Lecture 4: Reliable transport

Anirudh Sivaraman

2021/10/19

Last week, we began to look at each of the five layers of the Internet's protocol stack. We started with the application layer and the Sockets API that applications use to communicate with other applications. The next two weeks will focus on the transport layer. This week we'll focus on how the transport layer provides the abstraction of a reliable, in-order bytestream (TCP). Next week, we'll look at how the transport layer regulates the transmission rate of end hosts on a network to prevent end hosts from overwhelming the network.

Specifically, we'll discuss two topics in this lecture.

1. *Stop-And-Wait*: This is likely the simplest technique to achieve reliability. As the name suggests, a sender in the Stop-And-Wait protocol sends some data to the receiver, stops and waits until it receives an acknowledgement from the receiver, and then moves on to the next piece of data. The Stop-And-Wait protocol is simple, but provides a good starting point to understand the sliding window protocol in the next lecture. The sliding window protocol is similar in spirit to real TCP implementations.

2. *Retransmission timers:* Because a network might drop packets (e.g., an overflowing queue, bit flips on wireless transmission media[1]), a Stop-And-Wait sender needs a strategy to retransmit lost packets. Typically, the sender doesn't know with certainty whether a packet has been lost, and if so, which packet it is and when it was lost. Instead a sender has to indirectly infer a lost packet by the lack of an acknowledgement for that packet.[2] Typically, the sender infers a lost packet by starting a retransmission timer as soon as a packet is transmitted. This timer times out if it hasn't received an acknowledgement for a long time. This long time is called the timer's timeout. We'll look at how the sender decides on the timer's timeout. If the timeout is too short, the sender risks overwhelming the network with duplicate packets, and if it's too long, the sender risks degrading the transport-layer throughput of the protocol by waiting too long and letting the network idle.

1 Stop-And-Wait

1.1 The problem

Before we describe the Stop-And-Wait protocol, it is useful to formally state the reliability problem. Specifically, let's assume we have an ordered list of packets P1, P2, . . . , PN at a sender. The sender needs to get these to the receiver in the same order even if the network reorders packets arbitrarily, duplicates packets arbitrarily, delays packets arbitrarily, and drops packets arbitrarily. In a real TCP implementation, we would replace an ordered list of packets with an ordered list of bytes—hence the term reliable in-order *bytestream*. This does require some changes, which we won't get into in this course, but the techniques are not materially different from what we discuss here.

However, we will assume some good things of the network. First, we will assume that the network is not *partitioned*, i.e., there is a path from the sender to the receiver that has the ability to carry a non-zero number

[1] Wireless transmission media such as cellular and WiFi networks are particularly prone to packet errors. This is because at the analog level, bits are represented by voltage signals, and these signals are distorted from their original values by many sources: noise, interference from other senders, and signal attenuation when traveling through the atmosphere and through walls.

[2] Some router mechanisms allow routers to send a notification packet back to the end host if their queues overflow. But such techniques don't cover all errors (e.g., bit flips) and the notification packet itself might be lost.

of bits per second. If the network is partitioned, there is no hope of sending anything from the sender to the receiver. Second, we will assume that the sender and receiver computers do not crash during the process of reliably transmitting the ordered list P1, P2, . . . , PN. Providing reliability in the face of crashes is much harder and requires a non-volatile storage medium such as a hard disk that survives between crashes.

1.2 The Stop-And-Wait protocol

The Stop-And-Wait protocol is very straightforward at both the sender and the receiver. At the sender side, we carry out the following steps.

1. Transmit the first packet.
2. Wait for this packet to be acknowledged by the receiver. This may require the sender to retransmit the packet if the packet has not been acknowledged even after a long time. We'll define what "long time" means in §2.
3. Repeat the same procedure with the second, third, fourth, . . . , nth packet.

At the receiver side, we carry out the following steps.

1. Send an acknowledgement for the packet that was just received.
2. Maintain a variable (`next_in_order`) corresponding to the packet number of the next packet that is required to provide reliable and in-order packet delivery. For instance, if the packets received so far have packet numbers 1, 2, 3, 4, 4, the value of `next_in_order` is 5.
3. If the just received packet's number equals `next_in_order`, increment `next_in_order`. Otherwise, just discard the packet. This step is required because packets can be duplicated. Packet duplication can happen because of errors within the network. It can also happen because the Stop-And-Wait retransmitted a data packet too soon, in which case both the original and retransmitted packet will be received at the receiver.

The Stop-And-Wait protocol needs a way to identify the packet number both in the data and the acknowledgement packets. This is commonly called a sequence number and is carried by both data and acknowledgement packets in the Stop-And-Wait protocol.

