Basic Reliable Transport Protocols

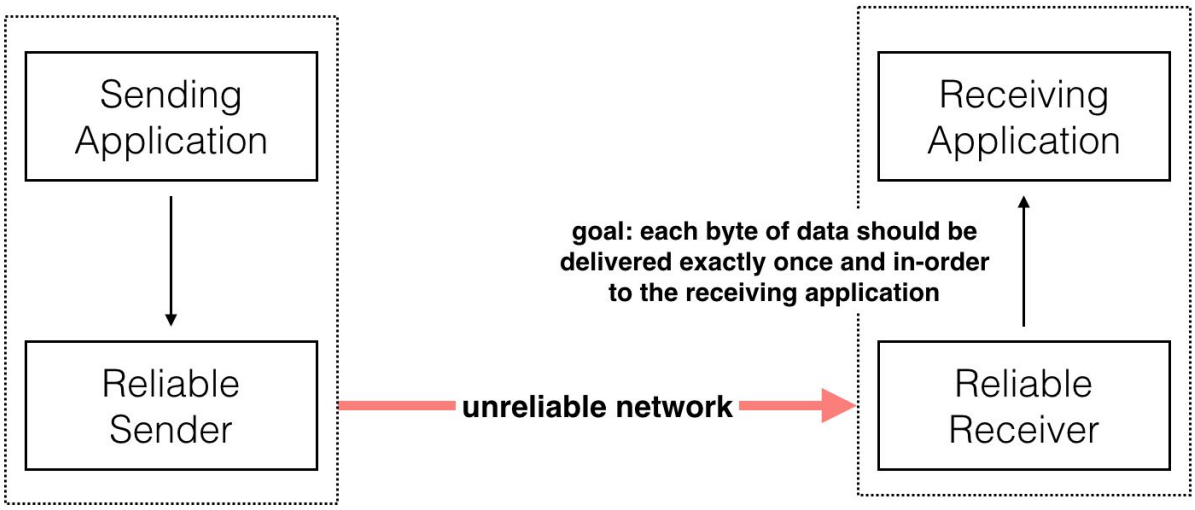
Do not be alarmed by the length of this guide. There are a lot of pictures.

You've seen in lecture that most of the networks we're dealing with are *best-effort*: they do their best, but do not guarantee perfect reliability. This means that packets can be delayed, reordered, or lost. Most commonly, packets are lost because they arrive at a switch whose queue for incoming packets is already full. Reordering (in the absence of loss) typically occurs if the optimal route between the sender and receiver changes during their communication.

In this hands-on, you're going to use the Unix tool `tcpdump` to get some information about TCP, the most common *reliable transport protocol* on the Internet. A reliable transport protocol is a protocol that (attempts to) provide reliability on a best-effort network. In addition to reliability, TCP also provides *congestion control*, which we'll get to in a few lectures. This hands-on is all about reliability.

Before you interact with TCP, we're going to build a simpler reliable transport protocol (but one that is still similar to TCP). The goal of any reliable transport protocol is as follows: we want the receiving application to receive each byte of data exactly once, and in the exact same order that the sending application sent it — another way to say that is that we want **exactly-once, in-order delivery**.

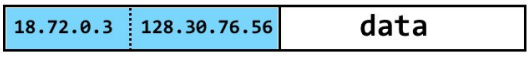
It's helpful to think of reliable transport protocols in terms of our layering model, though we really only need two layers here:



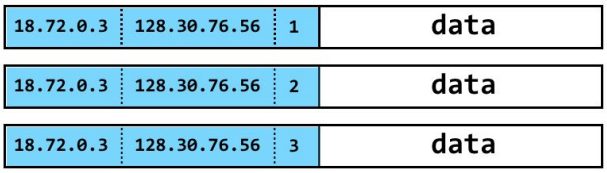
The sending application is generating the data and attempting to deliver the data to the receiving application. We'll focus on constructing a reliable sender and reliable receiver "underneath" underneath the application (at the transport layer). It will be the job of this reliable sender and receiver to achieve exactly-once, in-order delivery.

