.
  - When Alice receives ACK 1, she again advances her window by 1 SEQ, and sends segment 5, but Bob rejects it.
  - When Bob receives segment 3, he still ACKs segment 1, indicating that that is the last segment that he received in order.

When does Alice realize that something is wrong?
When she times out waiting for an ACK for segment 2.

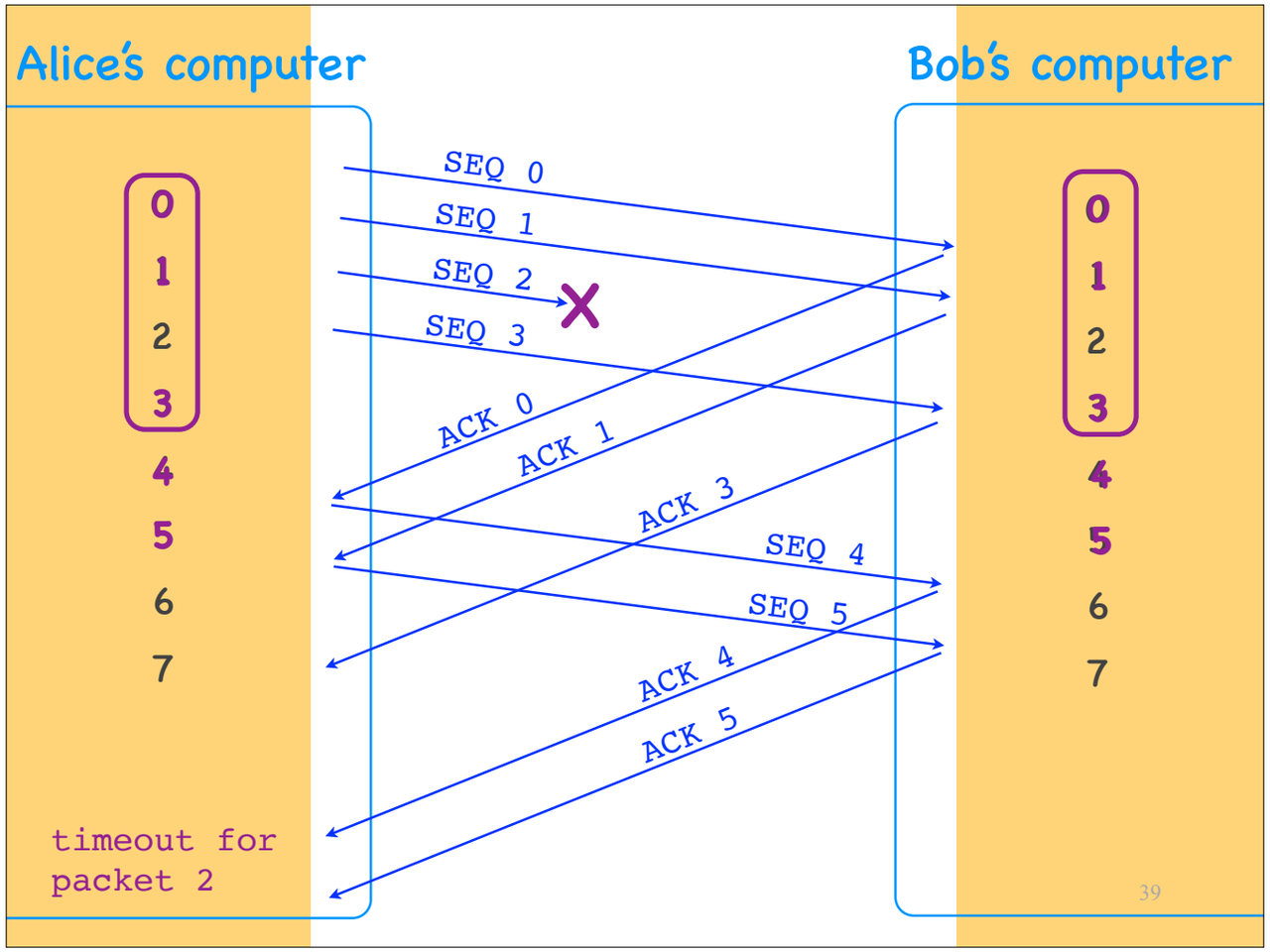


At that point, she retransmits segment 2 plus all the subsequent segments that she has sent out, because she knows that Bob must have discarded them, since he did not receive her original segment 2. Once Bob has received all the retransmitted segments correctly and ACKed them, Alice can advance her window and move on.

In this approach, when Bob ACKs segment 1, that means that he has correctly received every segment up to and including 1. This is called a **cumulative ACK**.

