MIT 6.02 **DRAFT** *Lecture Notes*
Last update: November 3, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 15
Sharing a Channel: Media Access (MAC) Protocols

There are many communication channels, including radio and acoustic channels, and certain kinds of wired links (coaxial cables), where multiple nodes can all be connected and hear each other's transmissions (either perfectly or with some non-zero probability). This chapter addresses the fundamental question of how such a common communication channel—also called a *shared medium*—can be shared between the different nodes.

There are two fundamental ways of sharing such channels (or media): *time sharing* and *frequency sharing*.[1] The idea in time sharing is to have the nodes coordinate with each other to divide up the access to the medium one at a time, in some fashion. The idea in frequency sharing is to divide up the frequency range available between the different transmitting nodes in a way that there is little or no interference between concurrently transmitting nodes. The methods used here are the same as in frequency division multiplexing, which we described in the previous chapter.

This chapter focuses on time sharing. We will investigate two common ways: *time division multiple access*, or *TDMA*, and *contention protocols*. Both approaches are used in networks today.

These schemes for time and frequency sharing are usually implemented as *communication protocols*. The term *protocol* refers to the rules that govern what each node is allowed to do and how it should operate. Protocols capture the "rules of engagement" that nodes must follow, so that they can collectively obtain good performance. Because these sharing schemes define how multiple nodes should control their access to a shared medium, they are termed *media access (MAC) protocols* or *multiple access protocols*.

Of particular interest to us are contention protocols, so called because the nodes *contend* with each other for the medium without pre-arranging a schedule that determines who should transmit when, or a frequency reservation that guarantees little or no interference. These protocols operate in *laissez faire* fashion: nodes get to send according to their own

[1] There are other ways too, involving codes that allow multiple concurrent transmissions in the same frequency band, with mechanisms to decode the individual communications. We won't study these more advanced ideas here. These ideas are sometimes used in practice, but all real-world systems use a combination of time and frequency sharing.

Figure 15-1: The locations of some of the Alohanet's original ground stations are shown in light blue markers.

volition without any external agent telling them what to do. These contention protocols are well-suited for *data* networks, which are characterized by nodes transmitting data in bursts and at variable rates (we will describe the properties of data networks in more detail in a later chapter on packet switching).

In this chapter and the subsequent ones, we will assume that any message is broken up into a set of one or more packets, and a node attempts to send each packet separately over the shared medium.

■ 15.1 Examples of Shared Media

Satellite communications. Perhaps the first example of a shared-medium network deployed for data communication was a satellite network: the *Alohanet* in Hawaii. The Alohanet was designed by a team led by Norm Abramson in the 1960s at the University of Hawaii as a way to connect computers in the different islands together (Figure 15-1). A computer on the satellite functioned as a *switch* to provide connectivity between the nodes on the islands; any packet between the islands had to be first sent over the *uplink* to the switch,[2] and from there over the *downlink* to the desired destination. Both directions used radio communication and the medium was shared. Eventually, this satellite network was connected to the ARPANET (the precursor to today's Internet).

Such satellite networks continue to be used today in various parts of the world, and they are perhaps the most common (though expensive) way to obtain connectivity in the high seas and other remote regions of the world. Figure 15-2 shows the schematic of such a network connecting islands in the Pacific Ocean and used for teleconferencing.

In these satellite networks, the downlink usually runs over a different frequency band from the uplinks, which all share the same frequency band. The different uplinks, however, need to be shared by different concurrent communications from the ground stations to the satellite.

[2]We will study switches in more detail in later lectures.

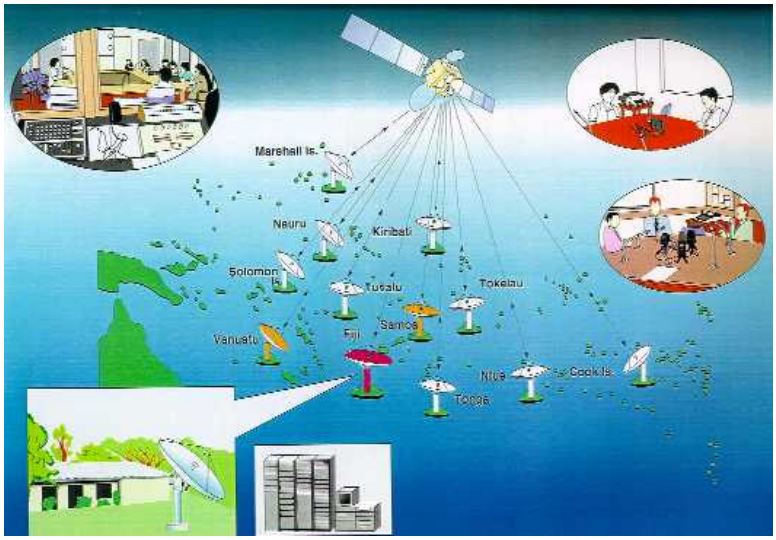


Figure 15-2: A satellite network. The "uplinks" from the ground stations to the satellite form a shared medium. (Picture from `http://vidconf.net/`)

Wireless networks. The most common example of a shared communication medium today, and one that is only increasing in popularity, uses radio. Examples include cellular wireless networks (including standards like EDGE, 3G, and 4G), wireless LANs (such as 802.11, the WiFi standard), and various other forms of radio-based communication. Another example of a communication medium with similar properties is the acoustic channel explored in the 6.02 labs. Broadcast is an inherent property of radio and acoustic communication, especially with so-called omni-directional antennas, which radiate energy in all (or many) different directions. However, radio and acoustic broadcasts are not perfect because of interference and the presence of obstacles on certain paths, so different nodes may correctly receive different parts of any given transmission. This reception is *probabilistic* and the underlying random processes that generate bit errors are hard to model.

Shared bus networks. An example of a wired shared medium is Ethernet, which when it was first developed (and for many years after) used a shared cable to which multiple nodes could be connected. Any packet sent over the Ethernet could be heard by all stations connected physically to the network, forming a perfect shared broadcast medium. If two or more nodes send packets that overlap in time, both packets ended up being garbled and received in error.

Over-the-air radio and television. Even before data communication, many countries in the world had (and still have) radio and television broadcast stations. Here, a relatively small number of transmitters share a frequency range to deliver radio or television content. Because each station was assumed to be active most of the time, the natural approach to sharing is to divide up the frequency range into smaller sub-ranges and allocate each sub-range to a station (frequency division multiplexing).

Given the practical significance of these examples, and the sea change in network access brought about by wireless technologies, developing methods to share a common medium

is an important problem.

15.2 Model and Goals

Before diving into the protocols, let's first develop a simple abstraction for the shared medium and more rigorously model the problem we're trying to solve. This abstraction is a reasonable first-order approximation of reality.

We are given a set of N nodes sharing a communication medium. We will assume N is fixed, but the protocols we develop will either continue to work when N varies, or can be made to work with some more effort. Depending on the context, the N nodes may or may not be able to hear each other; in some cases, they may not be able to at all, in some cases, they may, with some probability, and in some cases, they will always hear each other. Each node has some source of data that produces packets. Each packet may be destined for some other node in the network. For now, we will assume that every node has packets destined to one given "master" node in the network. Of course, the master must be capable of hearing every other node, and receiving packets from those nodes. We will assume that the master perfectly receives packets from each node as long as there are no "collisions" (we explain what a "collision" is below).

The model we consider has the following rules:

1. Time is divided into slots of equal length, τ.
2. Each node can send a packet only at the beginning of a slot.
3. All packets are of the same size, and equal to an integral multiple of the *slot length*. In practice, packets will of course be of varying lengths, but this assumption simplifies our analysis and does not affect the correctness of any of the protocols we study.
4. Packets arrive for transmission according to some random process; the protocol should work correctly regardless of the process governing packet arrivals. If two or more nodes send a packet in the same time slot, they are said to *collide*, and *none* of the packets are received successfully. Note that even if only part of a packet encounters a collision, the entire packet is assumed to be lost. This "perfect collision" assumption is an accurate model for wired shared media like Ethernet, but is only a crude approximation of wireless (radio) communication. The reason is that it might be possible for multiple nodes to concurrently transmit data over radio, and depending on the positions of the receivers and the techniques used to decode packets, for the concurrent transmissions to be received successfully.
5. The sending node can discover that a packet transmission collided and may choose to retransmit such a packet.
6. Each node has a queue; any packets waiting to be sent are in the queue. A node with a non-empty queue is said to be *backlogged*.

Performance goals. An important goal is to provide high **throughput**, i.e., to deliver packets successfully at as high a rate as possible, as measured in bits per second. A mea-

sure of throughput that is independent of the rate of the channel is the **utilization**, which is defined as follows:

Definition. The **utilization** that a protocol achieves is defined as the **ratio of the total throughput to the maximum data rate of the channel**.

For example, if there are 4 nodes sharing a channel whose maximum bit rate is 10 Megabits/s,[3] and they get throughputs of 1, 2, 2, and 3 Megabits/s, then the utilization is $(1+2+2+3)/10 = 0.8$. Obviously, the utilization is always between 0 and 1. Note that the utilization may be smaller than 1 either because the nodes have enough offered load and the protocol is inefficient, *or* because there isn't enough offered load. By *offered load*, we mean the load presented to the network by a node, or the aggregate load presented to the network by all the nodes. It is measured in bits per second as well.

But utilization alone isn't sufficient: we need to worry about **fairness** as well. If we weren't concerned about fairness, the problem would be quite easy because we could arrange for a particular backlogged node to always send data. If all nodes have enough load to offer to the network, this approach would get high utilization. But it isn't too useful in practice because it would also starve one or more other nodes.

A number of notions of fairness have been developed in the literature, and it's a topic that continues to generate activity and interest. For our purposes, we will use a simple, standard definition of fairness: we will measure the throughput achieved by each node over some time period, T, and say that an allocation with lower standard deviation is "fairer" than one with higher standard deviation. Of course, we want the notion to work properly when the number of nodes varies, so some normalization is needed. We will use the following simplified *fairness index*:

$$F = \frac{(\sum_{i=1}^{N} x_i)^2}{N \sum x_i^2}, \tag{15.1}$$

where x_i is the throughput achieved by node i and there are N backlogged nodes in all.

Clearly, $1/N \leq F \leq 1$; $F = 1/N$ implies that a single node gets all the throughput, while $F = 1$ implies perfect fairness. We will consider fairness over both the long-term (many thousands of "time slots") and over the short term (tens of slots). It will turn out that in the schemes we study, some schemes will achieve high utilization but poor fairness, and that as we improve fairness, the overall utilization will drop.

The next section discusses Time Division Multiple Access, or TDMA, a scheme that achieves high fairness, but whose utilization may be low when the offered load is non-uniform between the nodes, and is not easy to implement in a fully distributed way without a central coordinator when nodes join and leave dynamically. However, there are practical situations when TDMA works well, and such protocols are used in some cellular wireless networks. Then, we will discuss a variant of the *Aloha* protocol, the first contention MAC protocol that was invented. Aloha forms the basis for many widely used contention protocols, including the ones used in the IEEE 802.11 (WiFi) standard.

[3] In this course, and in most, if not all, of the networking and communications world, "kilo" = 10^3, "mega" = 10^6 and "giga" = 10^9, when talking about network rates, speeds, or throughput. When referring to storage units, however, one needs to be more careful because "kilo", "mega" and "giga" often (but not always) refer to 2^{10}, 2^{20}, and 2^{30}, respectively.

15.3 Time Division Multiple Access (TDMA)

If one had a centralized resource allocator, such as a base station in a cellular network, and a way to ensure some sort of time synchronization between nodes, then a "time division" is not hard to develop. As the name suggests, the goal is to divide time evenly between the N nodes. One way to achieve this goal is to divide time into slots starting from 0 and incrementing by 1, and for each node to be given a unique identifier (ID) in the range $[0, N-1]$.

A simple TDMA protocol uses the following rule:

> If the current time slot is t, then the node with ID i transmits if, and only if, it is backlogged and $t \text{mod} N = i$.

If the node whose turn it is to transmit in time slot t is not backlogged, then that time slot is "wasted".

This TDMA scheme has some good properties. First, it is fair: each node gets the same number of transmission attempts because the protocol provides access to the medium in round-robin fashion among the nodes. The protocol also incurs no packet collisions (assuming it is correctly implemented!): exactly one node is allowed to transmit in any time slot. And if the number of nodes is static, and there is a central coordinator (e.g., a master nodes), this TDMA protocol is simple to implement.

This TDMA protocol does have some drawbacks. First and foremost, if the nodes send data in bursts, alternating between periods when they are backlogged and when they are not, or if the amount of data sent by each node is different, then TDMA under-utilizes the medium. The degree of under-utilization depends on how skewed the traffic pattern; the more the imbalance, the lower the utilization. An "ideal" TDMA scheme would provide equal access to the medium only among currently backlogged nodes, but even in a system with a central master, knowing which nodes are currently backlogged is somewhat challenging. Second, if each node sends packets that are of different sizes (as is the case in practice, though the model we specified above did not have this wrinkle), making TDMA work correctly is more involved. It can still be made to work, but it takes more effort. An important special case is when each node sends packets of the same size, but the size is bigger than a single time slot. This case is not hard to handle, though it requires a little more thinking, and is left as an exercise for the reader.) Third, making TDMA work in a fully distributed way in a system without a central master, and in cases when the number of nodes changes dynamically, is tricky. It can be done, but the protocol quickly becomes more complex than the simple rule stated above.

Contention protocols like Aloha and CSMA don't suffer from these problems, but unlike TDMA, they encounter packet collisions. In general, burst data and skewed workloads favor contention protocols over TDMA. The intuition in these protocols is that we somehow would like to allocate access to the medium fairly, but only among the *backlogged* nodes. Unfortunately, only each node knows with certainty if it is backlogged or not. Our solution is to use *randomization*, a simple but extremely powerful idea; if each backlogged node transmits data with some probability, perhaps we can arrange for the nodes to pick their transmission probabilities to engineer an outcome that has reasonable utilization (throughput) and fairness!

The rest of this chapter describes such randomized contention protocols, starting with

the ancestor of them all, Aloha.

15.4 Aloha

The basic variant of the Aloha protocol that we're going to start with is simple, and as follows:

If a node is backlogged, it sends a packet from its queue with probability p.

From here, until Section 15.6, we will assume that each packet is exactly one slot in length. Such a system is also called **slotted Aloha**.

We have not specified what p is; we will figure that out later, once we analyze the protocol as a function of p. Suppose there are N backlogged nodes and each node uses the same value of p. We can then calculate the utilization of the shared medium as a function of N and p by simply counting the number of slots in which *exactly one node sends a packet*. By definition, a slot with 0 or greater than 1 transmissions does not correspond to a successfully delivered packet, and therefore does not contribute toward the utilization.

If each node sends with probability p, then the probability that exactly one node sends in any given slot is $Np(1-p)^{N-1}$. The reason is that the probability that a specific node sends in the time slot is p, and for its transmission to be successful, all the other nodes should not send. That combined probability is $p(1-p)^{N-1}$. Now, we can pick the successfully transmitting node in N ways, so the probability of exactly one node sending in a slot is $Np(1-p)^{N-1}$.

This quantity is the utilization achieved by the protocol because it is the fraction of slots that count toward useful throughput. Hence,

$$U_{\text{Slotted Aloha}}(p) = Np(1-p)^{N-1}. \tag{15.2}$$

Figure 15-3 shows Eq.(15.2) for $N = 8$ as a function of p. The maximum value of U occurs when $p = 1/N$, and is equal to $(1 - \frac{1}{N})^{N-1}$. As $N \to \infty, U \to 1/e \approx 37\%$.[4] This result is an important one: the maximum utilization of slotted Aloha for a large number of backlogged nodes is roughly $1/e$.

37% might seem like a small value (after all, the majority of the slots are being wasted), but notice that the protocol is *extremely simple* and has the virtue that it is hard to botch its implementation! It is fully distributed and requires no coordination or other specific communication between the nodes. That simplicity in system design is worth a lot—oftentimes, it's a very good idea to trade simplicity off for high performance, and worry about optimization only when a specific part of the system is likely to become (or already has become) a bottleneck.

That said, the protocol as described thus far requires a way to set p. Ideally, if each node knew the value of N, setting $p = 1/N$ achieves the maximum. Unfortunately, this isn't as simple as it sounds because N here is the number of *backlogged* nodes that currently have data in their queues. The question then is: how can the nodes pick the best p? We

[4]Here, we use the fact that $\lim_{N\to\infty}(1-1/N)^N = 1/e$. To see why this limit holds, expand the log of the left hand side using a Taylor series: $\log(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \ldots$. for $|x| < 1$.

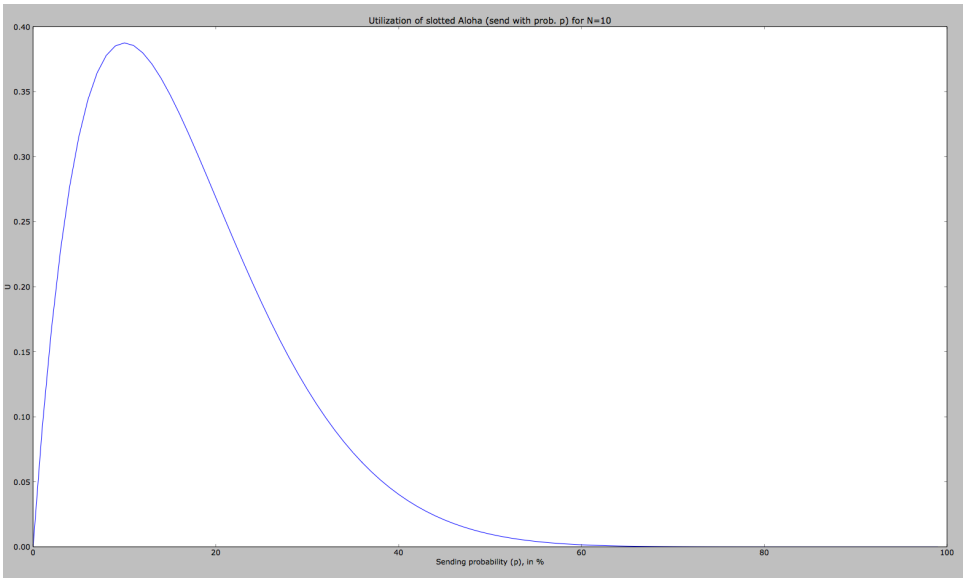


Figure 15-3: The utilization of slotted Aloha as a function of p for $N = 10$. The maximum occurs at $p = 1/N$ and the maximum utilization is $U = (1 - \frac{1}{N})^{N-1}$. As $N \to \infty$, $U \to \frac{1}{e} \approx 37\%$. N doesn't have to be particularly large for the $1/e$ approximation to be close—for instance, when $N = 10$, the maximum utilization is 0.387.

turn to this important question next, because without such a mechanism, the protocol is impractical.

■ 15.5 Stabilizing Aloha: Binary Exponential Backoff

We use a special term for the process of picking a good "p" in Aloha: **stabilization**. In general, in distributed protocols and algorithms, "stabilization" refers to the process by which the method operates around or at a desired operating point. In our case, the desired operating point is around $p = 1/N$, where N is the number of backlogged nodes.

Stabilizing a protocol like Aloha is a difficult problem because the nodes may not be able to directly communicate with each other (or even if they could, the overhead involved in doing so would be significant). Moreover, each node has bursty demands for the medium, and the set of backlogged nodes could change quite rapidly with time. What we need is a "search procedure" by which each node converges toward the best "p".

Fortunately, this search for the right p can be guided by feedback: whether a given packet transmission has been successful or not is invaluable information. In practice, this feedback may be obtained either using an acknowledgment for each received packet from the receiver (as in most wireless networks) or using the ability to directly detect a collision by listening on one's own transmission (as in wired Ethernet). In either case, the feedback has the same form: "yes" or "no", depending on whether the packet was received successfully or not.

Given this feedback, our stabilization strategy at each node is conceptually simple:

1. Maintain the current estimate of p, p_{est}, initialized to some value. (We will talk about initialization later.)

2. If "no", then consider decreasing p.

3. If "yes", then consider increasing p.

This simple-looking structure is at the core of a wide range of distributed network protocols that seek to operate around some desired or optimum value. The devil, of course, is in the details, in that the way in which the increase and decrease rules work depend on the problem and dynamics at hand.

Let's first talk about the decrease rule for our protocol. The intuition here is that because there was a collision, it's likely that the node's current estimate of the best p is too high (equivalently, its view of the number of backlogged nodes is too small). Since the actual number of nodes could be quite a bit larger, a good strategy that quickly gets to the true value is *multiplicative decrease*: reduce p by a factor of 2. Akin to binary search, this method can reach the true probability within a logarithmic number of steps from the current value; absent any other information, it is also the most efficient way to do so.

Thus, the decrease rule is:

$$p \leftarrow p/2 \tag{15.3}$$

This multiplicative decrease scheme has a special name: *binary exponential backoff*. The reason for this name is that if a packet has been unsuccessful k times, the probability with which it is sent decays proportional to 2^{-k}. The "2" is the "binary" part, the k in the exponent is the "exponential" part, and the "backoff" is what the sender is doing in the face of these failures.

To develop an increase rule upon a successful transmission, observe that two factors must be considered: first, the estimate of the number of other backlogged nodes whose queues might have emptied during the time it took us to send our packet successfully, and second, the potential waste of slots that might occur if the increased value of p is too small. In general, if n backlogged nodes contended with a given node x, and x eventually sent its packet, we can expect that some fraction of the n nodes also got their packets through. Hence, the increase in p should at least be multiplicative. p_{max} is a parameter picked by the protocol designer, and must not exceed 1 (obviously).

Thus, one possible increase rule is:

$$p \leftarrow min(2p, p_{\text{max}}). \tag{15.4}$$

Another possible rule is even simpler:

$$p \leftarrow p_{\text{max}}. \tag{15.5}$$

The second rule above isn't unreasonable; in fact, under burst traffic arrivals, it is quite possible for a much smaller number of other nodes to continue to remain backlogged, and in that case resetting to a fixed maximum probability would be a good idea.

For now, let's assume that $p_{\text{max}} = 1$ and use (15.4) to explore the performance of the protocol; one can obtain similar results with (15.5) as well.