Let's tackle the "in-order" part of our reliable transport protocol first.

In general, packets are data with an attached header. That attached header contains some meta-information about the packet; for right now, we'll assume that it includes both the source address and the destination address. A packet from a machine with address 18.72.0.3, sent to a machine with address 128.30.76.56, might look something like this (the blue components are part of the header):



To help with ordering, we're going to assign each packet a **sequence number**, which will also be stored in the packet header. You will often see us use SEQ as an abbreviation for sequence number. Now, packets will look something like this:

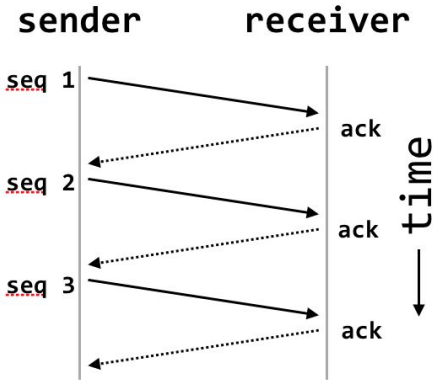


Why is this helpful? Before, the receiver had no way to discern the order in which packets had been sent by the sender. Now, the receiver can detect when packets have been received out of order, and can also reorder them.

Check your understanding: How can a receiver tell if it has received packets out of order?

Now for the reliability part of our protocol. We'll use two different mechanisms to help with this. The first is **acknowledgments**, or ACKs. Every time the receiver receives a packet, it will send a small packet — an "ACK" — back to the sender. This lets the sender know that the packet was received successfully.

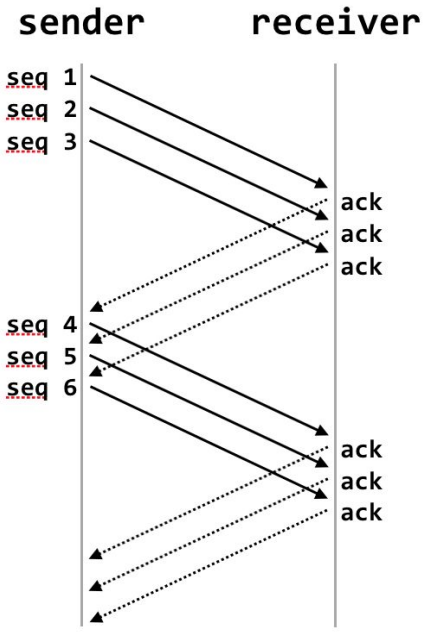
So in our communication, we'll start to see this:



This type of diagram is known as a *waterfall diagram*. In it, time flows down. The diagram above depicts a sender sending packet 1, and the receiver sending an ACK in response. When that ACK arrives at the sender, it sends packet 2, and so on. The ACKs are shown as dotted lines only to make it easier to differentiate them from packets that carry data.

In the above example, the sender only has one *outstanding packet* at a time, i.e., there is never more than one packet on the network between the sender and receiver.

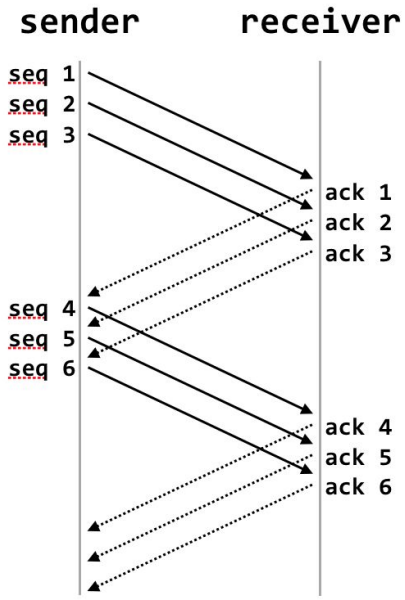
There's no reason that our sender can't have more than one outstanding packet at a time. For example:



Here, the sender has three outstanding packets at a time. This collection of outstanding packets is known as its **window**. For now, we'll assume that the sender has a fixed **window size**, W. There can be no more than W packets in the sender's window at a time. Above, W = 3.