Go-back-N

- The receiver accepts no
  out-of—order segments
- ACKs are cumulative
  - an ACK for segment 10 indicates that all segments until and including 10 have been received
- When the sender retransmits,
  it retransmits all the un-ACK-ed segments



The other pipelining protocol is called Selective Repeat:

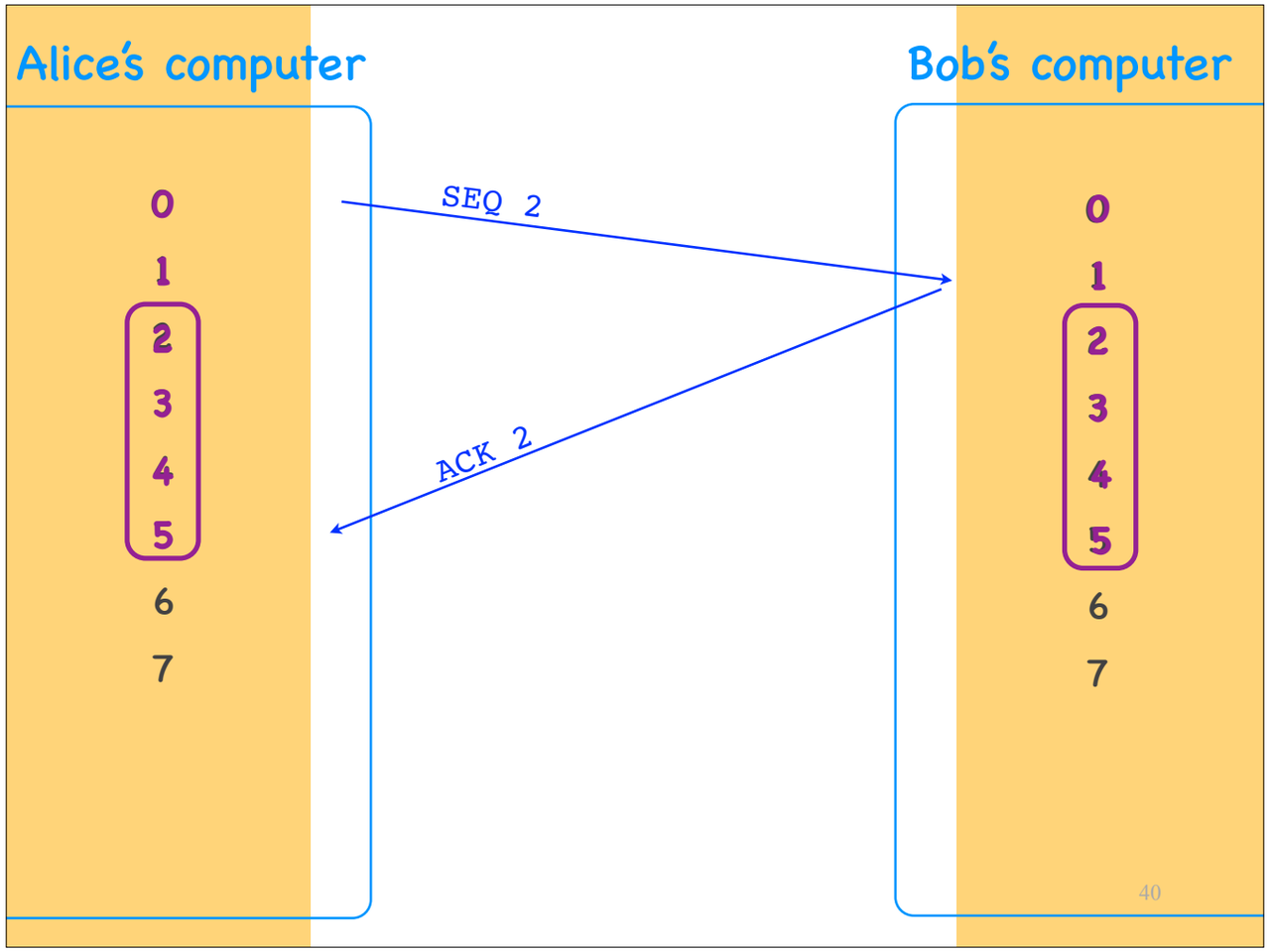
The receiver does accept a certain number of out-of-order segments:
- The receiver has a window, same size as the sender's (in our example 4),
- which means that the receiver is willing to accept up to 4 out-of-order segments.

For example:

- When segment 0 arrives, Bob advances his window by 1 SEQ.
- Same when segment 1 arrives.
- Segment 2 is dropped.
- When segment 3 arrives, Bob does not discard it, because SEQ 3 is inside his window, so he accepts and ACKs segment 3.

- In the meantime, Bob has ACKed segment 0, causing Alice to advance her window by 1 SEQ and send out a new segment.
- Same when Bob ACKs segment 1.
- When Bob ACKs segment 3, Alice does not transmit a new segment, because her window is now closed.
- Same when Bob ACKs segments 4 and 5.

Alice realizes that something is wrong when she times out waiting for an ACK for segment 2.



At that point, she retransmits only segment 2. When Bob receives the retransmitted segment, he can finally advance his window and ACK 2. When Alice receives ACK 2, she also advances her window.

In this approach, when Bob ACKs segment 1, that means that he has correctly segment 1, it does not say anything about previous segments. This is called a **selective ACK**.

Selective Repeat

- The receiver accepts N-1 out-of-order segments
- ACKs are selective
  - an ACK for segment 10 indicates that segment 10 has been received
- When the sender retransmits, it retransmits only one segment

—> When would you prefer a GoBackN protocol, and when would you prefer a Selective Repeat protocol?