15.5.1 Performance

Let's look at how this protocol works in simulation using WSim, a shared medium simulator that you will use in the lab. Running a randomized simulation with $N = 6$ nodes, each generating traffic in a random fashion in such a way that in most slots many of the nodes are backlogged, we see the following result:

```
  Node 0 attempts 335 success 196 coll 139
  Node 1 attempts 1691 success 1323 coll 367
  Node 2 attempts 1678 success 1294 coll 384
  Node 3 attempts 114 success 55 coll 59
  Node 4 attempts 866 success 603 coll 263
  Node 5 attempts 1670 success 1181 coll 489
Time 10000 attempts 6354 success 4652 util 0.47
Inter-node fairness: 0.69
```

Each line starting with "Node" above says what the total number of transmission `attempts` from the specified node was, how many of them were `success`es, and how many of them were `coll`isions. The line starting with "Time" says what the total number of simulated time slots was, and the total number of packet attempts, successful packets (i.e., those without collisions), and the utilization. The last line lists the fairness.

A fairness of 0.69 with six nodes is actually quite poor (in fact, even a value of 0.8 would be considered poor for $N = 6$). Figure 15-4 shows two rows of dots for each node; the top row corresponds to successful transmissions while the bottom one corresponds to collisions. The bar graph in the bottom panel is each node's throughput. Observe how nodes 3 and 0 get very low throughput compared to the other nodes, a sign of significant *long-term unfairness*. In addition, for each node there are long periods of time when both nodes send no packets, because each collision causes their transmission probability to reduce by two, and pretty soon both nodes are made to starve, unable to extricate themselves from this situation. Such "bad luck" tends to happen often because a node that has backed off heavily is competing against a successful backlogged node whose p is a lot higher; hence, the "rich get richer".

How can we overcome this fairness problem? One approach is to set a lower bound on p, something that's a lot smaller than the reciprocal of the largest number of backlogged nodes we expect in the network. In most networks, one can assume such a quantity; for example, we might set the lower bound to 1/128 or 1/1024.

Setting such a bound greatly reduces the long-term unfairness (Figure 15-5) and the corresponding simulation output is as follows:

```
  Node 0 attempts 1516 success 1214 coll 302
  Node 1 attempts 1237 success 964 coll 273
  Node 2 attempts 1433 success 1218 coll 215
  Node 3 attempts 1496 success 1207 coll 289
  Node 4 attempts 1616 success 1368 coll 248
  Node 5 attempts 1370 success 1115 coll 254
Time 10000 attempts 8668 success 7086 util 0.71
Inter-node fairness: 0.99
```

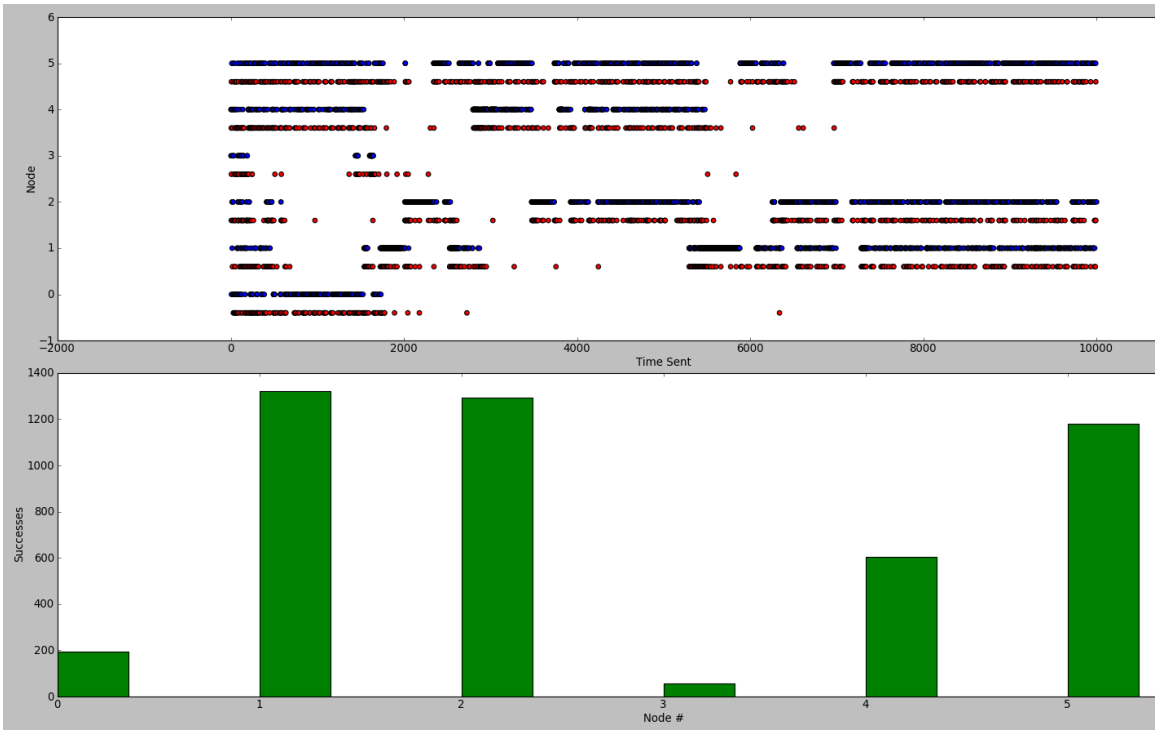


Figure 15-4: For each node, the top row (blue) shows the times at which the node successfully sent a packet, while the bottom row (red) shows collisions. Observe how nodes 3 and 0 are both clobbered getting almost no throughput compared to the other nodes. The reason is that both nodes end up with repeated collisions, and on each collision the probability of transmitting a packet reduces by 2, so pretty soon both nodes are completely shut out. The bottom panel is a bar graph of each node's throughput.

The careful reader will notice something fishy about the simulation output shown above (and also in the output from the simulation where we didn't set a lower bound on p): the reported utilization is 0.71, considerably higher than the "theoretical maximum" of $(1 - 1/N)^{N-1} = 0.4$ when $N = 6$. What's going on here is more apparent from Figure 15-5, which shows that there are long periods of time where any given node, though backlogged, does not get to transmit. Over time, every node in the experiment encounters times when it is starving, though over time the nodes all get the same share of the medium (fairness is 0.99). If p_{max} is 1 (or close to 1), then a backlogged node that has just succeeded in transmitting its packet will continue to send, while other nodes with smaller values of p end up backing off. This phenomenon is also sometimes called the *capture effect*, manifested by unfairness over time-scales on the order several packets. This behavior is not desirable.

Setting p_{max} to a more reasonable value (less than 1) yields the following:[5]

```
Node 0 attempts 941 success 534 coll 407
Node 1 attempts 1153 success 637 coll 516
Node 2 attempts 1076 success 576 coll 500
Node 3 attempts 1471 success 862 coll 609
```

[5]We have intentionally left the value unspecified because you will investigate how to set it in the lab.

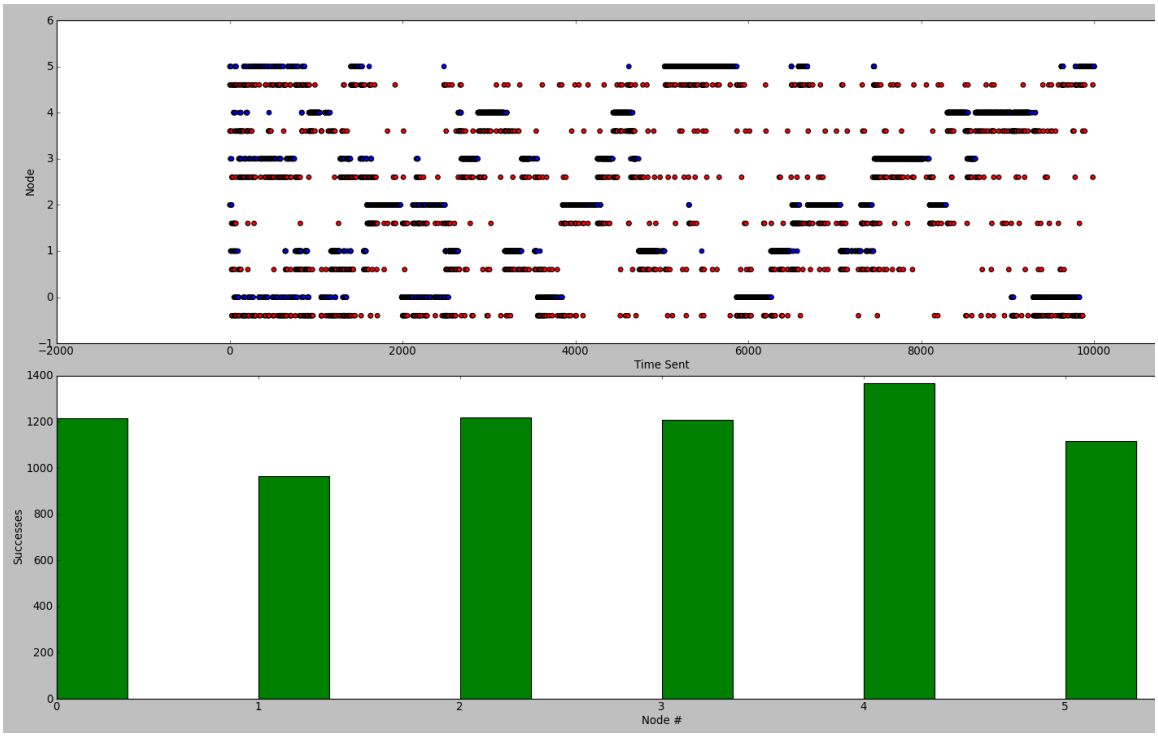


Figure 15-5: Node transmissions and collisions when backlogged v. slot index and each node's throughput (bottom row) when we set a lower bound on each backlogged node's transmission probability. Note the "capture effect" when some nodes hog the medium for extended periods of time, starving others. Over time, however, every node gets the same throughput (fairness is 0.99), but the long periods of inactivity while backlogged is undesirable.

```
  Node 4 attempts 1348 success 780 coll 568
  Node 5 attempts 1166 success 683 coll 483
Time 10000 attempts 7155 success 4072 util 0.41
Inter-node fairness: 0.97
```

Figure 15-6 shows the corresponding plot, which has reasonable per-node fairness over both long and short time-scales. The utilization is also close to the value we calculated analytically of $(1 - 1/N)^{N-1}$. Even though the utilization is now lower, the overall result is better because all backlogged nodes get equal share of the medium even over short time scales.

These experiments show the trade-off between achieving both good utilization and ensuring fairness. If our goal were only the former, the problem would be trivial: starve all but one of the backlogged nodes. Achieving a good balance between various notions of fairness and network utilization (throughput) is at the core of many network protocol designs.

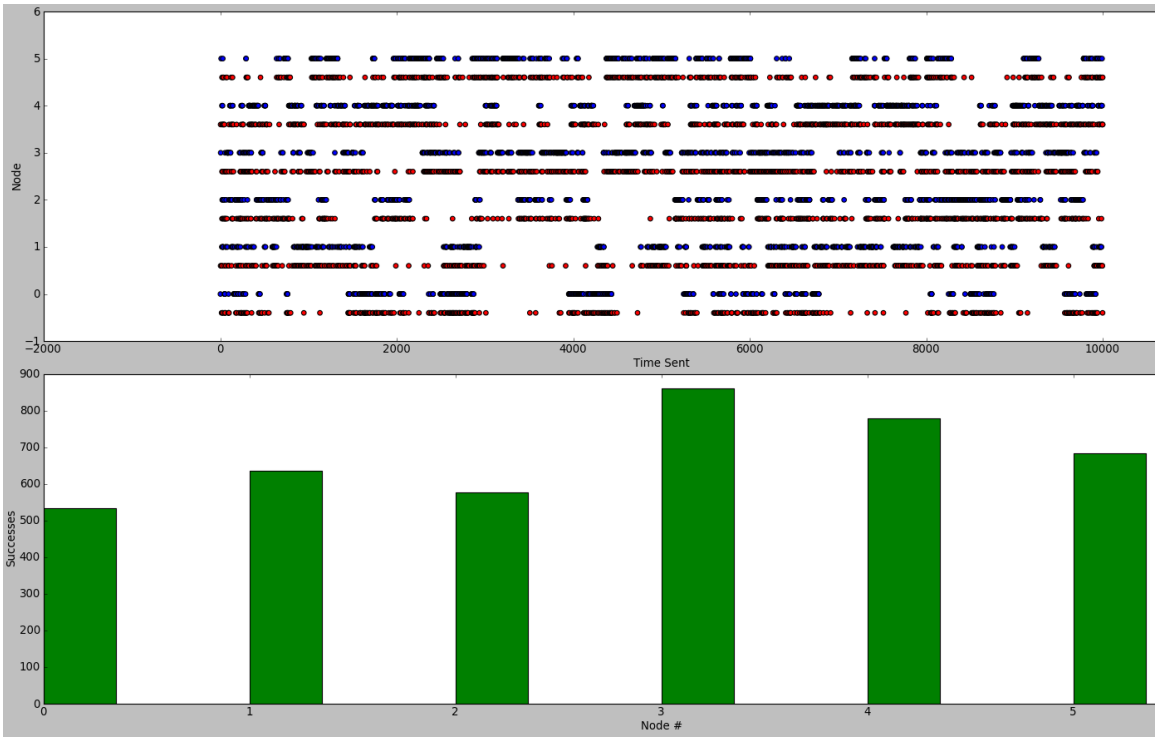


Figure 15-6: Node transmissions and collisions when we set both lower and upper bounds on each backlogged node's transmission probability. Notice that the capture effect is no longer present. The bottom panel is each node's throughput.

■ 15.6 Generalizing to Bigger Packets, and "Unslotted" Aloha

So far, we have looked at perfectly slotted Aloha, which assumes that each packet fits exactly into a slot. But what happens when packets are bigger than a single slot? In fact, one might even ask why we need slotting. What happens when nodes just transmit without regard to slot boundaries? In this section, we analyze these issues, starting with packets that span multiple slot lengths. Then, by making a slot length much smaller than a single packet size, we can calculate the utilization of the Aloha protocol where nodes can send without concern for slot boundaries—that variant is also called **unslotted Aloha**.

Note that the pure unslotted Aloha model is one where there are no slots at all, and each node can send a packet any time it wants. However, this model may be approximated by a model where a node sends a packet only at the beginning of a time slot, but each packet is many slots long. When we make the size of a packet large compared to the length of a single slot, we get the unslotted case. We will abuse terminology slightly and use the term *unslotted Aloha* to refer to the case when there are slots, but the packet size is large compared to the slot time.

Suppose each node sends a packet of size T slots. One can then work out the probability of a successful transmission in a network with N backlogged nodes, each attempting to send its packet with probability p whenever it is not already sending a packet. The key insight here is that *any packet whose transmission starts in* $2T - 1$ *slots that have any overlap with the current packet can collide.* Figure 15-7 illustrates this point, which we discuss in

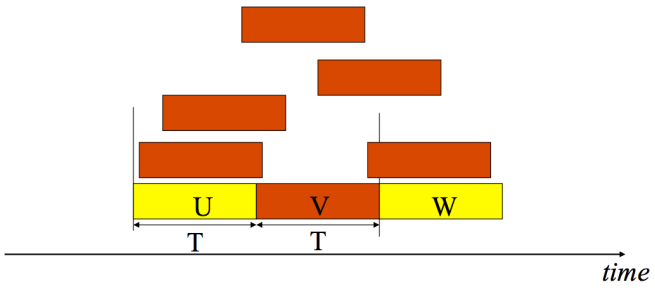


Figure 15-7: Each packet is T slots long. Packet transmissions begin at a slot boundary. In this picture, every packet except U and W collide with V. Given packet V, any other packet sent in any one of $2T-1$ slots—the T slots of V as well as the $T-1$ slots immediately preceding V's transmission—collide with V.

more detail next.

Suppose that some node sends a packet in some slot. What is the probability that this transmission has no collisions? From Figure 15-7, for this packet to not collide, no other node should start its transmission in $2T-1$ slots. Because p is the probability of a backlogged node sending a packet in a slot, and there are $N-1$ nodes, this probability is equal to $(1-p)^{(2T-1)(N-1)}$. (There is a bit of an inaccuracy in this expression, which doesn't make a significant material difference to our conclusions below, but which is worth pointing out. This expression assumes that a node sends packet independently in each time slot with probability p. Of course, in practice a node will not be able to send a packet in a time slot if it is sending a packet in the previous time slot, unless the packet being sent in the previous slot has completed. But our assumption in writing this formula is that such "self inteference" is permissible, which can't occur in reality. But it doesn't matter much for our conclusion because we are interested in the utilization when N is large, which means that p would be quite small. Moreover, this formula does represent an accurate *lower bound* on the throughput.)

Now, the transmitting node can be chosen in N ways, and the node has a probability p of sending a packet. Hence, the utilization, U, is equal to

$$\begin{aligned} U &= \text{Throughput/Maximum rate} \\ &= Np(1-p)^{(2T-1)(N-1)}/(1/T) \\ &= TNp(1-p)^{(2T-1)(N-1)}. \end{aligned} \tag{15.6}$$

For what value of p is U maximized, and what is the maximum value? By differentiating U wrt p and crunching through some algebra, we find that the maximum value, for large N, is $\frac{T}{(2T-1)e}$.

Now, we can look at what happens in the pure unslotted case, when nodes send without regard to slot boundaries. As explained above, the utilization of this scheme is identical to the case when we make the packet size T much larger than 1; i.e., if each packet is large compared to a time slot, then the fact that the model assumes that packets are sent along slot boundaries is irrelevant as far as throughput (utilization) is concerned. The maximum utilization in this case when N is large is therefore equal to $\frac{1}{2e} \approx 0.18$. **Note that this value is one-half of the maximum utilization of pure slotted Aloha where each packet is one**

slot long. (We're making this statement for the case when N is large, but it doesn't take N to become all that large for the statement to be roughly true, as we'll see in the lab.)

This result may be surprising at first glance, but it is intuitively quite pleasing. Slotting makes it so two packets destined to collide do so fully. Because partial collisions are just as bad as full ones in our model of the shared medium, forcing a full collision improves utilization. Unslotted Aloha has "twice the window of vulnerability" as slotted Aloha, and in the limit when the number of nodes is large, achieves only one-half the utilization.

■ 15.7 Carrier Sense Multiple Access (CSMA)

So far, we have assumed that no two nodes using the shared medium can hear each other. This assumption is true in some networks, notably the satellite network example mentioned here. Over a wired Ethernet, it is decidedly not true, while over wireless networks, the assumption is sometimes true and sometimes not (if there are three nodes A, B, and C, such that A and C can't usually hear each other, but B can usually hear both A and C, then A and C are said to be *hidden terminals*).

The ability to first *listen* on the medium before attempting a transmission can be used to reduce the number of collisions and improve utilization. The technical term given for this capability is called **carrier sense**: a node, before it attempts a transmission, can listen to the medium to see if the analog voltage or signal level is higher than if the medium were unused, or even attempt to detect if a packet transmission is in progress by processing ("demodulating", a concept we will see in later lectures) a set of samples. Then, if it determines that another packet transmission is in progress, it considers the medium to be *busy*, and *defers* its own transmission attempt until the node considers the medium to be *idle*. The idea is for a node to send only when it believes the medium to be idle.

One can modify the stabilized version of Aloha described above to use CSMA. One advantage of CSMA is that it no longer requires each packet to be one time slot long to achieve good utilization; packets can be larger than a slot duration, and can also vary in length.

Note, however, that in any practical implementation, it will takes some time for a node to detect that the medium is idle after the previous transmission ends, because it takes time to integrate the signal or sample information received and determine that the medium is indeed idle. This duration is called the *detection time* for the protocol. Moreover, multiple backlogged nodes might discover an "idle" medium at the same time; if they both send data, a collision ensues. For both these reasons, CSMA does not achieve 100% utilization, and needs a backoff scheme, though it usually achives higher utilization than stabilized slotted Aloha over a single shared medium. You will investigate this protocol in the lab.

■ 15.8 A Note on Implementation: Contention Windows

In the protocols described so far, each backlogged node sends a packet with probability p, and the job of the protocol is to adapt p in the best possible way. With CSMA, the idea is to send with this probability but only when the medium is idle. In practice, many contention protocols such as the IEEE 802.3 (Ethernet) and 802.11 (WiFi) standards do something a little different: rather than each node transmitting with a probability in each time slot,

they use the concept of a **contention window**.

A contention window scheme works as follows. Each node maintains its own current value of the window, which we call CW. CW can vary between CWmin and CWmax; CWmin may be 1 and CWmax may be a number like 1024. When a node decides to transmit, it does so by picking a random number r uniformly in $[1, \text{CW}]$ and sends in time slot $C + r$, where C is the current time slot. If a collision occurs, the node doubles CW; on a successful transmission, a node halves CW (or, as is often the case in practice, directly resets it to CWmin).

You should note that this scheme is similar to the one we studied and analyzed above. The doubling of CW is analogous to halving the transmission probability, and the halving of CW is analogous to doubling the probability (CW has a lower bound; the transmission probability has an upper bound). But there are two crucial differences:

1. Transmissions with a contention window are done according to a uniform probability distribution and not a geometrically distributed one. In the previous case, the a priori probability that the first transmission occurs t slots from now is geometrically distributed; it is $p(1-p)^{t-1}$, while with a contention window, it is equal to $1/\text{CW}$ for $t \in [1, \text{CW}]$ and 0 otherwise. This means that each node is *guaranteed* to attempt a transmission within CW slots, while that is not the case in the previous scheme, where there is always a chance, though exponentially decreasing, that a node may not transmit within any fixed number of slots.

2. The second difference is more minor: each node can avoid generating a random number in each slot; instead, it can generate a random number once per packet transmission attempt.

In the lab, you will implement the key parts of the contention window protocol and experiment with it in conjunction with CSMA. There is one important subtlety to keep in mind while doing this implementation. The issue has to do with how to count the slots before a node decides to transmit. Suppose a node decides that it will transmit x slots from now as long as the medium is idle after x slots; if x includes the busy slots when another node transmits, then multiple nodes may end up trying to transmit in the same time slot after the ending of a long packet transmission from another node, leading to excessive collisions. So it is important to only count down the idle slots; i.e., x should be the number of *idle* slots before the node attempts to transmit its packet (and of course, a node should try to send a packet in a slot only if it believes the medium to be idle in that slot).