Since there can be multiple packets in the sender's window, the receiver needs to specify which packet it's ACKing. There are a few different approaches to this problem; the one we'll use is known as cumulative ACKs: an ACK for packet *k* means the receiver has received *all packets up to and including k*.

Our picture is now:

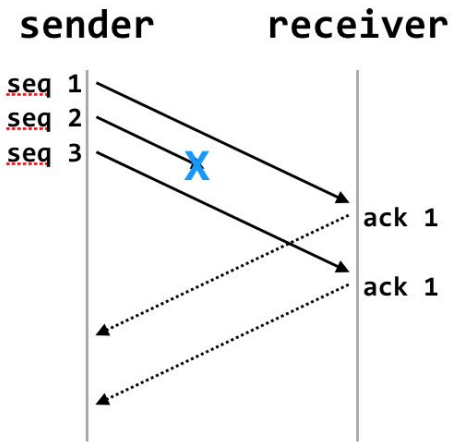


Check your understanding: Suppose that packets 4 and 5 got reordered by the network, such that packet 5 arrived at the receiver before packet 4. How would this picture — in particular, the ACKs — look different?

From this picture, you can also see how the sender is able to make sure there are no more than W packets in its window at a time: when it receives an ACK that indicates that a particular packet *k* has been received, the sender knows that *k* is no longer in the window, and so the sender can send a new packet.

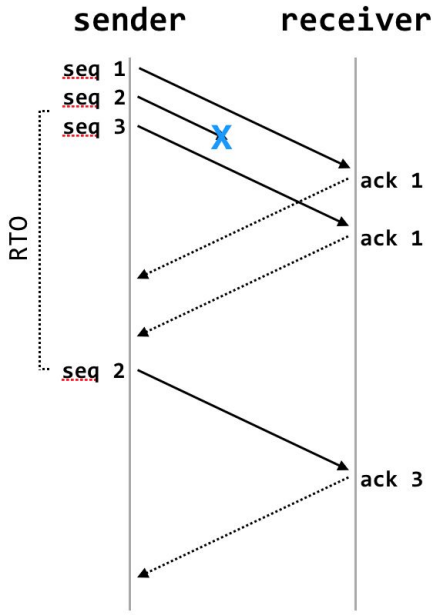
The type of protocol we're currently building is known as a **sliding-window protocol**: as the communication occurs, the sender's window slides across the stream of packets it has to send.

Let's see what happens if a packet is lost on its way to the receiver:



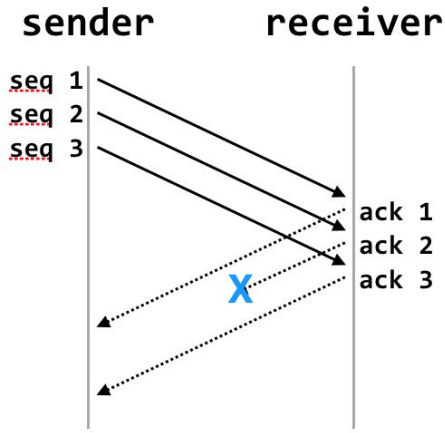
The sender can tell that packet 2 got lost because it didn't receive an ACK for it, but right now, our protocol has no mechanism to get packet 3 to the receiver. We'll add in **retransmissions**: the sender will retransmit a lost packet if it doesn't receive an ACK for it after a certain amount of time. This amount of time is known as the *retransmission timeout*, or RTO.