This sequence number field is also found in TCP today, except that TCP sequences bytes and not packets. If you look at the TCP header, there are two 32-bit sequence number fields in it. One is called the sequence number and the other is called the acknowledgement number. The sequence number identifies the first byte of data being carried by this packet (called a TCP segment), while the acknowledgement number identifies the next byte that the sender of this packet is expecting.[3]

1.3 Throughput of the Stop-And-Wait protocol

The Stop-And-Wait protocol sends a packet, waits for an acknowledgement, and then sends the next packet. So in every "round," the Stop-And-Wait sender transmits a single packet. Assuming losses are relatively rare, each round takes up time equal to the round-trip time (RTT), i.e., the time taken to send a single packet from the sender to the receiver plus the time taken for the receiver to send an acknowledgement back to the sender.

In general, the RTT consists of several components: (1) the propagation delay from the sender to the receiver plus the propagation delay from the receiver to the sender, (2) any queueing delay incurred along the path from the sender to the receiver depending on how many other applications happen to traverse routers along the same path, and (3) the transmission delay required to transmit packets and acks of a finite size on a link with a finite capacity. The RTT does not include the delay of retransmitting data because each retransmission corresponds to a new packet and each RTT measurement pertains to a single packet—not multiple packets. The RTT also does not include application-layer delays because the RTT measurement starts after a packet

[3]Recall that TCP communication is bidirectional, so the data packet in one direction also serves as the ACK packet in the other direction.

has been created using data supplied by the application. We will occasionally use the term RTT_{min} to denote the lowest possible RTT, i.e., the RTT when the queueing delay is 0. This represents a lower bound on the RTT that can actually be realized in practice.

This means that the Stop-And-Wait protocol sends roughly one packet every RTT, which is a throughput of 1/RTT packets per second.[4] Let's plug in some numbers to understand this throughput better. Your WiFi link can carry 1460 bytes of TCP data in each packet. Let's assume the round-trip time between here and California is about 100 ms. Then the throughput of the Stop-And-Wait protocol will be 1460 bytes every 100 ms or about 116 kbit/s. For comparison, your WiFi links provide at least 10 Mbit/s of capacity to most points on the Internet. So if you use the Stop-And-Wait protocol, you will only be using about 1% of the link's capacity.

Clearly, we should try to do better. In the next lecture, we'll look at a better reliable transport protocol called the sliding window protocol, which provides much better throughput than the Stop-And-Wait protocol. The basic idea in the sliding window protocol is to start transmitting the second, third, and subsequent packets while still waiting for the acknowledgement of the first packet. In other words, instead of permitting only one unacknowledged packet at a time, we allow multiple unacknowledged packets at a time. For now though, we'll return to the Stop-And-Wait protocol and look at the question of deciding when to declare packets lost and retransmit them.

2 Retransmission timers

2.1 Deciding when a packet has been lost

Suppose the sender transmits a packet to the receiver. It then starts a retransmission timer. When this timer times out, the sender decides the packet or its acknowledgement has been lost and retransmits the packet. What should the timeout of this timer be so that the sender can be reasonably confident that the data packet (or its acknowledgement packet) has been lost and hence needs to be retransmitted? Clearly, the sender must at least wait for the current round-trip time (RTT) between the sender and the receiver. For instance, if it takes at least 100 ms for a packet to get from NYC to LA, there is no point retransmitting after 1 ms; the receiver will then just end up with multiple copies of the same data packet, which would waste the link's capacity.

So we want to wait at least the RTT and maybe a little more. How much more? RTTs can vary for a number of reasons: queueing delays in the network, changes in the path between the sender and the receiver, etc. If we treat the RTT as a random variable drawn from a probability distribution instead of a single fixed value, we want to pick a high enough timeout such that the probability of the RTT random variable exceeding that threshold (also known as the tail probability) is really small.

One way to pick a high enough timeout with low tail probability is to estimate the mean (μ) and the standard deviation (σ) of the RTT random variable distribution and set the timeout to $\mu + k.\sigma$, where k is some constant that reflects how low we want the tail probability to be. For instance, if the RTT random variable is normally distributed, then the probability that an random RTT sample is within one standard deviation ($k = 1$) is 68%, two standard deviations ($k = 2$) is 95%, and three standard deviations ($k = 3$) is 99.7%. Correspondingly, the tail probabilities are 32%, 5%, and 0.3%. Real RTTs are not normally distributed, but most probability distributions share this property of the tail probability decreasing sharply with increasing k, even though they don't decrease as sharply as the normal distribution.[5]

Hence, picking a small value of k is sufficient. TCP's own retransmission timer uses a $k = 4$, and we'll use that for concreteness. Assuming we have some estimate of μ and σ, we can calculate the timeout as:

$$timeout \leftarrow \mu + 4.\sigma \tag{1}$$

[4] A more detailed analysis would take into account the fact that retransmitted packets need multiple round trips before the round is completed, but wouldn't change the overall expression for throughput by too much.