■ 15.9 Summary

This lecture discussed the issues involved in sharing a communication medium amongst multiple nodes. We focused on contention protocols, developing ways to make them provide reasonable utilization and fairness. This is what we learned:

1. Good MAC protocols optimize utilization (throughput) and fairness, but must be able to solve the problem in a distributed way. In most cases, the overhead of a central controller node knowing which nodes have packets to send is too high. These protocols must also provide good utilization and fairness under dynamic load.

2. TDMA provides high throughput when all (or most of) the nodes are backlogged and the offered loads is evenly distributed amongst the nodes. When per-node loads are bursty or when different nodes send different amounts of data, TDMA is a poor choice.

3. Slotted Aloha has surprisingly high utilization for such a simple protocol, if one can pick the transmission probability correctly. The probability that maximizes throughput is $1/N$, where N is the number of backlogged nodes, the resulting utilization tends toward $1/e \approx 37\%$, and the fairness is close to 1 if all nodes present the same load. The utilization does remains high even when the nodes present different loads, in contrast to TDMA.

   It is also worth calculating (and noting) how many slots are left idle and how many slots have more than one node transmitting at the same time in slotted Aloha with $p = 1/N$. When N is large, these numbers are $1/e$ and $1 - 2/e \approx 26\%$, respectively. It is interesting that the number of idle slots is the same as the utilization: if we increase p to reduce the number of idle slots, we don't increase the utilization but actually increase the collision rate.

4. Stabilization is crucial to making Aloha practical. We studied a scheme that adjusts the transmission probability, reducing it multiplicatively when a collision occurs and increasing it (either multiplicatively or to a fixed maximum value) when a successful transmission occurs. The idea is to try to converge to the optimum value.

5. A non-zero lower bound on the transmission probability is important if we want to improve fairness, in particular to prevent some nodes from being starved. An upper bound smaller than 1 improves fairness over shorter time scales by alleviating the capture effect, a situation where one or a small number of nodes capture all the transmission attempts for many time slots in succession.

6. Slotted Aloha has double the utilization of unslotted Aloha when the number of backlogged nodes grows. The intuitive reason is that if two packets are destined to collide, the "window of vulnerability" is larger in the unslotted case by a factor of two.

7. A broadcast network that uses packets that are multiple slots in length (i.e., mimicking the unslotted case) can use carrier sense if the medium is a true broadcast medium (or approximately so). In a true broadcast medium, all nodes can hear each other reliably, so they can sense the carrier before transmitting their own packets. By "listening before transmitting" and setting the transmission probability using stabilization, they can reduce the number of collisions and increase utilization, but it is hard (if not impossible) to eliminate all collisions. Fairness still requires bounds on the transmission probability as before.

8. With a contention window, one can make the transmissions from backlogged nodes occur according to a uniform distribution, instead of the geometric distribution imposed by the "send with probability p" schemes. A uniform distribution in a finite window guarantees that each node will attempt a transmission within some fixed number of slots, which is not true of the geometric distribution.

■ Problems and Questions

1. We studied TDMA, (stabilized) Aloha, and CSMA protocols in this chapter. In each statement below, assume that the protocols are implemented correctly. Which of these statements is true (more than might be).

   (a) TDMA may have collisions when the size of a packet exceeds one time slot.

   (b) There exists some offered load for which TDMA has lower throughput than slotted Aloha.

   (c) In stabilized Aloha, two nodes have a certain probability of colliding in a time slot. If they actually collide in that slot, then they will experience a lower probability of colliding with each other when they each retry.

   (d) There is **no** workload for which stabilized Aloha achieves a utilization greater that $(1 - 1/N)^{N-1}$ ($\approx 1/e$ for large N) when run for a long period of time.

   (e) In slotted Aloha with stabilization, each node's transmission probability converges to $1/N$, where N is the number of backlogged nodes.

   (f) In a network in which all nodes can hear each other, CSMA will have no collisions when the packet size is larger than one time slot.

2. In the Aloha stabilization protocols we studied, when a node experiences a collision, it decreases its transmission probability, but sets a lower bound, p_{min}. When it transmits successfully, it increases its transmission probability, but sets an upper bound, p_{max}.

   (a) Why would we set a lower bound on p_{min} that is not too close to 0?

   (b) Why would we set p_{max} to be significantly smaller than 1?

   (c) Let N be the average number of backlogged nodes. What happens if we set $p_{\text{min}} >> 1/N$?

3. Alyssa and Ben are all on a shared medium wireless network running a variant of slotted Aloha (all packets are the same size and each packet fits in one slot). Their computers are configured such that Alyssa is 1.5 times as likely to send a packet as Ben. Assume that both computers are backlogged.

   (a) For Alyssa and Ben, what is their probability of transmission such that the utilization of their network is maximized?

   (b) What is the maximum utilization?

4. You have two computers, A and B, sharing a wireless network in your room. The network runs the slotted Aloha protocol with equal-sized packets. You want B to get twice the throughput over the wireless network as A whenever both nodes are backlogged. You configure A to send packets with probability p. What should you set the transmission probability of B to, in order to achieve your throughput goal?

5. Which of the following statements are *always* true for networks with $N > 1$ nodes using correctly implemented versions of unslotted Aloha, slotted Aloha, Time Division Multiple Access (TDMA) and Carrier Sense Multiple Access (CSMA)? Unless otherwise stated, assume that the slotted and unslotted versions of Aloha are stabilized and use the same stabilization method and parameters. Explain your answer for each statement.

   (a) There exists some offered load pattern for which TDMA has lower throughput than slotted Aloha.

   (b) Suppose nodes I, II and III use a fixed probability of $p = 1/3$ when transmitting on a 3-node slotted Aloha network (i.e., $N = 3$). If all the nodes are backlogged then over time the utilization averages out to $1/e$.

   (c) When the number of nodes, N, is large in a stabilized slotted Aloha network, setting $p_{\max} = p_{\min} = 1/N$ will achieve the same utilization as a TDMA network if all the nodes are backlogged.

   (d) Using contention windows with a CSMA implementation guarantees that a packet will be transmitted successfully within some bounded time.

6. Suppose that there are three nodes, $A, B,$ and C, seeking access to a shared medium using slotted Aloha, each using some fixed probability of transmission, where each packet takes one slot to transmit. Assume that the nodes are always backlogged, and that node A has half the probability of transmission as the other two, i.e., $p_A = p$ and $p_B = p_C = 2p$.

   (a) If $p_A = 0.3$, compute the average utilization of the network.

   (b) What value of p_A maximizes the average utilization of the network and what is the corresponding maximum utilization?

7. Ben Bitdiddle sets up a shared medium wireless network with one access point and N client nodes. Assume that the N client nodes are backlogged, each with packets destined for the access point. The access point is also backlogged, with each of its packets destined for some client. The network uses slotted Aloha with each packet fitting exactly in one slot. Recall that each backlogged node in Aloha sends a packet with some probability p. Two or more distinct nodes (whether client or access point) sending in the same slot causes a collision. Ben sets the transmission probability, p, of each client node to $1/N$ and sets the transmission probability of the access point to a value p_a.

   (a) What is the utilization of the network in terms of N and p_a?

   (b) Suppose N is large. What value of p_a ensures that the aggregate throughput of packets received successfully by the N clients is the same as the throughput of the packets received successfully by the access point?

8. Consider the same setup as the previous problem, but *only the client nodes are backlogged—the access point has no packets to send.* Each client node sends with probability p (don't assume it is $1/N$).

Ben Bitdiddle comes up with a cool improvement to the receiver at the access point. If exactly one node transmits, then the receiver works as usual and is able to correctly decode the packet. If exactly two nodes transmit, he uses a method to *cancel the interference* caused by each packet on the other, and is (quite remarkably) able to decode both packets correctly.

(a) What is the probability, P_2, of *exactly* two of the N nodes transmitting in a slot? Note that we want the probability of *any two* nodes sending in a given slot.

(b) What is the utilization of slotted Aloha with Ben's receiver modification? Write your answer in terms of N, p, and P_2, where P_2 is defined in the problem above.

9. Imagine a shared medium wireless network with N nodes. Unlike a perfect broadcast network in which all nodes can reliably hear any other node's transmission attempt, nodes in our network hear each other probabilistically. That is, between any two nodes i and j, i can hear j's transmission attempt with some probability p_{ij}, where $0 \leq p_{ij} \leq 1$. Assume that all packets are of the same size and that the time slot used in the MAC protocol is much smaller than the packet size.

   (a) Show a configuration of nodes where the throughput achieved when the nodes all use carrier sense is higher than if they didn't.

   (b) Show a configuration of nodes where the throughput achieved when slotted Aloha without carrier sense is higher than with carrier sense.

10. Token-passing is a variant of a TDMA MAC protocol. Here, the N nodes sharing the medium are numbered $0, 1, \ldots N-1$. The token starts at node 0. A node can send a packet if, and only if, it has the token. When node i with the token has a packet to send, it sends the packet and then passes the token to node $(i+1)$ mod N. If node i with the token does not have a packet to send, it passes the token to node $(i+1)$ mod N. To pass the token, a node broadcasts a token packet on the medium and all other nodes hear it correctly.

    A data packet occupies the medium for time T_d. A token packet occupies the medium for time T_k. If s of the N nodes in the network have data to send when they get the token, what is the utilization of the medium? Note that the bandwidth used to send tokens is pure overhead; the throughput we want corresponds to the rate at which data packets are sent.

11. Alyssa P. Hacker is designing a MAC protocol for a network used by people who: live on a large island, never sleep, never have guests, and are always on-line. Suppose the island's network has N nodes, and the island dwellers always keep **exactly some four of these nodes backlogged**. The nodes communicate with each other by beaming their data to a satellite in the sky, which in turn broadcasts the data down. If two or more nodes transmit in the same slot, their transmissions collide (the satellite uplink doesn't interfere with the downlink). The nodes on the ground **cannot hear each other**, and each node's packet transmission probability is non-zero. Alyssa uses a slotted protocol with **all packets equal to one slot in length**.

(a) For the slotted Aloha protocol with a **fixed** per-node transmission probability, what is the maximum utilization of this network? (Note that there are N nodes in all, of which some four are constantly backlogged.)

(b) Suppose the protocol is the slotted Aloha protocol, and the each island dweller greedily doubles his node transmission probability on each packet collision (but not exceeding 1). What do you expect the network utilization to be?

(c) In this network, as mentioned above, four of the N nodes are constantly backlogged, but the set of backlogged nodes is not constant. Suppose Alyssa must decide between slotted Aloha with a transmission probability of $1/5$ or time division multiple access (TDMA) among the N nodes. For what N does the expected utilization of this slotted Aloha protocol exceed that of TDMA?

(d) Alyssa implements a stabilization protocol to adapt the node transmission probabilities on collisions and on successful transmissions. She runs an experiment and finds that the measured utilization is 0.5. Ben Bitdiddle asserts that this utilization is too high and that she must have erred in her *measurements*. Explain whether or not it is possible for Alyssa's implementation of stabilization to be consistent with her measured result.

12. Tim D. Vider thinks Time Division Multiple Access (TDMA) is the best thing since sliced bread ("if equal slices are good for bread, then equal slices of time must be good for the MAC too", he says). Each packet is one time slot long.

    However, in Tim's network with N nodes, the **offered load is not uniform** across the different nodes. The **rate** at which node i generates new packets to transmit is $r_i = \frac{1}{2^i}$ packets per time slot ($1 \leq i \leq N$). That is, in each time slot, the **application** on node i produces a packet to send over the network with probability r_i.

    (a) Tim runs an experiment with TDMA for a large number of time slots. At the end of the experiment, how many nodes (as a function of N) will have a substantial backlog of packets (i.e., queues that are growing with time)?

    (b) Let $N = 20$. Calculate the **utilization** of this non-uniform workload running over TDMA.

13. Recall the MAC protocol with contention windows from §15.8. Here, each node maintains a contention window, W, and sends a packet t idle time slots after the current slot, where t is an integer picked uniformly in $[1, W]$. Assume that each packet is 1 slot long.

    Suppose there are two backlogged nodes in the network with contention windows W_1 and W_2, respectively ($W_1 \geq W_2$). Suppose that both nodes pick their random value of t at the same time. What is the probability that the two nodes will collide the next time they each transmit?

14. Eager B. Eaver gets a new computer with *two* radios. There are N other devices on the shared medium network to which he connects, but each of the other devices has only one radio. The MAC protocol is slotted Aloha with a packet size equal to 1 time

slot. Each device uses a fixed transmission probability, and only one packet can be sent successfully in any time slot. All devices are backlogged.

Eager persuades you that because he has paid for two radios, his computer has a moral right to get twice the throughput of any other device in the network. You begrudgingly agree.

Eager develops two protocols:

Protocol A: Each radio on Eager's computer runs its MAC protocol independently. That is, each radio sends a packet with fixed probability p. Each other device on the network sends a packet with probability p as well.

Protocol B: Eager's computer runs a single MAC protocol across its two radios, sending packets with probability $2p$, and alternating transmissions between the two radios. Each other device on the network sends a packet with probability p.

(a) With which protocol, A or B, will Eager achieve higher throughput?

(b) Which of the two protocols would you allow Eager to use on the network so that his expected throughput is double any other device's?

15. Carl Coder implements a simple slotted Aloha-style MAC for his room's wireless network. His room has only two backlogged nodes, A and B. Carl picks a transmission probability of $2p$ for node A and p for node B. Each packet is one time slot long and all transmissions occur at the beginning of a time slot.

(a) What is the utilization of Carl's network in terms of p?

(b) What value of p maximizes the utilization of this network, **and** what is the maximum utilization?

(c) Instead of maximizing the utilization, suppose Carl chooses p so that the throughput achieved by A is **three times** the throughput achieved by B. What is the utilization of his network now?

16. Carl Coder replaces the "send with fixed probability" MAC of the previous problem with one that uses a contention window at each node. He configures node A to use a fixed contention window of W and node B to use a fixed contention window of $2W$. Before a transmission, each node independently picks a random integer t uniformly between 1 and its contention window value, and transmits a packet t time slots from now. Each packet is one time slot long and all transmissions occur at the beginning of a time slot.

(a) Which node, A or B, has a higher probability of being the next to transmit a packet successfully? (Use intuition, don't calculate!)

(b) What is the probability that A and B will collide the next time they each transmit?

(c) Suppose A and B each pick a contention window value at some point in time. What is the probability that A transmits before B successfully on its next transmission *attempt*? Note that this probability is equal to the probability that the

value picked by A value is strictly smaller than the value picked by B. (It may be useful to apply the formula $\sum_{i=1}^{n} i = n(n+1)/2$.)

(d) Suppose there is no collision at the next packet transmission. Calculate the probability that A will transmit before B? Explain why this answer is different from the answer to the previous part. You should be able to obtain the solve this problem using the previous two parts.

(e) None of the previous parts directly answer the question, "What is the probability that A will be the first node to successfully transmit a packet before B?" Explain why.

17. Ben Bitdiddle runs the slotted Aloha protocol with stabilization. Each packet is one time slot long. At the beginning of time slot T, node i has a probability of transmission equal to p_i, $1 \leq i \leq N$, where N is the number of backlogged nodes. The increase/decrease rules for p_i are doubling/halving, with $p_{\min} \leq p_i \leq p_{\max}$, as described in this chapter.

Ben notes that exactly two nodes, j and k, transmit in time slot T. After thinking about what happens to these two packets, derive an expression for the probability that **exactly one node** (out of the N backlogged nodes) will transmit successfully in time slot $T+1$.

MIT 6.02 Lecture Notes
Last update: November 3, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 16
Communication Networks: Sharing and Switches

Thus far we have studied techniques to engineer a *point-to-point* communication link to send messages between two directly connected devices. These techniques give us a communication link between two devices that, in general, has a certain error rate and a corresponding message loss rate. Message losses occur when the error correction mechanism is unable to correct all the errors that occur due to noise or interference from other concurrent transmissions in a contention MAC protocol.

We now turn to the study of *multi-hop communication networks*—systems that connect three or more devices together.[1] The key idea that we will use to engineer communication networks is *composition*: we will build small networks by composing links together, and build larger networks by composing smaller networks together.

The fundamental challenges in the design of a communication network are the same as those that face the designer of a communication link: **sharing for efficiency** and **reliability**. The big difference is that the sharing problem has different challenges because the system is now *distributed*, spread across a geographic span that is much larger than even the biggest shared medium we can practically build. Moreover, as we will see, many more things can go wrong in a network in addition to just bit errors on the point-to-point links, making communication more unreliable than a single link's unreliability.[2]. The next few chapters will discuss these two challenges and the key principles to overcome them.

In addition to sharing and reliability, an important and difficult problem that many communication networks (such as the Internet) face is *scalability*: how to engineer a very large, global system. We won't say very much about scalability in this book, leaving this important topic for more advanced courses.

This chapter focuses on the sharing problem and discusses the following concepts:

1. Switches and how they enable *multiplexing* of different communications on individual links and over the network. Two forms of switching: circuit switching and packet

[1] By device, we mean things like computer, phones, embedded sensors, and the like—pretty much anything with some computation and communication capability that can be part of a network.

[2] As one wag put it: "Networking, just one letter away from not working."

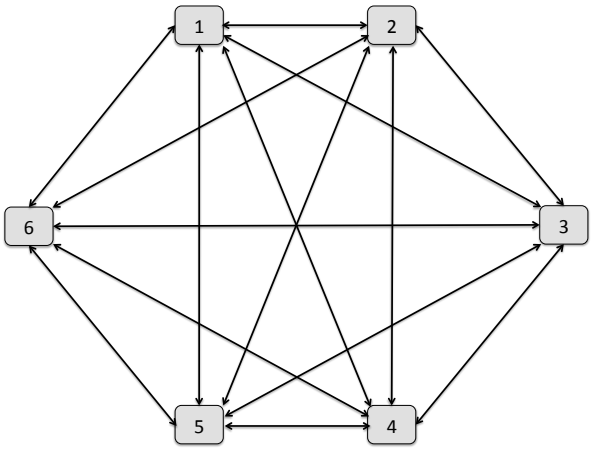


Figure 16-1: A communication network with a link between every pair of devices has a quadratic number of links. Such topologies are generally too expensive, and are especially untenable when the devices are far from each other.

switching.

2. Understanding the role of queues to absorb bursts of traffic in packet-switched networks.

3. Understanding the factors that contribute to delays in networks: three largely fixed delays (propagation, processing, and transmission delays), and one significant variable source of delays (queueing delays).

4. Little's law, relating the average delay to the average rate of arrivals and the average queue size.

■ 16.1 Sharing with Switches

The collection of techniques used to design a communication link, including modulation and error-correcting channel coding, is usually implemented in a module called the *physical layer* (or "PHY" for short). The sending PHY takes a stream of bits and arranges to send it across the link to the receiver; the receiving PHY provides its best estimate of the stream of bits sent from the other end. On the face of it, once we know how to develop a communication link, connecting a collection of N devices together is ostensibly quite straightforward: one could simply connect each pair of devices with a wire and use the physical layer running over the wire to communicate between the two devices. This picture for a small 5-node network is shown in Figure 16-1.

This simple strawman using dedicated pairwise links has two severe problems. First, it is extremely expensive. The reason is that the number of distinct communication links that one needs to build scales quadratically with N—there are $\binom{N}{2} = \frac{N(N-1)}{2}$ bi-directional links in this design (a *bi-directional* link is one that can transmit data in both directions,

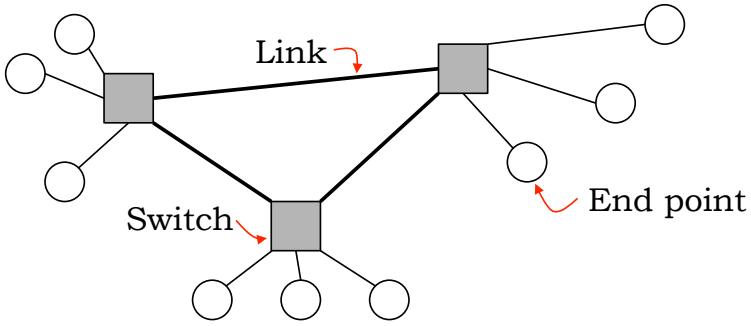


Figure 16-2: A simple network topology showing communicating end points, links, and switches.

as opposed to a *uni-directional* link). The cost of operating such a network would be prohibitively expensive, and each additional node added to the network would incur a cost proportional to the size of the network! Second, some of these links would have to span an enormous distance; imagine how the devices in Cambridge, MA, would be connected to those in Cambridge, UK, or (to go further) to those in India or China. Such "long-haul" links are difficult to engineer, so one can't assume that they will be available in abundance.

Clearly we need a better design, one that can "do for a dime what any fool can do for a dollar".[3] The key to a practical design of a communication network is a special computing device called a *switch*. A switch has multiple "interfaces" (often also called "ports") on it; a link (wire or radio) can be connected to each interface. The switch allows multiple different communications between different pairs of devices to run over each individual link—that is, it arranges for the network's links to be *shared* by different communications. In addition to the links, the switches themselves have some resources (memory and computation) that will be shared by all the communicating devices.

Figure 16-2 shows the general idea. A switch receives bits that are encapsulated in *data frames* arriving over its links, processes them (in a way that we will make precise later), and forwards them (again, in a way that we will make precise later) over one or more other links. In the most common kind of network, these frames are called *packets*, as explained below.

We will use the term *end points* to refer to the communicating devices, and call the switches and links over which they communicate the *network infrastructure*. The resulting structure is termed the *network topology*, and consists of *nodes* (the switches and end points) and links. A simple network topology is shown in Figure 16-2. We will model the network topology as a *graph*, consisting of a set of nodes and a set of links (edges) connecting various nodes together, to solve various problems.

Figure 16-3 show a few switches of relatively current vintage (ca. 2006).

■ 16.1.1 Three Problems That Switches Solve

The fundamental functions performed by switches are to multiplex and demultiplex data frames belonging to different device-to-device information transfer sessions, and to determine the link(s) along which to forward any given data frame. This task is essential be-

[3]That's what an engineer does, according to an old saying.