Our picture is now:



Setting the RTO involves a fair amount of work. For the purposes of 6.033, you can assume that the RTO is set to be slightly larger than the round-trip-time between the sender and the receiver (i.e., slightly larger than the amount of time we expect to elapse between the sending of a packet and the receipt of its ACK).

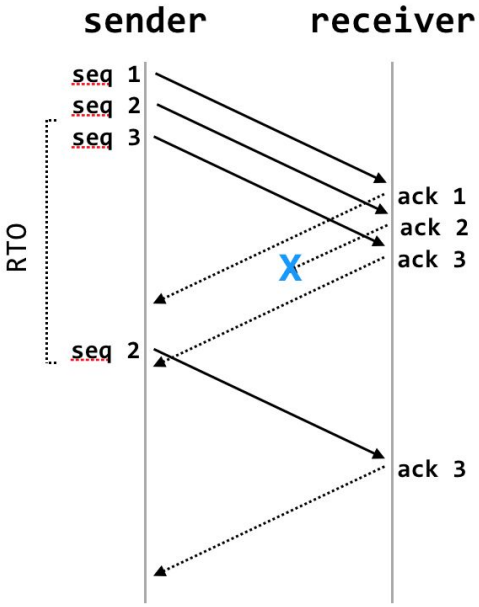
We've talked about what happens with packets are lost. What about when ACKs are lost?



Above, we see that packet 2 got to the receiver just fine, but its ACK was lost. The sender can't tell the difference between this scenario and one where packet 3 itself is lost.

Check your understanding: Why can't the sender tell the difference between these two scenarios? What signals to the sender that a packet has been lost?

As such, the sender will retransmit the packet:



The receiver, then, has received two copies of packet 2. It will ACK both of them.

Check your understanding: Why does the receiver need to send an ACK for the second copy of packet 2? You may need to think about scenarios other than the one pictured above.

Check your understanding: If the ACK for packet 3 had arrived before the sender's retransmission of packet 2, could the sender have avoided that retransmission? (I.e., is there a way for the sender to tell that packet 2 had actually been received?)

However, the receiver shouldn't deliver both copies of packet 2 to the receiving application — that would violate the exactly-once part of our exactly-once in-order delivery. To do this, the receiver keeps track of which packets it has received so that it doesn't send duplicates to the receiving application.

As part of that process, it keeps a buffer of packets that it has received, but that aren't ready to be sent to the application yet (for example, if packet 2 had indeed been lost, the receiver would not have been able to deliver packet 3 until it got a copy of packet 2). The receiver will store packets in that buffer until it's able to deliver them in-order, at which point it will deliver as many packets to the application as it can.

All told, here is our reliable transport protocol:

At the sender:

- The sender will keep a list of all of its outstanding packets. We'll call that list outstanding_packets; initially it is empty.
- If len(outstanding_packets) < W, transmit the next packet (call it packet k). Store packet k and the current time in outstanding_packets.
- When an ACK for packet k is received, remove k from outstanding_packets.
- Periodically check the packets in outstanding_packets; if any were received more than RTO seconds ago, re-transmit them.

At the receiver:

- Send an ACK for every received packet.
- Save delivered packets — ignoring duplicates — in a local buffer.
- Keep track of the next packet the receiving application expects. After each reception, deliver as many in-order packets as possible.

Note that the protocol at the receiver is a bit simpler than at the sender. The sender has to continuously check whether packets need to be retransmitted. The receiver only needs to worry about ACKing every packet and delivering packets in order.

In sliding-window protocols, it matters *very much* what the sender's window size is (i.e., what W is). Too small, and the sender won't get as much throughput as it could. Too large, and the sender will fill up the queues between it and the receiver.

For a single sender and receiver, there's a fairly straightforward calculation that will tell us what the optimal window size is. However, on the Internet, there are many senders and receivers, and the amount of data that they are sending changes all of the time. As a result, the optimal window size for a sender changes all of the time.

In Lecture 11, we'll look at how the TCP deals with this problem by implementing *congestion control* in addition to providing reliable transport.