[5] The formal result is called "Chebyshev's inequality." It's not something you are expected to know for this course. All you should remember is that the tail probability decreases rapidly with increasing k.

2.2 Online estimation of mean RTT

Ideally, we would be able to estimate the mean RTT (μ above) by collecting enough RTT samples offline and fitting them to some distribution that captures the distribution of RTTs. Unfortunately, this estimate would be rendered useless the instant network conditions changed, e.g., a new application starts sharing the same queue as the Stop-And-Wait sender, which leads to an increase in queueing delays. Alternatively, a router outage could result in a different path with a different RTT from the sender to the receiver.

The solution is to estimate the mean RTT in an online manner. Because the estimation is done online and happens on every packet, this estimator has to use a very small number of computations. The estimator that TCP uses is called an exponentially weighted moving average filter (EWMA). The EWMA is updated on every new acknowledgement received at the sender as follows.

$$\mu \leftarrow (1-\alpha).\mu + \alpha.RTT \tag{2}$$

The equation updates the RTT mean estimate, μ, to a weighted combination of its previous estimate μ and the RTT sample obtained from the current acknowledgement. The weights used are $(1-\alpha)$ and α for the previous estimate and the current RTT sample respectively. The RTT sample is obtained by adding a new packet header that carries the timestamp at which the sender transmitted the packet. This header is then echoed back in the acknowledgement. When the acknowledgement is received back at the sender, the sender subtracts the timestamp header from the current wall-clock time to determine the RTT. In TCP, this header is called the TCP timestamp header field.

But what is the justification for this EWMA-based estimator? The EWMA can be recursively expanded using the equation above to get the equation below:

$$\mu \leftarrow \alpha.RTT + \alpha.(1-\alpha).RTT_{-1} + \alpha.(1-\alpha)^2.RTT_{-2} + \alpha.(1-\alpha)^3.RTT_{-3} + \ldots \tag{3}$$

Here RTT is the RTT of the current acknowledgement, RTT_{-1} is the RTT of the previous acknowledgement, RTT_{-2} is the RTT of the acknowledgement from two samples ago, and so on. This estimator exponentially decays the contribution of samples further back in the past (hence the name) and gives greater weight to more recent samples, allowing it to more quickly react to changes in RTT caused by changing network conditions.

Further, the EWMA closely tracks the average of samples. For instance, if you start the EWMA from some arbitrary initial value and present it with a set of RTT samples that all have the same constant value C, the EWMA will quickly converge to the value C.

How quickly depends on the value of α. A large α weighs recent samples more aggressively, which allows it to converge faster to the new value. But, it also means that it could react too soon to a transient change in network conditions. On the other hand, a smaller α is more sluggish to react to changing conditions, but is more robust against transient changes. TCP implementations on the Internet use an α of 0.125, which empirically seems to balance these two concerns.

2.3 Online estimation of RTT standard deviation

Finally, we use a similar online EWMA for the standard deviation, σ, as well.

$$\sigma \leftarrow (1-\beta).\sigma + \beta.|RTT - \mu| \tag{4}$$

Here, the online version of σ is a linear combination of the previous standard deviation and the absolute deviation ($|RTT - \mu|$), which is easier to calculate in an online manner than the standard deviation.

2.4 Handling lost retransmissions

It is possible that the retransmitted packet might again be deemed lost in the network, which might be a symptom that the timeout estimates are much lower than the actual RTTs in the network. This can happen at the very beginning, when the timeout estimates are set to some default value or when the network conditions change drastically because of a large number of applications suddenly using the network.

When this happens, the retransmission timer *exponentially backs off*: it repeatedly doubles its estimate of the timeout until some retransmitted packet is acknowledged successfully. This allows the sender to restart its μ, σ, and $timeout$ calculations. Exponential back off is a good way to be extremely conservative when conditions are unknown and we'll see it later in the course in the context of link-layer protocols.

A Difference between TCP acks and our acks.

There is one important difference in semantics between TCP acknowledgements and the Stop-And-Wait acknowledgements as we have presented them here. A Stop-And-Wait receiver's acknowledgement with the sequence number n indicates that the packet with sequence number n has been received. TCP's acknowledgements on the other hand are *cumulative*: a TCP acknowledgement with the sequence number n indicates that *all* bytes with sequence number less than or equal to n have been received.