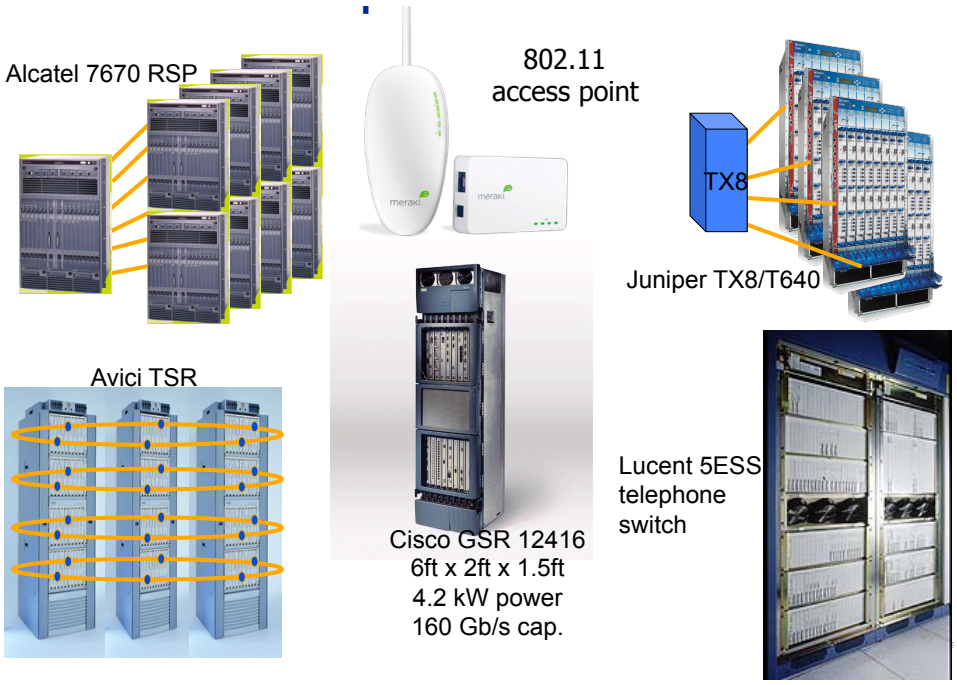


Figure 16-3: A few modern switches.

cause a given physical link will usually be shared by several concurrent sessions between different devices. We break these functions into three problems:

1. **Forwarding:** When a data frame arrives at a switch, the switch needs to process it, determine the correct outgoing link, and decide when to send the frame on that link.

2. **Routing:** Each switch somehow needs to determine the topology of the network, so that it can correctly construct the data structures required for proper forwarding. The process by which the switches in a network collaboratively compute the network topology, adapting to various kinds of failures, is called routing. It does not happen on each data frame, but occurs in the "background". The next two chapters will discuss forwarding and routing in more detail.

3. **Resource allocation:** Switches allocate their resources—access to the link and local memory—to the different communications that are in progress.

Over time, two radically different methods have been developed for solving these problems. These techniques differ in the way the switches forward data and allocate resources (there are also some differences in routing, but they are less significant). The first method, used by networks like the telephone network, is called *circuit switching*. The second method, used by networks like the Internet, is called *packet switching*.

There are two crucial differences between the two methods, one philosophical and the other mechanistic. The mechanistic difference is the easier one to understand, so we'll talk about it first. In a circuit-switched network, the frames do not (need to) carry any special information that tells the switches how to forward information, while in packet-switched

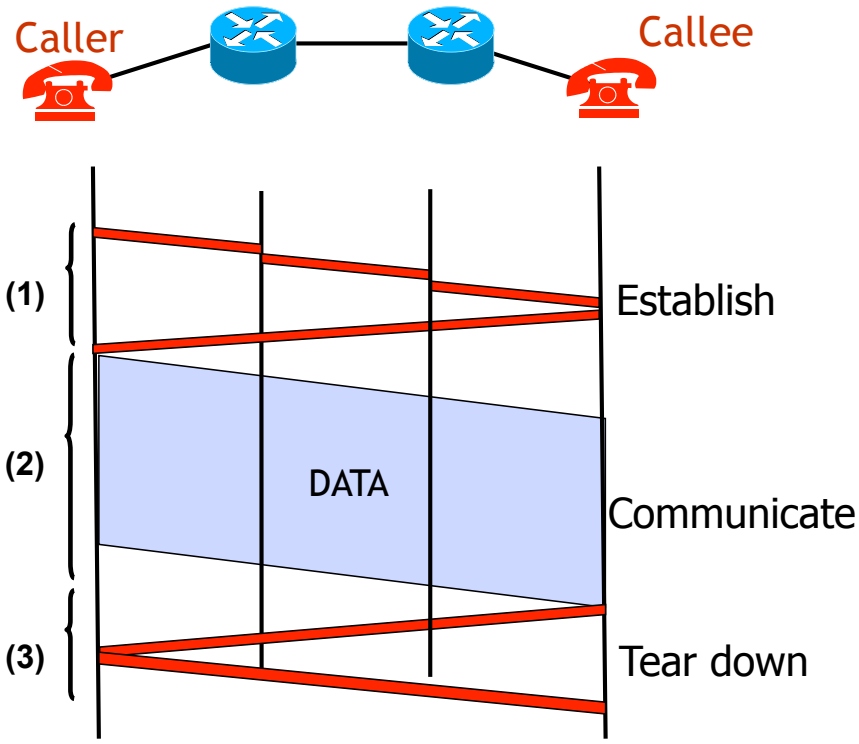


Figure 16-4: Circuit switching requires setup and teardown phases.

networks, they do. The philosophical difference is more substantive: a circuit-switched network provides the abstraction of a *dedicated link* of some bit rate to the communicating entities, whereas a packet switched network does not.[4] Of course, this dedicated link traverses multiple physical links and at least one switch, so the end points and switches must do some additional work to provide the illusion of a dedicated link. A packet-switched network, in contrast, provides no such illusion; once again, the end points and switches must do some work to provide reliable and efficient communication service to the applications running on the end points.

■ 16.2 Circuit Switching

The transmission of information in circuit-switched networks usually occurs in three phases (see Figure 16-4):

1. The *setup phase*, in which some state is configured at each switch along a path from source to destination,
2. The *data transfer phase* when the communication of interest occurs, and
3. The *teardown phase* that cleans up the state in the switches after the data transfer ends.

[4]One can try to layer such an abstraction atop a packet-switched network, but we're talking about the inherent abstraction provided by the network here.

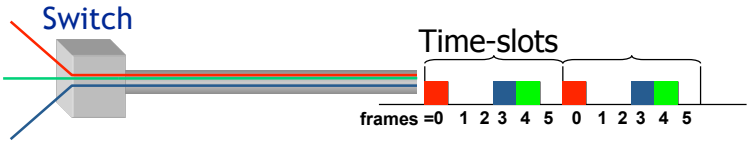


Figure 16-5: Circuit switching with Time Division Multiplexing (TDM). Each color is a different conversation and there are a maximum of $N = 6$ concurrent communications on the link in this picture. Each communication (color) is sent in a fixed time-slot, modulo N.

Because the frames themselves contain no information about where they should go, the setup phase needs to take care of this task, and also configure (reserve) any resources needed for the communication so that the illusion of a dedicated link is provided. The teardown phase is needed to release any reserved resources.

■ 16.2.1 Example: Time-Division Multiplexing (TDM)

A common (but not the only) way to implement circuit switching is using *time-division multiplexing (TDM)*, also known as *isochronous transmission*. Here, the physical capacity, or *bit rate*,[5] of a link connected to a switch, C (in bits/s), is conceptually divided into N "virtual links", each virtual link being allocated C/N bits/s and associated with a data transfer session. Call this quantity R, the *rate* of each independent data transfer session. Now, if we constrain each frame to be of some fixed size, s bits, then the switch can perform time multiplexing by allocating the link's capacity in time-slots of length s/C units each, and by associating the i^{th} time-slice to the i^{th} transfer (modulo N), as shown in Figure 16-5. It is easy to see that this approach provides each session with the required rate of R bits/s, because each session gets to send s bits over a time period of Ns/C seconds, and the ratio of the two is equal to $C/N = R$ bits/s.

Each data frame is therefore forwarded by simply using the time slot in which it arrives at the switch to decide which port it should be sent on. Thus, the state set up during the first phase has to associate one of these channels with the corresponding soon-to-follow data transfer by allocating the i^{th} time-slice to the i^{th} transfer. The end points transmitting data send frames only at the specific time-slots that they have been told to do so by the setup phase.

Other ways of doing circuit switching include *wavelength division multiplexing (WDM)*, *frequency division multiplexing (FDM)*, and *code division multiplexing (CDM)*; the latter two (as well as TDM) are used in some wireless networks, while WDM is used in some high-

[5]This number is sometimes referred to as the "bandwidth" of the link. Technically, bandwidth is a quantity measured in Hertz and refers to the width of the frequency over which the transmission is being done. To avoid confusion, we will use the term "bit rate" to refer to the number of bits per second that a link is currently operating at, but the reader should realize that the literature often uses "bandwidth" to refer to this term. The reader should also be warned that some people (curmudgeons?) become apoplectic when they hear someone using "bandwidth" for the bit rate of a link. A more reasonable position is to realize that when the context is clear, there's not much harm in using "bandwidth". The reader should also realize that in practice most wired links usually operate at a single bit rate (or perhaps pick one from a fixed set when the link is configured), but that wireless links using radio communication can operate at a range of bit rates, adaptively selecting the modulation and coding being used to cope with the time-varying channel conditions caused by interference and movement.

speed wired optical networks.

16.2.2 Pros and Cons

Circuit switching makes sense for a network where the workload is relatively uniform, with all information transfers using the same capacity, and where each transfer uses a *constant bit rate* (or near-constant bit rate). The most compelling example of such a workload is telephony, where each digitized voice call might operate at 64 kbits/s. Switching was first invented for the telephone network, well before devices were on the scene, so this design choice makes a great deal of sense. The classical telephone network as well as the cellular telephone network in most countries still operate in this way, though telephony over the Internet is becoming increasingly popular and some of the network infrastructure of the classical telephone networks is moving toward packet switching.

However, circuit-switching tends to waste link capacity if the workload has a *variable bit rate*, or if the frames arrive in bursts at a switch. Because a large number of computer applications induce burst data patterns, we should consider a different link sharing strategy for computer networks. Another drawback of circuit switching shows up when the $(N+1)^{\text{st}}$ communication arrives at a switch whose relevant link already has the maximum number (N) of communications going over it. This communication must be denied access (or admission) to the system, because there is no capacity left for it. For applications that require a certain minimum bit rate, this approach might make sense, but even in that case a "busy tone" is the result. However, there are many applications that don't have a minimum bit rate requirement (file delivery is a prominent example); for this reason as well, a different sharing strategy is worth considering.

Packet switching doesn't have these drawbacks.

16.3 Packet Switching

An attractive way to overcome the inefficiencies of circuit switching is to permit any sender to transmit data at any time, but yet allow the link to be shared. Packet switching is a way to accomplish this task, and uses a tantalizingly simple idea: add to each frame of data a little bit of information that tells the switch how to forward the frame. This information is usually added inside a *header* immediately before the payload of the frame, and the resulting frame is called a *packet*.[6] In the most common form of packet switching, the header of each packet contains the *address* of the destination, which uniquely identifies the destination of data. The switches use this information to process and forward each packet. Packets usually also include the sender's address to help the receiver send messages back to the sender. A simple example of a packet header is shown in Figure 16-6. In addition to the destination and source addresses, this header shows a checksum that can be used for error detection at the receiver.

The figure also shows the packet header used by IPv6 (the Internet Protocol version 6), which is increasingly used on the Internet today. The Internet is the most prominent and successful example of a packet-switched network.

The job of the switch is to use the destination address as a key and perform a lookup on

[6]Sometimes, the term *datagram* is used instead of (or in addition to) the term "packet".

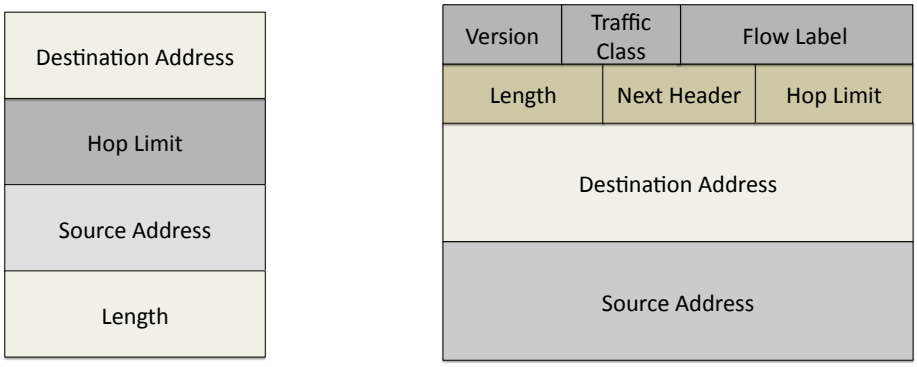


Figure 16-6: LEFT: A *simple and basic* example of a packet header for a packet-switched network. The destination address is used by switches in the forwarding process. The hop limit field will be explained in the chapter on network routing; it is used to discard packets that have been forwarded in the network for more than a certain number of hops, because it's likely that those packets are simply stuck in a loop. Following the header is the payload (or data) associated with the packet, which we haven't shown in this picture. RIGHT: For comparison, the format of the IPv6 ("IP version 6") packet header is shown. Four of the eight fields are similar to our simple header format. The additional fields are the version number, which specifies the version of IP, such as "6" or "4" (the current version that version 6 seeks to replace) and fields that specify, or hint at, how switches must prioritize or provide other traffic management features for the packet.

a data structure called a *routing table*. This lookup returns an outgoing link to forward the packet on its way toward the intended destination. There are many ways to implement the lookup opertion on a routing table, but for our purposes we can consider the routing table to be a dictionary mapping each destination to one of the links on the switch.

While forwarding is a relatively simple[7] lookup in a data structure, the trickier question that we will spend time on is determining how the entries in the routing table are obtained. The plan is to use a background process called a *routing protocol*, which is typically implemented in a distributed manner by the switches. There are two common classes of routing protocols, which we will study in later chapters. For now, it is enough to understand that if the routing protocol works as expected, each switch obtains a *route* to every destination. Each switch participates in the routing protocol, dynamically constructing and updating its routing table in response to information received from its neighbors, and providing information to each neighbor to help them construct their own routing tables.

Switches in packet-switched networks that implement the functions described in this section are also known as *routers*, and we will use the terms "switch" and "router" interchangeably when talking about packet-switched networks.

■ 16.3.1 Why Packet Switching Works: Statistical Multiplexing

Packet switching does not provide the illusion of a dedicated link to any pair of communicating end points, but it has a few things going for it:

[7] At low speeds. At high speeds, forwarding is a challenging problem.

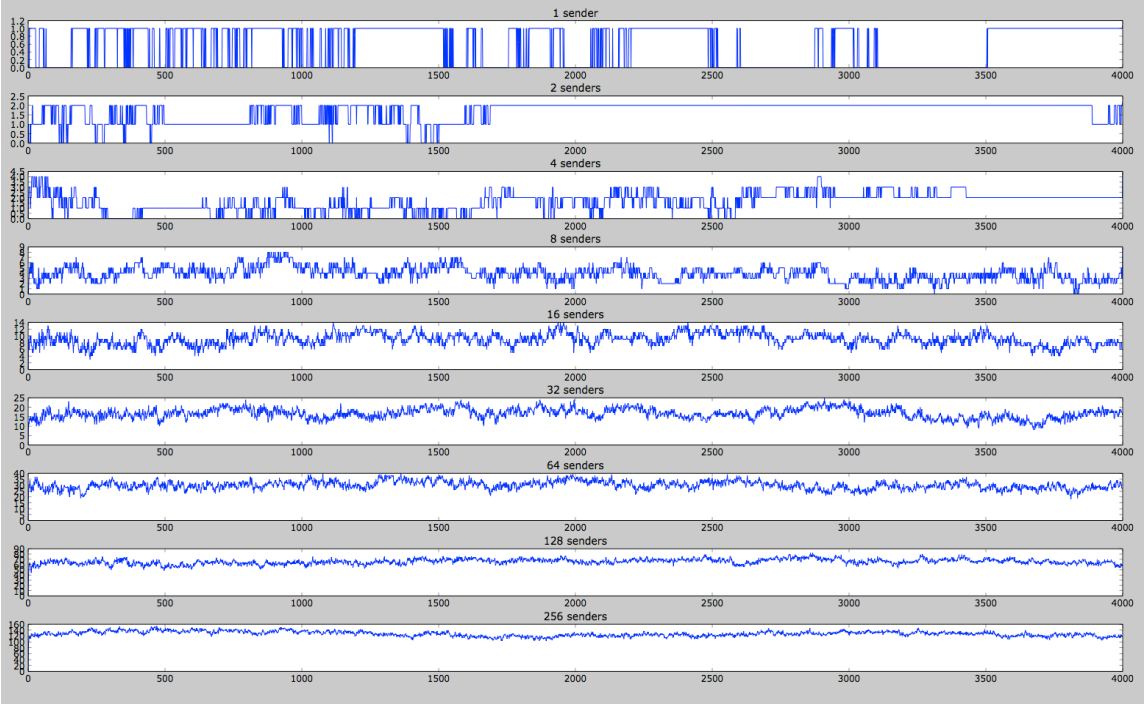


Figure 16-7: Packet switching works because of statistical multiplexing. This picture shows a simulation of N senders, each connected at a fixed bit rate of 1 megabit/s to a switch, sharing a single outgoing link. The y-axis shows the aggregate bit rate (in megabits/s) as a function of time (in milliseconds). In this simulation, each sender is in either the "on" (sending) state or the "off" (idle) state; the durations of each state are drawn from a Pareto distribution (which has a "heavy tail").

1. It doesn't waste the capacity of any link because each switch can send any packet available to it that needs to use that link.

2. It does not require any setup or teardown phases and so can be used even for small transfers without any overhead.

3. It can provide variable data rates to different communications essentially on an "as needed" basis.

At the same time, because there is no reservation of resources, packets could arrive faster than can be sent over a link, and the switch must be able to handle such situations. Switches deal with transient bursts of traffic that arrive faster than a link's bit rate using *queues*. We will spend some time understanding what a queue does and how it absorbs bursts, but for now, let's assume that a switch has large queues and understand why packet switching actually works.

Packet switching supports end points sending data at variable rates. If a large number of end points conspired to send data in a synchronized way to exercise a link at the same time, then one would end up having to provision a link to handle the peak synchronized rate to provide reasonable service to all the concurrent communications.

Fortunately, at least in a network with benign, or even greedy (but non-malicious) sending nodes, it is highly unlikely that all the senders will be perfectly synchronized. Even

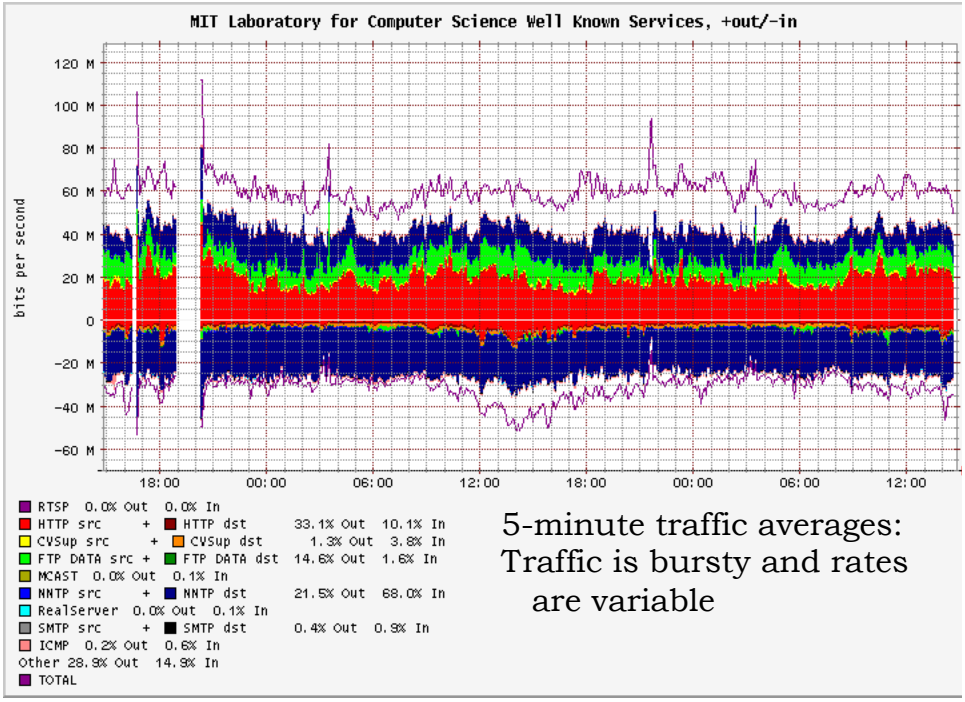


Figure 16-8: Network traffic variability.

when senders send long bursts of traffic, as long as they alternate between "on" and "off" states and move between these states at random (the probability distributions for these could be complicated and involve "heavy tails" and high variances), the aggregate traffic of multiple senders tends to smooth out a bit.[8]

An example is shown in Figure 16-7. The x-axis is time in milliseconds and the y-axis shows the bit rate of the set of senders. Each sender has a link with a fixed bit rate connecting it to the switch. The picture shows how the aggregate bit rate over this short time-scale (4 seconds), though variable, becomes smoother as more senders share the link. This kind of multiplexing relies on the randomness inherent in the concurrent communications, and is called *statistical multiplexing*.

Real-world traffic has bigger bursts than shown in this picture and the data rate usually varies by a large amount depending on time of day. Figure 16-8 shows the bit rates observed at an MIT lab for different network applications. Each point on the y-axis is a 5-minute average, so it doesn't show the variations over smaller time-scales as in the previous figure. However, it shows how much variation there is with time-of-day.

So far, we have discussed how the aggregation of multiple sources sending data tends to smooth out traffic a bit, enabling the network designer to avoid provisioning a link for the sum of the peak offered loads of the sources. In addition, for the packet switching idea to really work, one needs to appreciate the time-scales over which bursts of traffic occur in real life.

[8]It's worth noting that many large-scale *distributed denial-of-service attacks* try to take out web sites by saturating its link with a huge number of synchronized requests or garbage packets, each of which individually takes up only a tiny fraction of the link.

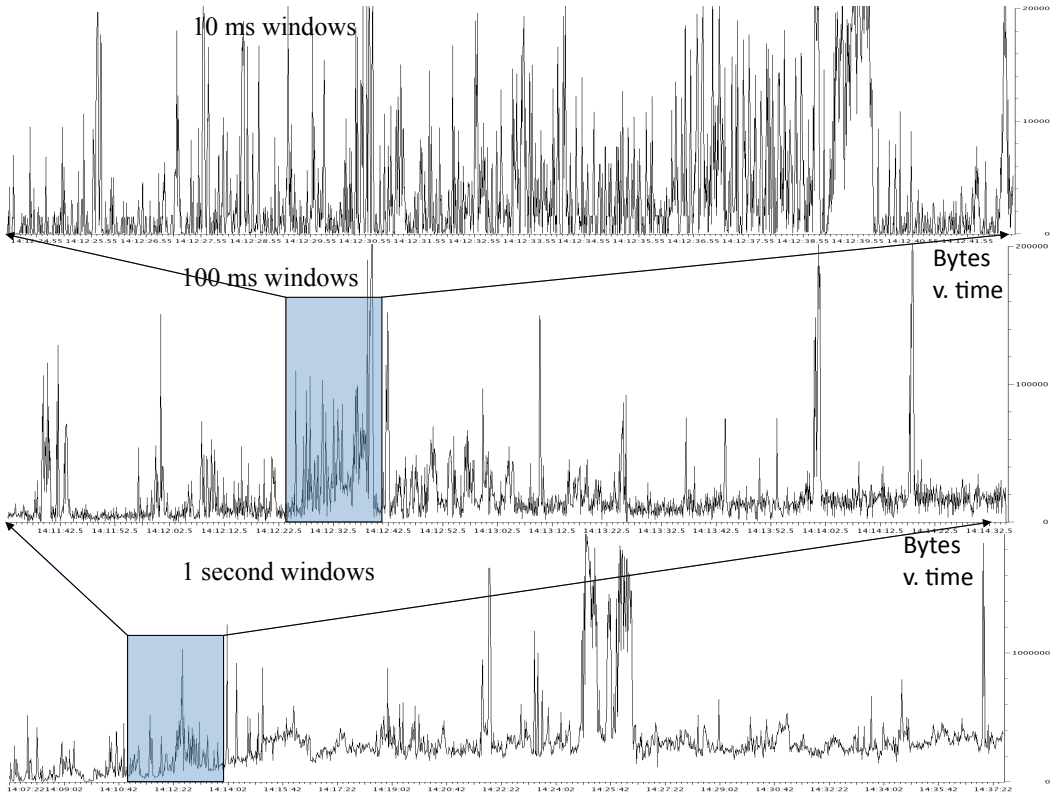


Figure 16-9: Traffic bursts at different time-scales, showing some smoothing. Bursts still persist, though.

What better example to use than traffic generated over the duration of a 6.02 lecture on the 802.11 wireless LAN in 34-101 to illustrate the point?! We captured all the traffic that traversed this shared wireless network on a few days during lecture in Fall 2010. On a typical day, we measured about 1 Gigabyte of traffic traversing the wireless network via the access point our monitoring laptop was connected to, with numerous applications in the mix. Most of the observed traffic was from Bittorrent, Web browsing, email, with the occasional IM sessions thrown in the mix. Domain name system (DNS) lookups, which are used by most Internet applications, also generate a sizable number of packets (but not bytes).

Figure 16-9 shows the aggregate amount of data, in bytes, as a function of time, over different time durations. The top picture shows the data over 10 millisecond windows—here, each y-axis point is the total number of bytes observed over the wireless network corresponding to a non-overlapping 10-millisecond time window. We show the data here for a randomly chosen time period that lasts 17 seconds. The most noteworthy aspect of this picture is the bursts that are evident: the maximum (not shown) is as high as 50,000 bytes over this duration, but also note how successive time windows could change between close to 20,000 bytes and 0 bytes. From time to time, larger bursts occur where the network is essentially continuously in use (for example, starting at 14:12:38.55).

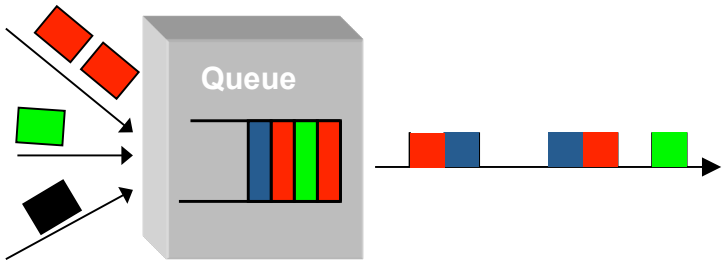


Figure 16-10: Packet switching uses queues to buffer bursts of packets that have arrived at a rate faster than the bit rate of the link.

The middle picture shows what happens when we look at windows that are 100 milliseconds long. Clearly, bursts persist, but one can see from the picture that the variance has reduced. When we move to longer windows of 1 second each, we see the same effect persisting, though again it's worth noting that the bursts don't actually disappear.

These data sets exemplify the traffic dynamics that a network designer has to plan for while designing a network. One could pick a data rate that is higher than the peak expected over a short time-scale, but that would be several times larger than picking a smaller value and using a queue to absorb the bursts and send out packets over a link of a smaller rate. In practice, this problem is complicated because network sources are not "open loop", but actually react to how the network responds to previously sent traffic. Understanding how this feedback system works is beyond the scope of 6.02; here, we will look at how queues work.

■ 16.3.2 Absorbing bursts with queues

Queues are a crucial component in any packet-switched network. The queues in a switch absorb bursts of data (see Figure 16-10): when packets arrives for an outgoing link faster than the speed of that link, the queue for that link stores the arriving packets. If a packet arrives and the queue is full, then that packet is simply dropped (if the packet is really important, then the original sender can always infer that the packet was lost because it never got an acknowledgment for it from the receiver, and might decide to re-send it).

One might be tempted to provision large amounts of memory for packet queues because packet losses sound like a bad thing. In fact, queues are like seasoning in a meal—they need to be "just right" in quantity (size). Too small, and too many packets may be lost, but too large, and packets may be excessively delayed, causing it to take *longer* for the senders to know that packets are only getting stuck in a queue and not being delivered.

So how big must queues be? The answer is not that easy: one way to think of it is to ask what we might want the maximum packet delay to be, and use that to size the queue. A more nuanced answer is to analyze the dynamics of how senders react to packet losses and use that to size the queue. Answering this question is beyond the scope of this course, but is an important issue in network design. (The short answer is that we typically want a few tens to ≈ 100 milliseconds of a queue size—that is, we want the queueing delay of a packet to not exceed this quantity, so the buffer size in bytes should be this quantity multiplied

by the rate of the link concerned.)

Thus, queues can prevent packet losses, but they cause packets to get delayed. These delays are therefore a "necessary evil". Moreover, queueing delays are *variable*—different packets experience different delays, in general. As a result, analyzing the performance of a network is not a straightforward task. We will discuss performance measures next.

■ 16.4 Network Performance Metrics

Suppose you are asked to evaluate whether a network is working well or not. To do your job, it's clear you need to define some metrics that you can measure. As a user, if you're trying to deliver or download some data, a natural measure to use is the time it takes to finish delivering the data. If the data has a size of S bytes, and it takes T seconds to deliver the data, the *throughput* of the data transfer is $\frac{S}{T}$ bytes/second. The greater the throughput, the happier you will be with the network.

The throughput of a data transfer is clearly upper-bounded by the rate of the slowest link on the path between sender and receiver (assuming the network uses only one path to deliver data). When we discuss reliable data delivery, we will develop protocols that attempt to optimize the throughput of a large data transfer. Our ability to optimize throughput depends more fundamentally on two factors: the first factor is the *per-packet delay*, sometimes called the per-packet *latency* and the second factor is the *packet loss rate*.

The packet loss rate is easier to understand: it is simply equal to the number of packets dropped by the network along the path from sender to receiver divided by the total number of packets transmitted by the sender. So, if the sender sent S_t packets and the receiver got S_r packets, then the packet loss rate is equal to $1 - \frac{S_r}{S_t} = \frac{S_t - S_r}{S_t}$. One can equivalently think of this quantity in terms of the sending and receiving rates too: for simplicity, suppose there is one queue that drops packets between a sender and receiver. If the arrival rate of packets into the queue from the sender is A packets per second and the departure rate from the queue is D packets per second, then the packet loss rate is equal to $1 - \frac{D}{A}$.

The delay experienced by packets is actually the sum of four distinct sources: *propagation*, *transmission*, *processing*, and *queueing*, as explained below:

1. **Propagation delay.** This source of delay is due to the fundamental limit on the time it takes to send any signal over the medium. For a wire, it's the speed of light over that material (for typical fiber links, it's about two-thirds the speed of light in vacuum). For radio communication, it's the speed of light in vacuum (air), about 3×10^8 meters/second.

   The best way to think about the propagation delay for a link is that it is equal to the *time for the first bit of any transmission to reach the intended destination*. For a path comprising multiple links, just add up the individual propagation delays to get the propagation delay of the path.

2. **Processing delay.** Whenever a packet (or data frame) enters a switch, it needs to be processed before it is sent over the outgoing link. In a packet-switched network, this processing involves, at the very least, looking up the header of the packet in a table to determine the outgoing link. It may also involve modifications to the header of

the packet. The total time taken for all such operations is called the processing delay of the switch.

3. **Transmission delay.** The transmission delay of a link is the time it takes for a packet of size S bits to traverse the link. If the bit rate of the link is R bits/second, then the transmission delay is S/R seconds.

   We should note that the processing delay adds to the other sources of delay in a network with *store-and-forward* switches, the most common kind of network switch today. In such a switch, each data frame (packet) is stored before any processing (such as a lookup) is done and the packet then sent. In contrast, some extremely low latency switch designs are *cut-through*: as a packet arrives, the destination field in the header is used for a table lookup, and the packet is sent on the outgoing link without any storage step. In this design, the switch *pipelines* the transmission of a packet on one link with the reception on another, and the processing at one switch is pipelined with the reception on a link, so the end-to-end per-packet delay is smaller than the sum of the individual sources of delay.

   Unless mentioned explicitly, we will deal only with store-and-forward switches in this course.

4. **Queueing delay.** Queues are a fundamental data structure used in packet-switched networks to absorb bursts of data arriving for an outgoing link at speeds that are (transiently) faster than the link's bit rate. The time spent by a packet *waiting* in the queue is its queueing delay.

   Unlike the other components mentioned above, the queueing delay is usually variable. In many networks, it might also be the dominant source of delay, accounting for about 50% (or more) of the delay experienced by packets when the network is congested. In some networks, such as those with satellite links, the propagation delay could be the dominant source of delay.

■ 16.4.1 Little's Law

A common method used by engineers to analyze network performance, particularly delay and throughput (the rate at which packets are delivered), is *queueing theory*. In this course, we will use an important, widely applicable result from queueing theory, called *Little's law* (or Little's theorem).[9] It's used widely in the performance evaluation of systems ranging from communication networks to factory floors to manufacturing systems.

For any stable (i.e., where the queues aren't growing without bound) queueing system, Little's law relates the average arrival rate of items (e.g., packets), λ, the average delay experienced by an item in the queue, D, and the average number of items in the queue, N. The formula is simple and intuitive:

$$N = \lambda \times D \tag{16.1}$$

Note that if the queue is stable, then the departure rate is equal to the arrival rate.

[9] This "queueing formula" was first proved in a general setting by John D.C. Little, who is now an Institute Professor at MIT (he also received his PhD from MIT in 1955). In addition to the result that bears his name, he is a pioneer in marketing science.

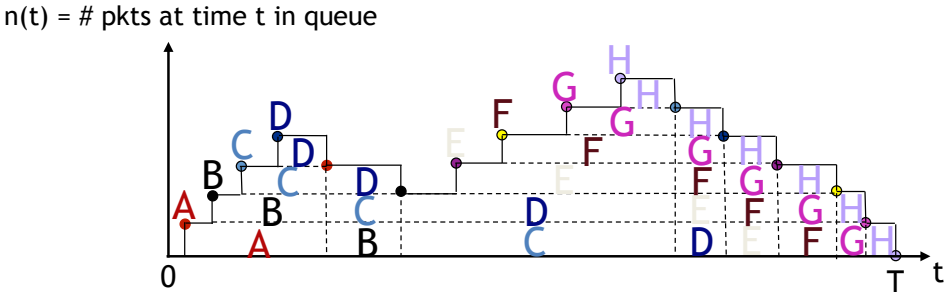


Figure 16-11: Packet arrivals into a queue, illustrating Little's law.

Example. Suppose packets arrive at an average rate of 1000 packets per second into a switch, and the rate of the outgoing link is larger than this number. (If the outgoing rate is smaller, then the queue will grow unbounded.) It doesn't matter how inter-packet arrivals are distributed; packets could arrive in weird bursts according to complicated distributions. Now, suppose there are 50 packets in the queue on average. That is, if we sample the queue size at random points in time and take the average, the number is 50 packets.

Then, from Little's law, we can conclude that **the average queueing delay experienced by a packet is 50/1000 seconds = 50 milliseconds**.

Little's law is quite remarkable because it is independent of how items (packets) arrive or are serviced by the queue. Packets could arrive according to any distribution. They can be serviced in any order, not just first-in-first-out (FIFO). They can be of any size. In fact, about the only practical requirement is that the queueing system be stable. It's a useful result that can be used profitably in back-of-the-envelope calculations to assess the performance of real systems.

Why does this result hold? Proving the result in its full generality is beyond the scope of this course, but we can show it quite easily with a few simplifying assumptions using an essentially pictorial argument. The argument is instructive and sheds some light into the dynamics of packets in a queue.

Figure 16-11 shows $n(t)$, the number of packets in a queue, as a function of time t. Each time a packet enters the queue, $n(t)$ increases by 1. Each time the packet leaves, $n(t)$ decreases by 1. The result is the step-wise curve like the one shown in the picture.

For simplicity, we will assume that the queue size is 0 at time 0 and that there is some time $T >> 0$ at which the queue empties to 0. We will also assume that the queue services jobs in FIFO order (note that the formula holds whether these assumptions are true or not).

Let P be the total number of packets forwarded by the switch in time T (obviously, in our special case when the queue fully empties, this number is the same as the number that entered the system).

Now, we need to define N, λ, and D. One can think of N as the *time average* of the number of packets in the queue; i.e.,

$$N = \sum_{t=0}^{T} n(t)/T.$$

The rate λ is simply equal to P/T, for the system processed P packets in time T.

D, the average delay, can be calculated with a little trick. Imagine taking the total area under the $n(t)$ curve and assigning it to packets as shown in Figure 16-11. That is, packets A, B, C, ... each are assigned the different rectangles shown. The height of each rectangle is 1 (i.e., one packet) and the length is the time until some packet leaves the system. Each packet's rectangle(s) last until the packet itself leaves the system.

Now, it should be clear that the time spent by any given packet is just the sum of the areas of the rectangles labeled by that packet.

Therefore, the average delay experienced by a packet, D, is simply the area under the $n(t)$ curve divided by the number of packets. That's because the total area under the curve, which is $\sum n(t)$, is the *total delay* experienced by all the packets.

Hence,

$$D = \sum_{t=0}^{T} n(t)/P.$$

From the above expressions, Little's law follows: $N = \lambda \times D$.

Little's law is useful in the analysis of networked systems because, depending on the context, one usually knows some two of the three quantities in Eq. (16.1), and is interested in the third. It is a statement about averages, and is remarkable in how little it assumes about the way in which packets arrive and are processed.

■ Problems and Questions

1. Under what conditions would circuit switching be a better network design than packet switching?

2. Which of these statements are correct?

   (a) Switches in a circuit-switched network process connection establishment and tear-down messages, whereas switches in a packet-switched network do not.

   (b) Under some circumstances, a circuit-switched network may prevent some senders from starting new conversations.

   (c) Once a connection is correctly established, a switch in a circuit-switched network can forward data correctly without requiring data frames to include a destination address.

   (d) Unlike in packet switching, switches in circuit-switched networks *do not* need any information about the network topology to function correctly.

3. Consider a switch that uses time division multiplexing (rather than statistical multiplexing) to share a link between four concurrent connections (A, B, C, and D) whose packets arrive in bursts. The link's data rate is 1 packet per time slot. Assume that the switch runs for a very long time.

   (a) The average packet arrival rates of the four connections (A through D), in packets per time slot, are 0.2, 0.2, 0.1, and 0.1 respectively. The average delays observed at the switch (in time slots) are 10, 10, 5, and 5. What are the average queue lengths of the four queues (A through D) at the switch?

(b) Connection A's packet arrival rate now changes to 0.4 packets per time slot. All the other connections have the same arrival rates and the switch runs unchanged. What are the average queue lengths of the four queues (A through D) now?

4. Annette Werker has developed a new switch. In this switch, 10% of the packets are processed on the "slow path", which incurs an average delay of 1 millisecond. All the other packets are processed on the "fast path", incurring an average delay of 0.1 milliseconds. Annette observes the switch over a period of time and finds that the average number of packets in it is 19. What is the average rate, in packets per second, at which the switch processes packets?

5. Alyssa P. Hacker has set up eight-node shared medium network running the Carrier Sense Multiple Access (CSMA) MAC protocol. The maximum data rate of the network is 10 Megabits/s. Including retries, each node sends traffic according to some unknown random process at an average rate of 1 Megabit/s per node. Alyssa measures the network's utilization and finds that it is 0.75. No packets get dropped in the network except due to collisions, and each node's average queue size is 5 packets. Each packet is 10000 bits long.

   (a) What fraction of packets sent by the nodes (including retries) experience a collision?

   (b) What is the average queueing delay, in milliseconds, experienced by a packet before it is sent over the medium?

6. Over many months, you and your friends have painstakingly collected 1,000 Gigabytes (aka 1 Terabyte) worth of movies on computers in your dorm (we won't ask where the movies came from). To avoid losing it, you'd like to back the data up on to a computer belonging to one of your friends in New York.

   You have two options:

   A. Send the data over the Internet to the computer in New York. The data rate for transmitting information across the Internet from your dorm to New York is 1 Megabyte per second.

   B. Copy the data over to a set of disks, which you can do at 100 Megabytes per second (thank you, firewire!). Then rely on the US Postal Service to send the disks by mail, which takes 7 days.

   Which of these two options (A or B) is faster? And by how much?

   Note on units:

   1 kilobyte = 10^3 bytes
   1 megabyte = 1000 kilobytes = 10^6 bytes
   1 gigabyte = 1000 megabytes = 10^9 bytes
   1 terabyte = 1000 gigbytes = 10^{12} bytes

7. Little's law can be applied to a variety of problems in other fields. Here are some simple examples for you to work out.

   (a) F freshmen enter MIT every year on average. Some leave after their SB degrees (four years), the rest leave after their MEng (five years). No one drops out (yes, really). The total number of SB and MEng students at MIT is N.

   What fraction of students do an MEng?

   (b) A hardware vendor manufactures $300 million worth of equipment per year. On average, the company has $45 million in accounts receivable. How much time elapses between invoicing and payment?

   (c) While reading a newspaper, you come across a sentence claiming that *"less than 1% of the people in the world die every year"*. Using Little's law (and some common sense!), explain whether you would agree or disagree with this claim. Assume that the number of people in the world does not decrease during the year (this assumption holds).

   (d) (This problem is actually almost related to networks.) Your friendly 6.02 professor receives 200 non-spam emails every day on average. He estimates that of these, 50 need a reply. Over a period of time, he finds that the average number of unanswered emails in his inbox that still need a reply is 100.

      i. On average, how much time does it take for the professor to send a reply to an email that needs a response?

      ii. On average, 6.02 constitutes 25% of his emails that require a reply. He responds to each 6.02 email in 60 minutes, on average. How much time on average does it take him to send a reply to any *non-6.02* email?

8. You send a stream of packets of size 1000 bytes each across a network path from Cambridge to Berkeley, at a mean rate of 1 Megabit/s. The receiver gets these packets without any loss. You find that the one-way delay is 50 ms in the absence of any queueing in the network. You find that each packet in your stream experiences a mean delay of 75 ms.

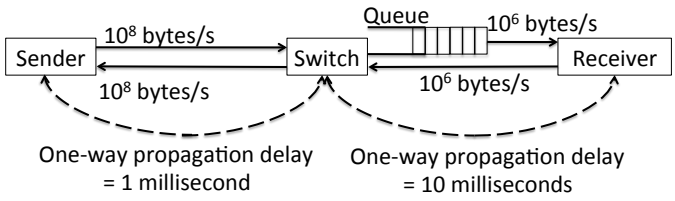
   (a) What is the mean number of packets in the queue at the bottleneck link along the path?

   You now increase the transmission rate to 1.25 Megabits/s. You find that the receiver gets packets at a rate of 1 Megabit/s, so packets are being dropped because there isn't enough space in the queue at the bottleneck link. Assume that the queue is full during your data transfer. You measure that the one-way delay for each packet in your packet stream is 125 milliseconds.

   (b) What is the packet loss rate for your stream at the bottleneck link?

   (c) Calculate the number of bytes that the queue can store.

9. Consider the network topology shown below. Assume that the processing delay at all the nodes is negligible.



(a) The sender sends two 1000-byte data packets back-to-back with a negligible inter-packet delay. The queue has no other packets. What is the time delay between the arrival of the first bit of the second packet and the first bit of the first packet at the receiver?

(b) The receiver acknowledges each 1000-byte data packet to the sender, and each acknowledgment has a size $A = 100$ bytes. What is the minimum possible round trip time between the sender and receiver? The round trip time is defined as the duration between the transmission of a packet and the receipt of an acknowledgment for it.

10. The wireless network provider at a hotel wants to make sure that anyone trying to access the network is properly authorized and their credit card charged before being allowed. This billing system has the following property: if the average number of requests currently being processed is N, then the average delay for the request is $a + bN^2$ seconds, where a and b are constants. What is the maximum rate (in requests per second) at which the billing server can serve requests?

11. *"It may be Little, but it's the law!"* Carrie Coder has set up an email server for a large email provider. The email server has two modules that process messages: the *spam filter* and the *virus scanner*. As soon as a message arrives, the spam filter processes the message. After this processing, if the message is spam, the filter throws out the message. The system sends all non-spam messages immediately to the virus scanner. If the scanner determines that the email has a virus, it throws out the message. The system then stores all non-spam, non-virus messages in the inboxes of users.

    Carrie runs her system for a few days and makes the following observations:

    1. On average, $\lambda = 10000$ messages arrive per second.
    2. On average, the spam filter has a queue size of $N_s = 5000$ messages.
    3. $s = 90\%$ of all email is found to be spam; spam is discarded.
    4. On average, the virus scanner has a queue size of $N_v = 300$ messages.
    5. $v = 20\%$ of all non-spam email is found to have a virus; these messages are discarded.

    (a) On average, in 10 seconds, how many messages are placed in the inboxes?

    (b) What is the **average delay** between the arrival of an email message to the email server and when it is ready to be placed in the inboxes? All transfer and processing delays are negligible compared to the queueing delays. Make sure to draw a picture of the system in explaining your answer. **Derive your answer**

in terms of the symbols given, plugging in all the numbers only in the final step.

12. *"Hunting in (packet) pairs:"* A sender S and receiver R are connected using a link with an unknown bit rate of C bits per second and an unknown propagation delay of D seconds. At time $t = 0$, S schedules the transmission of a **pair of packets** over the link. The bits of the two packets reach R in succession, spaced by a time determined by C. Each packet has the same known size, L bits.

    The last bit of the first packet reaches R at a known time $t = T_1$ seconds. The last bit of the second packet reaches R at a known time $t = T_2$ seconds. As you will find, this packet pair method allows us to estimate the unknown parameters, C and D, of the path.

    (a) Write an expression for T_1 in terms of L, C, and D.

    (b) At what time does the **first** bit of the **second** packet reach R? Express your answer in terms of T_1 and one or more of the other parameters given (C, D, L).

    (c) What is T_2, the time at which the **last** bit of the **second** packet reaches R? Express your answer in terms of T_1 and one or more of the other parameters given (C, D, L).

    (d) Using the previous parts, or by other means, derive expressions for the bit rate C and propagation delay D, in terms of the known parameters (T_1, T_2, L).

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: November 3, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 17
Network Routing - I Without Any Failures

This chapter and the next one discuss the key technical ideas in network routing. We start by describing the problem, and break it down into a set of sub-problems and solve them. The key ideas that you should understand by the end are:

1. Addressing and forwarding.
2. Distributed routing protocols: *distance-vector* and *link-state* protocols.
3. How routing protocols handle failures and find usable paths.

■ 17.1 The Problem

As explained in earlier chapters, sharing is fundamental to all practical network designs. We construct networks by interconnecting nodes (switches and end points) using point-to-point links and shared media. An example of a network topology is shown in Figure 17-1; the picture shows the "backbone" of the Internet2 network, which connects a large number of academic institutions in the U.S., as of early 2010. The problem we're going to discuss at length is this: what should the switches (and end points) in a packet-switched network do to ensure that a packet sent from some sender, S, in the network reaches its intended destination, D?

The word "ensure" is a strong one, as it implies some sort of guarantee. Given that packets could get lost for all sorts of reasons (queue overflows at switches, repeated collisions over shared media, and the like), we aren't going to worry about guaranteed delivery just yet.[1] Here, we are going to consider so-called *best-effort* delivery: i.e., the switches will "do their best" to try to find a way to get packets from S to D, but there are no guarantees. Indeed, we will see that in the face of a wide range of failures that we will encounter, providing even reasonable best-effort delivery will be hard enough.

[1] Subsequent chapters will address how to improve delivery reliability.

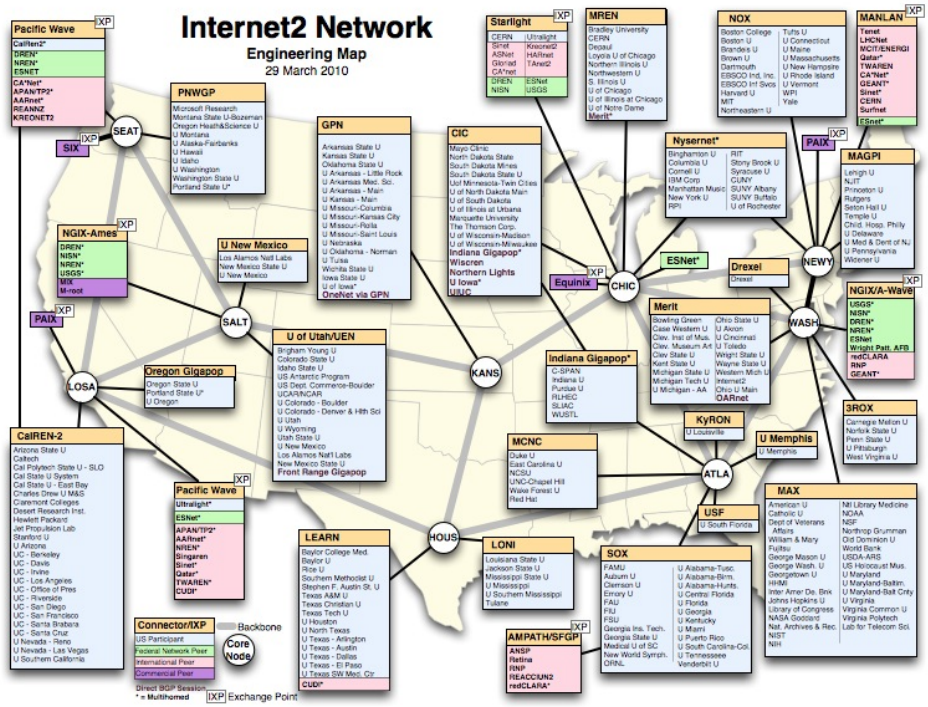


Figure 17-1: Topology of the Internet2 research and education network in the United States as of early 2010.

To solve this problem, we will model the network topology as a *graph*, a structure with nodes (vertices) connected by links (edges), as shown at the top of Figure 17-2. The nodes correspond to either switches or end points. The problem of finding paths in the network is challenging for the following reasons:

1. **Distributed information:** Each node only knows about its local connectivity, i.e., its immediate neighbors in the topology (and even determining that reliably needs a little bit of work, as we'll see). The network has to come up with a way to provide network-wide connectivity starting from this distributed information.

2. **Efficiency:** The paths found by the network should be reasonably "good"; they shouldn't be inordinately long in length, for that will increase the latency (delay) experienced by packets. For concreteness, we will assume that links have costs (these costs could model link latency, for example), and that we are interested in finding a path between any source and destination that minimizes the total cost. We will assume that all link costs are non-negative. Another aspect of efficiency that we must pay attention to is the extra network bandwidth consumed by the network in finding good paths.

3. **Failures:** Links and nodes may fail and recover arbitrarily. The network should be able to find a path if one exists, without having packets get "stuck" in the network forever because of glitches. To cope with the churn caused by the failure and recovery of links and switches, as well as by new nodes and links being set up or removed,

any solution to this problem must be dynamic and continually adapt to changing conditions.

In this description of the problem, we have used the term "network" several times while referring to the entity that solves the problem. The most common solution is for the network's switches to collectively solve the problem of finding paths that the end points' packets take. Although network designs where end points take a more active role in determining the paths for their packets have been proposed and are sometimes used, even those designs require the switches to do the hard work of finding a usable set of paths. Hence, we will focus on how switches can solve this problem. Clearly, because the information required for solving the problem is spread across different switches, the solution involves the switches cooperating with each other. Such methods are examples of *distributed computation*.

Our solution will be in three parts: first, we need a way to name the different nodes in the network. This task is called **addressing**. Second, given a packet with the name of a destination in its header we need a way for a switch to send the packet on the correct outgoing link. This task is called **forwarding**. Finally, we need a way by which the switches can determine how to send a packet to any destination, should one arrive. This task is done in the background, and continuously, building and updating the data structures required for forwarding to work properly. This background task, which will occupy most of our time, is called **routing**.

■ 17.2 Addressing and Forwarding

Clearly, to send packets to some end point, we need a way to uniquely identify the end point. Such identifiers are examples of *names*, a concept commonly used in computer systems: names provide a handle that can be used to refer to various objects. In our context, we want to name end points and switches. We will use the term *address* to refer to the name of a switch or an end point. For our purposes, the only requirement is that addresses refer to end points and switches uniquely. In large networks, we will want to constrain how addresses are assigned, and distinguish between the unique identifier of a node and its addresses. The distinction will allow us to use an address to refer to each distinct network link (aka "interface") available on a node; because a node may have multiple links connected to it, the unique name for a node is distinct from the addresses of its interfaces (if you have a computer with multiple active network interfaces, say a wireless link and an Ethernet, then that computer will have multiple addresses, one for each active interface).

In a packet-switched network, each packet sent by a sender contains the address of the destination. It also usually contains the address of the sender, which allows applications and other protocols running at the destination to send packets back. All this information is in the packet's header, which also may include some other useful fields. When a switch gets a packet, it consults a table keyed by the destination address to determine which link to send the packet on in order to reach the destination. This process is a *table lookup*, and the table in question is called the **routing table**.[2] The selected link is called the *outgoing link*.

[2]In practice, in high-speed networks, the routing table is distinct from the *forwarding table*. The former contains both the route to use for any destination and other properties of the route, such as the cost. The latter

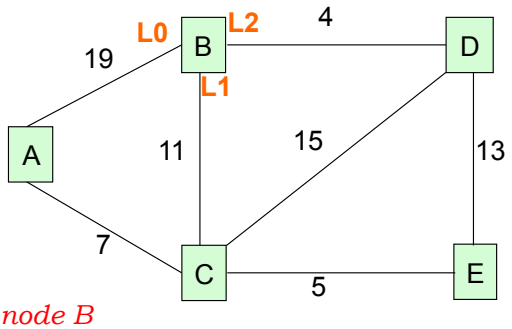


Table @ node B

Destination	*Link (next-hop)*	*Cost*
A	L1	18
B	'Self'	0
C	L1	11
D	L2	4
E	L1	16

ROUTE

Figure 17-2: A simple network topology showing the routing table at node B. The route for a destination is marked with an oval. The three links at node B are L0, L1, and L2; these names aren't visible at the other nodes but are internal to node B.

The combination of the destination address and outgoing link is called the **route** used by the switch for the destination. Note that the route is different from the *path* between source and destination in the topology; the sequence of routes at individual switches produces a sequence of links, which in turn leads to a path (assuming that the routing and forwarding procedures are working correctly). Figure 17-2 shows a routing table and routes at a node in a simple network.

Because data may be corrupted when sent over a link (uncorrected bit errors) or because of bugs in switch implementations, it is customary to include a checksum that covers the packet's header, and possibly also the data being sent.

These steps for forwarding work *as long as there are no failures* in the network. In the next chapter, we will expand these steps to combat problems caused by failures, packet losses, and other changes in the network that might cause packets to loop around in the network forever. We will use a "hop limit" field in the packet header to detect and discard packets that are being repeatedly forwarded by the nodes without finding their way to the intended destination.

is a table that contains only the route, and is usually placed in faster memory because it has to be consulted on every packet.

17.3 Overview of Routing

If you don't know where you are going, any road will take you there.
—Lewis Carroll

Routing is the process by which the switches construct their routing tables. At a high level, most routing protocols have three components:

1. **Determining neighbors**: For each node, which directly linked nodes are currently both reachable and running? We call such nodes *neighbors* of the node in the topology. A node may not be able to reach a directly linked node either because the link has failed or because the node itself has failed for some reason. A link may fail to deliver all packets (e.g., because a backhoe cuts cables), or may exhibit a high packet loss rate that prevents all or most of its packets from being delivered. For now, we will assume that each node knows who its neighbors are. In the next chapter, we will discuss a common approach, called the *HELLO protocol*, by which each node determines who its current neighbors are. The basic idea if for each node to send periodic "HELLO" messages on all its live links; any node receiving a HELLO knows that the sender of the message is currently alive and a valid neighbor.

2. **Sending advertisements**: Each node sends *routing advertisements* to its neighbors. These advertisements summarize useful information about the network topology. Each node sends these advertisements periodically, for two reasons. First, in vector protocols, periodic advertisements ensure that over time the nodes all have all the information necessary to compute correct routes. Second, in both vector and link-state protocols, periodic advertisements are the fundamental mechanism used to overcome the effects of link and node failures (as well as packet losses).

3. **Integrating advertisements**: In this step, a node processes all the advertisements it has recently heard and uses that information to produce its version of the routing table.

Because the network topology can change and because new information can become available, these three steps must run continuously, discovering the current set of neighbors, disseminating advertisements to neighbors, and adjusting the routing tables. This continual operation implies that the *state* maintained by the network switches is *soft*: that is, it refreshes periodically as updates arrive, and adapts to changes that are represented in these updates. This soft state means that the path used to reach some destination could change at any time, potentially causing a stream of packets from a source to destination to arrive reordered; on the positive side, however, the ability to refresh the route means that the system can adapt by "routing around" link and node failures. We will study how the routing protocol adapts to failures in the next chapter.

A variety of routing protocols have been developed in the literature and several different ones are used in practice. Broadly speaking, protocols fall into one of two categories depending on what they send in the advertisements and how they integrate advertisements to compute the routing table. Protocols in the first category are called **vector protocols** because each node, n, advertises to its neighbors a *vector*, with one component per destination, of information that tells the neighbors about n's route to the corresponding

destination. For example, in the simplest form of a vector protocol, n advertises its *cost* to reach each destination as a vector of destination:cost tuples. In the integration step, each recipient of the advertisement can use the advertised cost from each neighbor, together with some other information (the cost of the link from the node to the neighbor) known to the recipient, to calculate its own cost to the destination. A vector protocol that advertises such costs is also called a **distance-vector protocol**.[3]

Routing protocols in the second category are called **link-state protocols**. Here, each node advertises information about the link to its current neighbors on all its links, and each recipient re-sends this information on all of *its* links, *flooding* the information about the links through the network. Eventually, all nodes know about all the links and nodes in the topology. Then, in the integration step, each node uses an algorithm to compute the minimum-cost path to every destination in the network.

We will compare and contrast distance-vector and link-state routing protocols at the end of the next chapter, after we study how they work in detail. For now, keep in mind the following key distinction: in a distance-vector protocol (in fact, in any vector protocol), the route computation is itself distributed, while in a link-state protocol, the route computation process is done independently at each node and the dissemination of the topology of the network is done using *distributed flooding*.

The next two sections discuss the essential details of distance-vector and link-state protocols. In this chapter, *we will assume that there are no failures of nodes or links in the network*; we will assume that the only changes that can occur in the network are *additions of either nodes or links*. We will relax this assumption in the next chapter.

We will assume that all links in the network are *bi-directional* and that the costs in each direction are symmetric (i.e., the cost of a link from A to B is the same as the cost of the link from B to A, for any two directly connected nodes A and B).

17.4 A Simple Distance-Vector Protocol

The best way to understand any routing protocol is in terms of how the two distinctive steps—sending advertisements and integrating advertisements—work. In this section, we explain these two steps for a simple distance-vector protocol that achieves minimum-cost routing.

17.4.1 Distance-vector Protocol Advertisements

The advertisement in a distance-vector protocol is simple, consisting of a set of tuples as shown below:

[(dest1, cost1), (dest2, cost2), (dest3, cost3), ...]

Here, each "dest" is the address of a destination known to the node, and the corresponding "cost" is the cost of the current best path known to the node. Figure 17-3 shows an example of a network topology with the distance-vector advertisements sent by each node in *steady state*, after all the nodes have computed their routing tables. During the

[3]The actual costs may have nothing to do with physical distance, and the costs need not satisfy the triangle inequality. The reason for using the term "distance-vector" rather than "cost-vector" is historic.

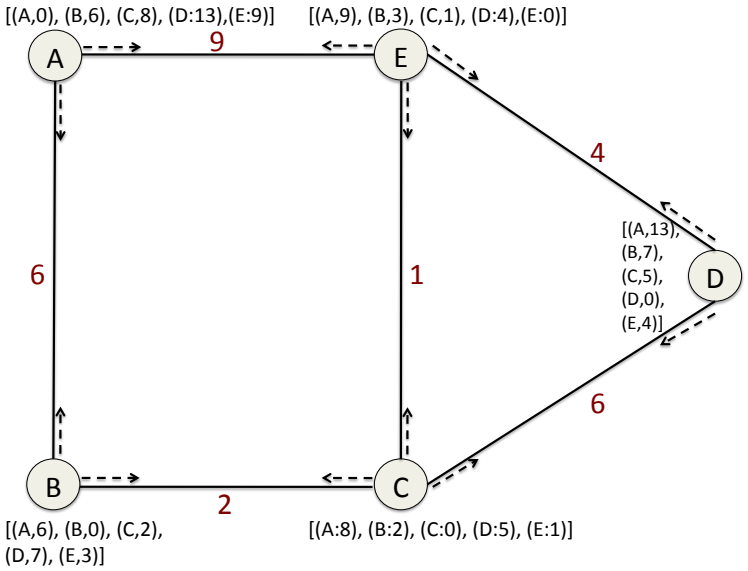


Figure 17-3: In *steady state*, each node in the the topology in this picture sends out the distance-vector advertisements shown near the node,along each link at the node.

process of computing the tables, each node advertises its *current* routing table (i.e., the destination and cost fields from the table), allowing the neighbors to make changes to their tables and advertise updated information.

What does a node do with these advertised costs? The answer lies in how the advertisements from all the neighbors are integrated by a node to produce its routing table.

■ 17.4.2 Distance-Vector Protocol: Integration Step

The key idea in the integration step uses an old observation about finding shortest-cost paths in graphs, originally due to Bellman and Ford. Consider a node n in the network and some destination d. Suppose that n hears from each of its neighbors, i, what its cost, c_i, to reach d is. Then, if n were to use the link n-i as its route to reach d, the corresponding cost would be $c_i + l_i$, where l_i is the cost of the n-i link. Hence, from n's perspective, it should choose the neighbor (link) for which the advertised cost *plus* the cost of the link from n to that neighbor is smallest. More formally, the lowest-cost path to use would be via the neighbor j, where

$$j = \arg\min_i (c_i + l_i). \tag{17.1}$$

The beautiful thing about this calculation is that it does not require the advertisements from the different neighbors to arrive synchronously. They can arrive at arbitrary times, and in any order; moreover, the integration step can run each time an advertisement arrives. The algorithm will eventually end up computing the right cost and finding the correct route (i.e., it will *converge*).

Some care must be taken while implementing this algorithm, as outlined below:

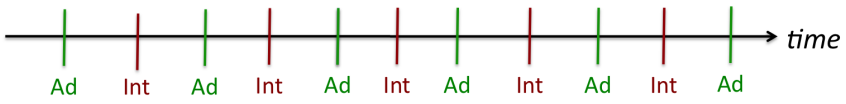


Figure 17-4: Periodic integration and advertisement steps at each node.

1. A node should update its cost and route if the new cost is smaller than the current estimate, *or* if the cost of the route currently being used changes. One question you might have is what the initial value of the cost should be before the node hears any advertisements for a destination. clearly, it should be large, a number we'll call "infinity". Later on, when we discuss failures, we will find that "infinity" for our simple distance-vector protocol can't actually be all that large. Notice that "infinity" does need to be larger than the cost of the longest minimum-cost path in the network for routing between any pair of nodes to work correctly, because a path cost of "infinity" between some two nodes means that there is no path between those two nodes.

2. In the advertisement step, each node should make sure to advertise the current best (lowest) cost along all its links.

The implementor must take further care in these steps to correctly handle packet losses, as well as link and node failures, so we will refine this step in the next chapter.

Conceptually, we can imagine the advertisement and integration processes running periodically, for example as shown in Figure 17-4. On each advertisement, a node sends the destination:cost tuples from its current routing table. In the integration step that follows, the node processes all the information received in the most recent advertisement from each neighbor to produce an updated routing table, and the subsequent advertisement step uses this updated information. Eventually, assuming no packet losses or failures or additions, the system reaches a steady state and the advertisements don't change.

■ 17.4.3 Correctness and Performance

These two steps are enough to ensure correctness in the absence of failures. To see why, first consider a network where each node has information about only itself and about no other nodes. At this time, the only information in each node's routing table is its own, with a cost of 0. In the advertisement step, a node sends that information to each of its neighbors (whose liveness is determined using the HELLO protocol). Now, the integration step runs, and each node's routing table has a set of new entries, one per neighbor, with the route set to the link along which the advertisement arrived and a path cost equal to the cost of the link.

The next advertisement sent by each node includes the node-cost pairs for each routing table entry, and the information is integrated into the routing table at a node if, and only if, the cost of the current path to a destination is larger than (or larger than or equal to) the advertised cost *plus* the cost of the link on which the advertisement arrived.

One can show the correctness of this method by induction on the length of the path. It is easy to see that if the minimum-cost path has length 1 (i.e., 1 hop), then the algorithm finds it correctly. Now suppose that the algorithm correctly computes the minimum-cost

path from a node s to any destination for which the minimum-cost path is $\leq \ell$ hops. Now consider a destination, d, whose minimum-cost path is of length $\ell + 1$. It is clear that this path may be written as $s, t, \ldots, d$, where t is a neighbor of s and the sub-path from t to d has length ℓ. By the inductive assumption, the sub-path from t to d is a path of length ℓ and therefore the algorithm must have correctly found it. The Bellman-Ford integration step at s processes all the advertisements from s's neighbors and picks the route whose link cost plus the advertised path cost is smallest. Because of this step, and the assumption that the minimum-cost path has length $\ell + 1$, the path $s, t, \ldots, d$ must be a minimum-cost route that is correctly computed by the algorithm. This completes the proof of correctness.

How well does this protocol work? In the absence of failures, and for small networks, it's quite a good protocol. It does not consume too much network bandwidth, though the size of the advertisements grows linearly with the size of the network. How long does it take for the protocol to *converge*, assuming no packet losses or other failures occur? The next chapter will discuss what it means for a protocol to "converge"; briefly, what we're asking here is the time it takes for each of the nodes to have the correct routes to every other destination. To answer this question, observe that after every integration step, assuming that advertisements and integration steps occur at the same frequency, every node obtains information about potential minimum-cost paths that are one hop longer compared to the previous integration step. This property implies that after H steps, each node will have correct minimum-cost paths to all destinations for which the minimum-cost paths are $\leq H$ hops. Hence, the convergence time in the absence of packet losses is equal to the length (i.e., number of hops) of the longest minimum-cost path in the network.

In the next chapter, when we augment the protocol to handle failures, we will calculate the bandwidth consumed by the protocol and discuss some of its shortcomings. In particular, we will discover that when link or node failures occur, this protocol behaves poorly. Unfortunately, it will turn out that many of the solutions to this problem are a two-edged sword: they will solve the problem, but do so in a way that does not work well as the size of the network grows. As a result, a distance vector protocol is limited to small networks. For these networks (tens of nodes), it is a good choice because of its relative simplicity. In practice, some examples of distance-vector protocols include RIP (Routing Information Protocol), the first distributed routing protocol ever developed for packet-switched networks; EIGRP, a proprietary protocol developed by Cisco; and a slew of wireless mesh network protocols (which are variants of the concepts described above) including some that are deployed in various places around the world.

■ 17.5 A Simple Link-State Routing Protocol

A link-state protocol may be viewed as a counter-point to distance-vector: whereas a node advertised only the best cost to each destination in the latter, in a link state protocol, a node advertises information about *all* its neighbors and the link costs to them in the advertisement step (note again: a node does not advertise information about its routes to various destinations). Moreover, upon receiving the advertisement, a node *re-broadcasts* the advertisement along all its links.[4] This process is termed *flooding*.

As a result of this flooding process, each node has a map of the entire network; this map

[4]We'll assume that the information is re-broadcast even along the link on which it came, for simplicity.

consists of the nodes and currently working links (as evidenced by the HELLO protocol at the nodes). Armed with the complete map of the network, each node can independently run a *centralized* computation to find the shortest routes to each destination in the network. As long as all the nodes optimize the same metric for each destination, the resulting routes at the different nodes will correspond to a valid path to use. In contrast, in a distance-vector protocol, the actual computation of the routes is distributed, with no node having any significant knowledge about the topology of the network. A link-state protocol distributes information about the state of each link (hence the name) and node in the topology to all the nodes, and *as long as the nodes have a consistent view of the topology and optimize the same metric, routing will work as desired.*

17.5.1 Flooding link-state advertisements

Each node uses the HELLO protocol (mentioned earlier, and which we will discuss in the next chapter in more detail) to maintain a list of current neighbors. Periodically, every `ADVERT_INTERVAL`, the node constructs a *link-state advertisement (LSA)* and sends it along all its links. The LSA has the following format:

[origin_addr, seq, (nbhr1, linkcost1), (nbhr2, linkcost2), (nbhr3, linkcost3), ...]

Here, "origin_addr" is the address of the node constructing the LSA, each "nbhr" refers to a currently active neighbor (the next chapter will describe more precisely what "currently active" means), and the "linkcost" refers to the cost of the corresponding link. An example is shown in Figure 17-5.

In addition, the LSA has a sequence number, "seq", that starts at 0 when the node turns on, and increments by 1 each time the node sends an LSA. This information is used by the flooding process, as follows. When a node receives an LSA that originated at another node, s, it first checks the sequence number of the last LSA from s. It uses the "origin_addr" field of the LSA to determine who originated the LSA. If the current sequence number is greater than the saved value for that originator, then the node *re-broadcasts the LSA on all its links*, and updates the saved value. Otherwise, it silently discards the LSA, because that same or later LSA *must* have been re-broadcast before by the node. There are various ways to improve the performance of this flooding procedure, but we will stick to this simple (and correct) process.

For now, let us assume that a node sends out an LSA every time it discovers a new neighbor or a new link gets added to the network. The next chapter will refine this step to send advertisements periodically, in order to handle failures and packet losses, as well as changes to the link costs.

17.5.2 Integration step: Dijkstra's shortest path algorithm

> *The competent programmer is fully aware of the limited size of his own skull. He therefore approaches his task with full humility, and avoids clever tricks like the plague.*
> —Edsger W. Dijkstra, in *The Humble Programmer,* CACM 1972

> *You probably know that arrogance, in computer science, is measured in nanodijkstras.*
> —Alan Kay, 1997

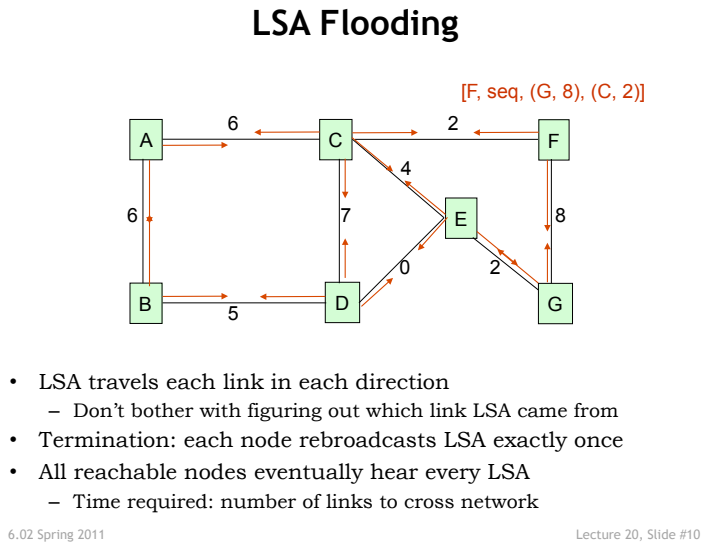


Figure 17-5: Link-state advertisement from node F in a network. The arrows show the same advertisement being re-broadcast (at different points in time) as part of the flooding process once per node, along all of the links connected to the node. The link state is shown in this example for one node; in practice, there is one of these originating from each node in the network, and re-broadcast by the other nodes.

The final step in the link-state routing protocol is to compute the minimum-cost paths from each node to every destination in the network. Each node independently performs this computation on its version of the network topology (map). As such, this step is quite straightforward because it is a centralized algorithm that doesn't require any inter-node coordination (the coordination occurred during the flooding of the advertisements).

Over the past few decades, a number of algorithms for computing various properties over graphs have been developed. In particular, there are many ways to compute minimum-cost path between any two nodes. For instance, one might use the Bellman-Ford method developed in Section 17.4. That algorithm is well-suited to a distributed implementation because it iteratively converges to the right answer as new updates arrive, but applying the algorithm on a complete graph is slower than some alternatives.

One of these alternatives was developed a few decades ago, a few years after the Bellman-Ford method, by a computer scientist named Edsger Dijkstra. Most link-state protocol implementations use Dijkstra's shortest-paths algorithm (and numerous extensions to it) in their integration step. One crucial assumption for this algorithm, which is fortunately true in most networks, is that the link costs must be non-negative.

Dijkstra's algorithm uses the following property of shortest paths: *if a shortest path from node X to node Y goes through node Z, then the sub-path from X to Z must also be a shortest path*. It is easy to see why this property must hold. If the sub-path from X to Z is not a shortest path, then one could find a shorter path from X to Y that uses a different, and shorter, sub-path from X to Z instead of the original sub-path, and then continue from Z to Y. By

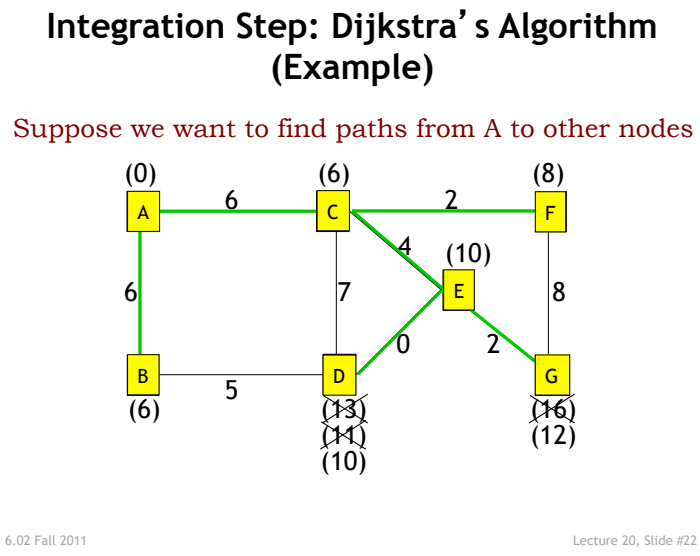


Figure 17-6: Dijkstra's shortest paths algorithm in operation, finding paths from A to all the other nodes. Initially, the set S of nodes to which the algorithm knows the shortest path is empty. Nodes are added to it in non-decreasing order of shortest path costs, with ties broken arbitrarily. In this example, nodes are added in the order (A, C, B, F, E, D, G). The numbers in parentheses near a node show the current value of `spcost` of the node as the algorithm progresses, with old values crossed out.

the same logic, the sub-path from Z to Y must also be a shortest path in the network. As a result, shortest paths can be concatenated together to form a shortest path between the nodes at the ends of the sub-paths.

This property suggests an iterative approach toward finding paths from a node, n, to all the other destinations in the network. The algorithm maintains two disjoint sets of nodes, S and $X = V - S$, where V is the set of nodes in the network. Initially S is empty. In each step, we will add one more node to S, and correspondingly remove that node from X. The node, v, we will add satisfies the following property: it is the node in X that has the shortest path from n. Thus, the algorithm adds nodes to S in non-decreasing order of shortest-path costs. The first node we will add to S is n itself, since the cost of the path from n to itself is 0 (and not larger than the path to any other node, since the links all have non-negative weights). Figure 17-6 shows an example of the algorithm in operation.

Fortunately, there is an efficient way to determine the next node to add to S from the set X. As the algorithm proceeds, it maintains the current shortest-path costs, $\texttt{spcost}(v)$, for each node v. Initially, $\texttt{spcost}(v) = \infty$ (some big number in practice) for all nodes, except for n, whose `spcost` is 0. Whenever a node u is added to S, the algorithm checks each of u's neighbors, w, to see if the current value of $\texttt{spcost}(w)$ is larger than $\texttt{spcost}(u)$ + $\texttt{linkcost}(uw)$. If it is, then update $\texttt{spcost}(w)$. Clearly, we don't need to check if the `spcost` of any other node that isn't a neighbor of u has changed because u was added to S—it couldn't have. Having done this step, we check the set X to find the next node to

add to S; as mentioned before, the node with the smallest `spcost` is selected (we break ties arbitrarily).

The last part is to remember that what the algorithm needs to produce is a *route* for each destination, which means that we need to maintain the outgoing link for each destination. To compute the route, observe that what Dijkstra's algorithm produces is a *shortest path tree* rooted at the source, n, traversing all the destination nodes in the network. (A tree is a graph that has no cycles and is connected, i.e., there is exactly one path between any two nodes, and in particular between n and every other node.) There are three kinds of nodes in the shortest path tree:

1. n itself: the route from n to n is not a link, and we will call it "Self".
2. A node v directly connected to n in the tree, whose *parent* is n. For such nodes, the route is the link connecting n to v.
3. All other nodes, w, which are not directly connected to n in the shortest path tree. For such nodes, the route to w is the same as the route to w's parent, which is the node one step closer to n along the (reverse) path in the tree from w to n. Clearly, this route will be one of n's links, but we can just set it equal to the route to w's parent and rely on the second step above to determine the link.
   We should also note that just because a node w is directly connected to n, it doesn't imply that the route from n is the direct link between them. If the cost of that link is larger than the path through another link, then we would want to use the route (outgoing link) corresponding to that better path.

■ Problems and Questions

1. Consider the network shown in Figure 17-7. The number near each link is its cost. We're interested in finding the shortest paths (taking costs into account) from S to every other node in the network.

   What is the result of running Dijkstra's shortest path algorithm on this network? To answer this question, near each node, list a pair of numbers: The first element of the pair should be the *order*, or the iteration of the algorithm in which the node is picked. The second element of each pair should be the *shortest path cost* from S to that node.

2. Alice and Bob are responsible for implementing Dijkstra's algorithm at the nodes in a network running a link-state protocol. On her nodes, Alice implements a minimum-cost algorithm. On his nodes, Bob implements a "shortest number of hops" algorithm. Give an example of a network topology with 4 or more nodes in which a routing loop occurs with Alice and Bob's implementations running simultaneously in the same network. Assume that there are no failures.

   (Note: A routing loop occurs when a group of $k \geq 1$ distinct nodes, $n_0, n_1, n_2, \ldots, n_{k-1}$ have routes such that n_i's next-hop (route) to a destination is $n_{i+1} \bmod k$.)

3. Consider any two graphs(networks) G and G' that are identical except for the costs of the links.

   (a) The cost of link l in graph G is $c_l > 0$, and the cost of the same link l in Graph G' is kc_l, where $k > 0$ is a constant. Are the shortest paths between any two nodes

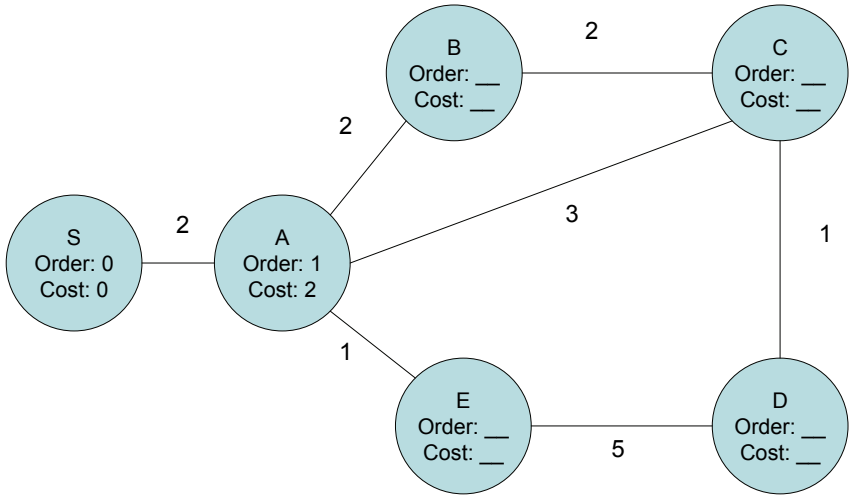


Figure 17-7: Topology for problem 1.

in the two graphs identical? Justify your answer.

(b) Now suppose that the cost of a link l in G' is $kc_l + h$, where $k > 0$ and $h > 0$ are constants. Are the shortest paths between any two nodes in the two graphs identical? Justify your answer.

4. Eager B. Eaver implements distance vector routing in his network in which the links all have arbitrary positive costs. In addition, there are at least two paths between any two nodes in the network. One node, u, has an erroneous implementation of the integration step: it takes the advertised costs from each neighbor and picks the route corresponding to the minimum advertised cost to each destination as its route to that destination, **without adding the link cost to the neighbor**. It breaks any ties arbitrarily. All the other nodes are implemented correctly.

   Let's use the term "correct route" to mean the route that corresponds to the minimum-cost path. Which of the following statements are true of Eager's network?

   (a) Only u may have incorrect routes to any other node.

   (b) Only u and u's neighbors may have incorrect routes to any other node.

   (c) In some topologies, all nodes may have correct routes.

   (d) Even if no HELLO or advertisements packets are lost and no link or node failures occur, a routing loop may occur.

5. Alyssa P. Hacker is trying to reverse engineer the trees produced by running Dijkstra's shortest paths algorithm at the nodes in the network shown in Figure 19-9 **on the left**. She doesn't know the link costs, but knows that they are all positive. All

link costs are symmetric (the same in both directions). She also knows that there is exactly one minimum-cost path between any pair of nodes in this network.

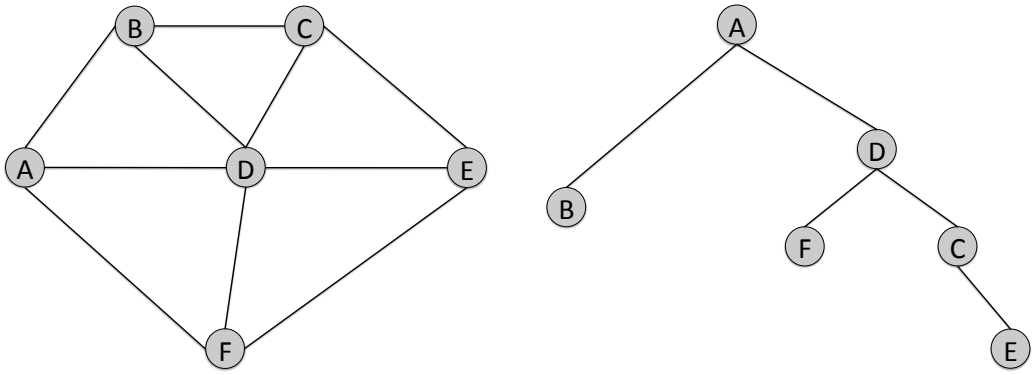


Figure 17-8: Topology for problem 5.

She discovers that the routing tree computed by Dijkstra's algorithm at node **A** looks like the picture in Figure 19-9 **on the right**. Note that the exact order in which the nodes get added in Dijkstra's algorithm is not obvious from this picture.

(a) Which of A's links has the highest cost? If there could be more than one, tell us what they are.

(b) Which of A's links has the lowest cost? If there could be more than one, tell us what they are.

Alyssa now inspects node **C**, and finds that it looks like Figure 17-9. She is sure that the bold (not dashed) links belong to the shortest path tree from node **C**, but is not sure of the dashed links.

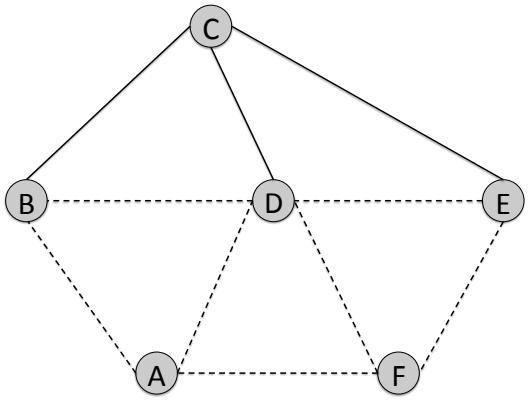


Figure 17-9: Picture for problems 5(c) and 5(d).

(c) List all the dashed links in Figure 17-9 that are **guaranteed to be** on the routing tree at node **C**.

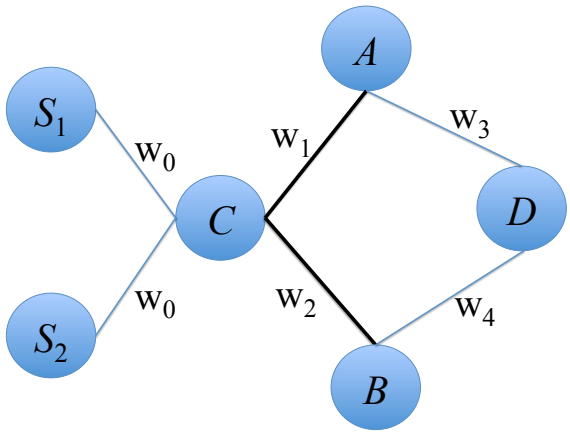


Figure 17-10: Fishnet topology for problem 6.

(d) List all the dashed links in Figure 17-9 that are **guaranteed not to be** (i.e., surely not) on the routing tree at node **C**.

6. Consider a network implementing minimum-cost routing using the distance-vector protocol. A node, S, has k neighbors, numbered 1 through k, with link cost c_i to neighbor i (all links have symmetric costs). Initially, S has no route for destination D. Then, S hears advertisements for D from each neighbor, with neighbor i advertising a cost of p_i. The node integrates these k advertisements. What is the cost for destination D in S's routing table after the integration?

7. Ben Bitdiddle is responsible for routing in FishNet, shown in Figure 17-10. He gets to pick the costs for the different links (the w's shown near the links). All the costs are non-negative.

   Goal: To ensure that the links connecting C to A and C to B, shown as darker lines, carry **equal traffic load**. All the traffic is generated by S_1 and S_2, in some unknown proportion. The rate (offered load) at which S_1 and S_2 together generate traffic for destinations A, B, and D are r_A, r_B, and r_D, respectively. Each network link has a bandwidth higher than $r_A + r_B + r_D$. There are no failures.

   Protocol: FishNet uses link-state routing; each node runs Dijkstra's algorithm to pick minimum-cost routes.

   (a) If $r_A + r_D = r_B$, then what constraints (equations or inequalities) must the link costs satisfy for the goal to be met? Explain your answer. If it's impossible to meet the goal, say why.

   (b) If $r_A = r_B = 0$ and $r_D > 0$, what constraints must the link costs satisfy for the goal to be met? Explain your answer. If it's impossible to meet the goal, say why.

8. Consider the network shown in Figure 17-11. Each node implements Dijkstra's shortest paths algorithm using the link costs shown in the picture.

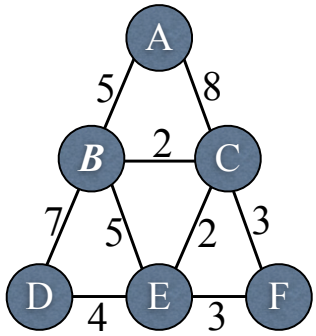


Figure 17-11: Topology for Problem 8.

(a) Initially, node **B**'s routing table contains only one entry, for itself. When **B** runs Dijkstra's algorithm, in what order are nodes added to the routing table? **List all possible answers.**

(b) Now suppose the link cost for one of the links changes but all costs remain non-negative. For each change in link cost listed below, **state whether it is possible for the *route* at node B** (i.e., the link used by **B**) for any destination to change, **and if so, name the destination(s) whose routes may change**.

i. The cost of link(A, C) increases:
ii. The cost of link(A, C) decreases:
iii. The cost of link(B, C) increases:
iv. The cost of link(B, C) decreases:

9. Eager B. Eaver implements the distance-vector protocol studied in this chapter, **but on some of the nodes**, his code sets the cost and route to each advertised destination D **differently**:

> Cost to $D = \min(\text{advertised\_cost})$ heard from each neighbor.
> Route to D = link to a neighbor that advertises the minimum cost to D.

Every node in the network periodically advertises its vector of costs to the destinations it knows about to all its neighbors. All link costs are positive.

At each node, a route for destination D is **valid** if packets using that route will eventually reach D.

At each node, a route for destination D is **correct** if packets using that route will eventually reach D along some minimum-cost path.

Assume that there are no failures and that the routing protocol has converged to produce *some* route to each destination at all the nodes.

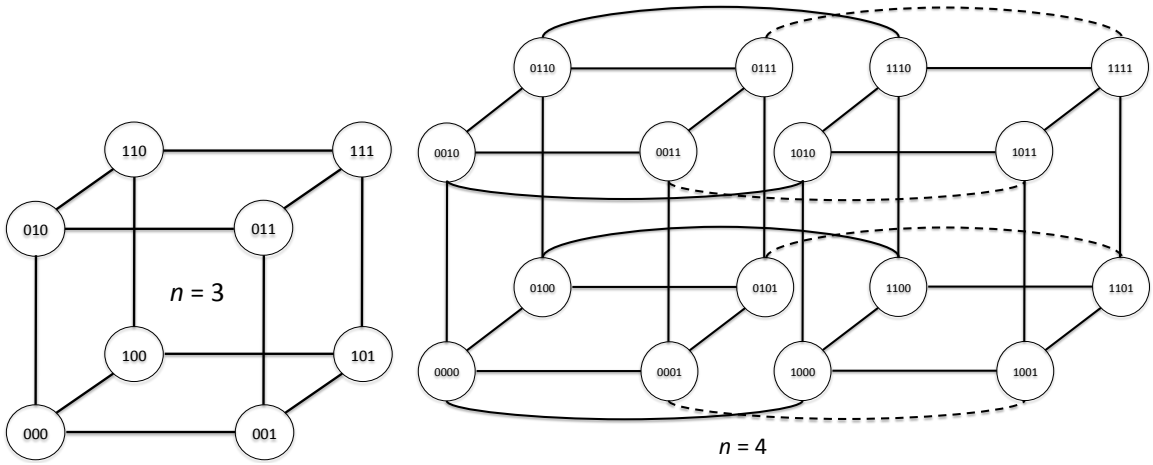
Explain whether each of these statements is True or False. Assume a network in which **at least two of the nodes** (and possibly all of the nodes) run Eager's modified version of the code, while the remaining nodes run the method discussed in this chapter.

(a) There exist networks in which some nodes will have invalid routes.

(b) There exist networks in which some nodes will **not** have correct routes.

(c) There exist networks in which **all** nodes will have correct routes.

10. The hypercube is an interesting network topology. An n-dimensional hypercube has 2^n nodes, each with a unique n-bit address. Two nodes in the hypercube are connected with a link if, and only if, their addresses have a Hamming distance of 1. The picture below shows hypercubes for $n = 3$ and 4. The solid and dashed lines are the links. We are interested in link-state routing over hypercube topologies.



(a) Suppose $n = 4$. Each node sends a link-state advertisement (LSA) periodically, starting with sequence number 0. All link costs are equal to 5. Node 1000 discovers that its link to 1001 has failed. There are no other failures. What are the contents of the **fourth LSA** originating from node 1000?

(b) Suppose $n = 4$. Three of the links at node 1000, including the link to node 1001, fail. No other failures or packet losses occur.

i. How many distinct copies of any given LSA originating from node 1000 does node 1001 receive?

ii. How many distinct copies of any given LSA originating from node 1001 does node 1000 receive?

(c) Suppose $n = 3$ and there are no failures. Each link has a **distinct**, **positive**, **integral** cost. Node 000 runs Dijkstra's algorithm (breaking ties arbitrarily) and finds that the minimum-cost path to 010 has 5 links on it. What can you say about the cost of the direct link between 000 and 010?

11. Alyssa P. Hacker runs the **link-state routing protocol** in the network shown below. Each node runs Dijkstra's algorithm to compute minimum-cost routes to all other destinations, breaking ties arbitrarily.

Answer the following questions, **explaining each answer**.

(a) In what order does C add destinations to its routing table in its execution of Dijkstra's algorithm? Give all possible answers.

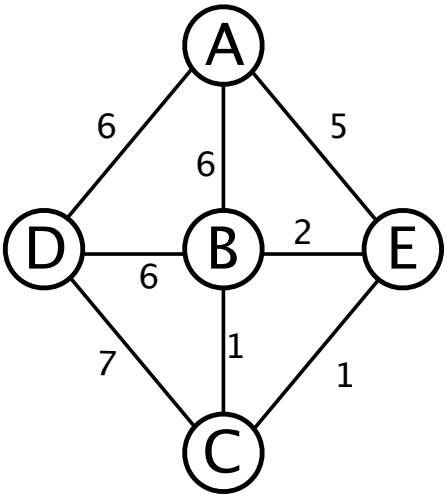


Figure 17-12: Alyssa's link-state routing problem.

(b) Suppose the cost of link $\langle CB \rangle$ increases. What is the largest value it can increase to, before **forcing** a change to any of the routes in the network? (On a tie, the old route remains.)

(c) Assume that no link-state advertisement (LSA) packets are lost on any link. When C generates a new LSA, how many copies of that LSA end up getting flooded in total over all the links of this network, using the link-state flooding protocol described in 6.02?

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: November 3, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 18
Network Routing - II
Routing Around Failures

This chapter describes the mechanisms used by distributed routing protocols to handle link and node failures, packet losses (which may cause advertisements to be lost), changes in link costs, and (as in the previous chapter) new nodes and links being added to the network. We will use the term *churn* to refer to any changes in the network topology. Our goal is to find the best paths in the face of churn. Of particular interest will be the ability to route around failures, finding the minimum-cost working paths between any two nodes from among the set of available paths.

We start by discussing what it means for a routing protocol to be correct, and define our correctness goal in the face of churn. The first step to solving the problem is to discover failures. In routing protocols, each node is responsible for discovering which of its links and corresponding nodes are still working; most routing protocols use a simple *HELLO protocol* for this task. Then, to handle failures, each node runs the *advertisement* and *integration* steps **periodically**. The idea is for each node to repeatedly propagate what it knows about the network topology to its neighbors so that any changes are propagated to all the nodes in the network. These periodic messages are the key mechanism used by routing protocols to cope with changes in the network. Of course, the routing protocol has to be robust to packet losses that cause various messages to be lost; for example, one can't use the absence of a single message to assume that a link or node has failed, for packet losses are usually far more common than actual failures.

We will see that the distributed computation done in the distance-vector protocol interacts adversely with the periodic advertisements and causes the routing protocol to not produce correct routing tables in the face of certain kinds of failures. We will present and analyze a few different solutions that overcome these adverse interactions, which extend our distance-vector protocol. We also discuss some circumstances under which link-state protocols don't work correctly. We conclude this chapter by comparing link-state and distance vector protocols.

18.1 Correctness and Convergence

In an ideal, correctly working routing protocol, two properties hold:

1. For any node, if the node has a route to a given destination, then there will be a usable path in the network topology from the node to the destination that traverses the link named in the route. We call this property *route validity*.

2. In addition, each node will have a route to each destination for which there is a usable path in the network topology, and any packet forwarded along the sequence of these routes will reach the destination (note that a route is the outgoing link at each switch; the sequence of routes corresponds to a path). We call this property *path visibility* because it is a statement of how visible the usable paths are to the switches in the network.

If these two properties hold in a network, then the network's routing protocol is said to have *converged*. It is impossible to guarantee that these properties hold at all times because it takes a non-zero amount of time for any change to propagate through the network to all nodes, and for all the nodes to come to some consensus on the state of the network. Hence, we will settle for a less ambitious—though still challenging—goal, **eventual convergence**. We define eventual convergence as follows: Given an arbitrary initial state of the network and the routing tables at time $t = 0$, suppose some sequence of failure and recovery events and other changes to the topology occur over some duration of time, τ. After $t = \tau$, suppose that no changes occur to the network topology, also that no route advertisements or HELLO messages are lost. *Then, if the routing protocol ensures that route validity and path visibility hold in the network after some finite amount of time following $t = \tau$, then the protocol is said to "eventually converge".*

In practice, it is quite possible, and indeed likely, that there will be no time τ after which there are no changes or packet losses, but even in these cases, eventual convergence is a valuable property of a routing protocol because it shows that the protocol is working toward ensuring that the routing tables are all correct. The time taken for the protocol to converge after a sequence of changes have occurred (or from some initial state) is called the *convergence time*. Thus, even though churn in real networks is possible at any time, eventual convergence is still a valuable goal.

During the time it takes for the protocol to converge, a number of things could go wrong: *routing loops* and *reduced path visibility* are two significant problems.

18.1.1 Routing Loops

Suppose the nodes in a network each want a route to some destination D. If the routes they have for D take them on a path with a sequence of nodes that form a cycle, then the network has a *routing loop*. That is, if the path resulting from the routes at each successive node forms a sequence of two or more nodes $n_1, n_2, \ldots, n_k$ in which $n_i = n_j$ for some $i \neq j$, then we have a routing loop. A routing loop violates the route validity correctness condition. If a routing loop occurs, packets sent along this path to D will be stuck in the network forever, unless other mechanisms are put in place (while packets are being *forwarded*) to "flush" such packets from the network (see Section 18.2).

18.1.2 Reduced Path Visibility

This problem usually arises when a failed link or node recovers after a failure and a previously unreachable part of the network now becomes reachable via that link or node. Because it takes time for the protocol to converge, it takes time for this information to propagate through the network and for all the nodes to correctly compute paths to nodes on the "other side" of the network. During that time, the routing tables have not yet converged, so as far as data packets are concerned, the previously unreachable part of the network still remains that way.

18.2 Alleviating Routing Loops: Hop Limits on Packets

If a packet is sent along a sequence of routers that are part of a routing loop, the packet will remain in the network until the routing loop is eliminated. The typical time scales over which routing protocols converge could be many seconds or even a few minutes, during which these packets may consume significant amounts of network bandwidth and reduce the capacity available to other packets that can be sent successfully to other destinations.

To mitigate this (hopefully transient) problem, it is customary for the packet header to include a **hop limit**. The source sets the "hop limit" field in the packet's header to some value larger than the number of hops it believes is needed to get to the destination. Each switch, before forwarding the packet, *decrements* the hop limit field by 1. If this field reaches 0, then it does not forward the packet, but drops it instead (optionally, the switch may send a diagnostic packet toward the source telling it that the switch dropped the packet because the hop limit was exceeded).

The forwarding process needs to make sure that *if* a checksum covers the hop limit field, then the checksum needs to be adjusted to reflect the decrement done to the hop-limit field.[1]

Combining this information with the rest of the forwarding steps discussed in the previous chapter, we can summarize the basic steps done while forwarding a packet in a best-effort network as follows:

1. Check the hop-limit field. If it is 0, discard the packet. Optionally, send a diagnostic packet toward the packet's source saying "hop limit exceeded"; in response, the source may decide to stop sending packets to that destination for some period of time.

2. If the hop-limit is larger than 0, then perform a routing table lookup using the destination address to determine the route for the packet. If no link is returned by the lookup or if the link is considered "not working" by the switch, then discard the packet. Otherwise, if the destination is the present node, then deliver the packet to the appropriate protocol or application running on the node. Otherwise, proceed to the next step.

3. Decrement the hop-limit by 1. Adjust the checksum (typically the header checksum) if necessary. Enqueue the packet in the queue corresponding to the outgoing link

[1] IP version 4 has such a header checksum, but IP version 6 dispenses with it, because higher-layer protocols used to provide reliable delivery have a checksum that covers portions of the IP header.

returned by the route lookup procedure. When this packet reaches the head of the queue, the switch will send the packet on the link.

■ 18.3 Neighbor Liveness: HELLO Protocol

As mentioned in the previous chapter, determining which of a node's neighbors is currently alive and working is the first step in any routing protocol. We now address this question: how does a node determine its current set of neighbors? The **HELLO protocol** solves this problem.

The HELLO protocol is simple and is named for the kind of message it uses. Each node sends a `HELLO` packet along all its links periodically. The purpose of the `HELLO` is to let the nodes at the other end of the links know that the sending node is still alive. As long as the link is working, these packets will reach. As long as a node hears another's `HELLO`, it presumes that the sending node is still operating correctly. The messages are periodic because failures could occur at any time, so we need to monitor our neighbors continuously.

When should a node remove a node at the other end of a link from its list of neighbors? If we knew how often the HELLO messages were being sent, then we could wait for a certain amount of time, and remove the node if we don't hear even one HELLO packet from it in that time. Of course, because packet losses could prevent a HELLO packet from reaching, the absence of just one (or even a small number) of HELLO packets may not be a sign that the link or node has failed. Hence, it is best to wait for enough time before deciding that a node whose HELLO packets we haven't heard should no longer be a neighbor.

For this approach to work, HELLO packets must be sent at some regularity, such that the expected number of HELLO packets within the chosen timeout is more or less the same. We call the mean time between HELLO packet transmissions the `HELLO_INTERVAL`. In practice, the actual time between these transmissions has small variance; for instance, one might pick a time drawn randomly from [`HELLO_INTERVAL` - δ, `HELLO_INTERVAL` + δ], where $\delta <$ `HELLO_INTERVAL`.

When a node doesn't hear a `HELLO` packet from a node at the other end of a direct link for some duration, $k\cdot$ `HELLO_INTERVAL`, it removes that node from its list of neighbors and considers that link "failed" (the node could have failed, or the link could just be experienced high packet loss, but we assume that it is unusable until we start hearing `HELLO` packets once more).

The choice of k is a trade-off between the time it takes to determine a failed link and the odds of falsely flagging a working link as "failed" by confusing packet loss for a failure (of course, persistent packet losses that last a long period of time should indeed be considered a link failure, but the risk here in picking a small k is that if that many successive HELLO packets are lost, we will consider the link to have failed). In practice, designers pick k by evaluating the latency before detecting a failure ($k\cdot$ `HELLO_INTERVAL`) with the probability of falsely flagging a link as failed. This probability is ℓ^k, where ℓ is the packet loss probability on the link, *assuming*—and this is a big assumption in some networks—that packet losses are independent and identically distributed.

18.4 Periodic Advertisements

The key idea that allows routing protocols to adapt to dynamic network conditions is **periodic routing advertisements** and the integration step that follows each such advertisement. This method applies to both distance-vector and link-state protocols. Each node sends an advertisement every `ADVERT_INTERVAL` seconds to its neighbors. In response, in a distance-vector protocol, each receiving node runs the integration step; in the link-state protocol each receiving node rebroadcasts the advertisement to its neighbors if it has not done so already for this advertisement. Then, every `ADVERT_INTERVAL` seconds, offset from the time of its own advertisement by `ADVERT_INTERVAL`/2 seconds, each node in the link-state protocol runs its integration step. That is, if a node sends its advertisements at times $t_1, t_2, t_3, \ldots$, where the mean value of $t_{i+1} - t_i =$`ADVERT_INTERVAL`, then the integration step runs at times $(t_1 + t_2)/2, (t_2 + t_3)/2, \ldots$. Note that one could implement a distance-vector protocol by running the integration step at such offsets, but we don't need to because the integration in that protocol is easy to run incrementally as soon as an advertisement arrives.

It is important to note that in practice the advertisements at the different nodes are *unsynchronized*. That is, each node has its own sequence of times at which it will send its advertisements. In a link-state protocol, this means that in general the time at which a node rebroadcasts an advertisement it hears from a neighbor (which originated at either the neighbor or some other node) is not the same as the time at which it originates *its own* advertisement. Similarly, in a distance-vector protocol, each node sends its advertisement asynchronously relative to every other node, and integrates advertisements coming from neighbors asynchronously as well.

18.5 Link-State Protocol Under Failure and Churn

We now argue that a link-state protocol will eventually converge (with high probability) given an arbitrary initial state at $t = 0$ and a sequence of changes to the topology that all occur within time $(0, \tau)$, assuming that each working link has a "high enough" probability of delivering a packet. To see why, observe that:

1. There exists some finite time $t_1 > \tau$ at which each node will correctly know, with high probability, which of its links and corresponding neighboring nodes are up and which have failed. Because we have assumed that there are no changes after τ and that all packets are delivered with high-enough probability, the HELLO protocol running at each node will correctly enable the neighbors to infer its liveness. The arrival of the first HELLO packet from a neighbor will provide evidence for liveness, and if the delivery probability is high enough that the chances of k successive HELLO packets to be lost before the correct link state propagates to all the nodes in the network is small, then such a time t_1 exists.

2. There exists some finite time $t_2 > t_1$ at which all the nodes have received, with high probability, at least one copy of every other node's link-state advertisement. Once a node has its own correct link state, it takes a time proportional to the diameter of the network (the number of hops in the longest shortest-path in the network) for that

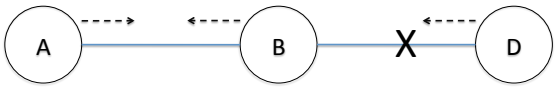


Figure 18-1: Distance-vector protocol showing the "count-to-infinity" problem (see Section 18.6 for the explanation).

advertisement to propagate to all the other nodes, assuming no packet loss. If there are losses, then notice that each node receives as many copies of the advertisement as there are neighbors, because each neighbor sends the advertisement once along each of its links. This flooding mechanism provides a built-in reliability at the cost of increased bandwidth consumption. Even if a node does not get another node's LSA, it will eventually get *some* LSA from that node given enough time, because the links have a high-enough packet delivery probability.

3. At a time roughly `ADVERT_INTERVAL`/2 after receiving every other node's correct link-state, a node will compute the correct routing table.

Thus, one can see that under good packet delivery conditions, a link-state protocol can converge to the correct routing state as soon as each node has advertised its own link-state advertisement, and each advertisement is received at least once by every other node. Thus, starting from some initial state, because each node sends an advertisement within time `ADVERT_INTERVAL` on average, the convergence time is expected to be at least this amount. We should also add a time of roughly `ADVERT_INTERVAL`/2 seconds to this quantity to account for the delay before the node actually computes the routing table. This time could be higher, if the routes are recomputed less often on average, or lower, if they are recomputed more often.

Ignoring when a node recomputes its routes, we can say that **if each node gets at least one copy of each link-state advertisement**, then the expected convergence time of the protocol is one advertisement interval plus the amount of time it takes for an LSA message to traverse the diameter of the network. Because the advertisement interval is many orders of magnitude larger than the message propagation time, the first term is dominant.

Link-state protocols are not free from routing loops, however, because packet losses could cause problems. For example, if a node A discovers that one of its links has failed, it may recompute a route to a destination via some other neighboring node, B. If B does not receive a copy of A's LSA, and if B were using the link to A as its route to the destination, then a routing loop would ensue, at least until the point when B learned about the failed link.

In general, link-state protocols are a good way to achieve fast convergence.

■ 18.6 Distance-Vector Protocol Under Failure and Churn

Unlike in the link-state protocol where the flooding was distributed but the route computation was centralized at each node, the distance-vector protocol distributes the computation too. As a result, its convergence properties are far more subtle.

Consider for instance a simple "chain" topology with three nodes, A, B, and destination D (Figure 18-1). Suppose that the routing tables are all correct at $t = 0$ and then that link between B and D fails at some time $t < \tau$. After this event, there are no further changes to the topology.

Ideally, one would like the protocol to do the following. First, B's HELLO protocol discovers the failure, and in its next routing advertisement, sends a cost of INFINITY (i.e., "unreachable") to A. In response, A would conclude that B no longer had a route to D, and remove its own route to D from its routing table. The protocol will then have converged, and the time taken for convergence not that different from the link-state case (proportional to the diameter of the network in general).

Unfortunately, things aren't so clear cut because each node in the distance-vector protocol advertises information about *all* destinations, not just those directly connected to it. What could easily have happened was that before B sent its advertisment telling A that the cost to D had become INFINITY, A's advertisement could have reached B telling B that the cost to D is 2. In response, B integrates this route into its routing table because 2 is smaller than B's own cost, which is INFINITY. You can now see the problem—B has a wrong route because it thinks A has a way of reaching D with cost 2, but it doesn't really know that A's route is based on what B had previously told him! So, now A thinks it has a route with cost 2 of reaching D and B thinks it has a route with cost $2 + 1 = 3$. The next advertisement from B will cause A to increase its own cost to $3 + 1 = 4$. Subsequently, after getting A's advertisement, B will increase its cost to 5, and so on. In fact, this mess will continue, with both nodes believing that there is some way to get to the destination D, even though there is no path in the network (i.e., the route validity property does not hold here).

There is a colorful name for this behavior: *counting to infinity*. The only way in which each node will realize that D is unreachable is for the cost to reach INFINITY. Thus, for this distance-vector protocol to converge in reasonable time, the value of INFINITY must be quite small! And, of course, INFINITY must be at least as large as the cost of the longest usable path in the network, for otherwise that routes corresponding to that path will not be found at all.

We have a problem. The distance-vector protocol was attractive because it consumed far less bandwidth than the link-state protocol, and so we thought it would be more appopriate for large networks, but now we find that INFINITY (and hence the size of networks for which the protocol is a good match) must be quite small! Is there a way out of this mess?

First, let's consider a flawed solution. Instead of B waiting for its normal advertisment time (every `ADVERT_INTERVAL` seconds on average), what if B sent news of any unreachable destination(s) as soon as its integration step concludes that a link has failed and some destination(s) has cost INFINITY? If each node propagated this "bad news" fast in its advertisement, then perhaps the problem will disappear.

Unfortunately, this approach does not work because advertisement packets could easily be lost. In our simple example, even if B sent an advertisement immediately after discovering the failure of its link to D, that message could easily get dropped and not reach A. In this case, we're back to square one, with B getting A's advertisement with cost 2, and so on. Clearly, we need a more robust solution. We consider two, in turn, each with fancy names: *split horizon routing* and *path vector routing*. Both generalize the distance-vector

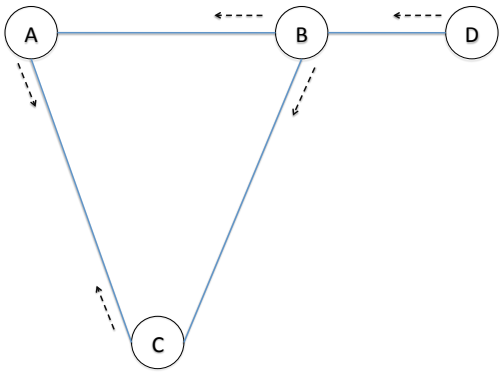


Figure 18-2: Split horizon (with or without poison reverse) doesn't prevent routing loops of three or more hops. The dashed arrows show the routing advertisements for destination D. If link BD fails, as explained in the text, it is possible for a "count-to-infinity" routing loop involving $A, B,$ and C to ensue.

protocol in elegant ways.

■ 18.7 Distance Vector with Split Horizon Routing

The idea in the split horizon extension to distance-vector routing is simple:

> *If a node A learns about the best route to a destination D from neighbor B, then A will not advertise its route for D back to B.*

In fact, one can further ensure that B will not use the route advertised by A by having A advertise a route to D with a *cost of INFINITY*. This modification is called a *poison reverse*, because the node (A) is poisoning its route for D in its advertisement to B.

It is easy to see that the two-node routing loop that showed up earlier disappears with the split horizon technique.

Unfortunately, this method does not solve the problem more generally; loops of three or more hops can persist. To see why, look at the topology in Figure 18-2. Here, B is connected to destination D, and two other nodes A and C are connected to B as well as to each other. Each node has the following correct routing state at $t = 0$: A thinks D is at cost 2 (and via B), B thinks D is at cost 1 via the direct link, and C thinks D is at cost S (and via B). Each node uses the distance-vector protocol with the split horizon technique (it doesn't matter whether they use poison reverse or not), so A and C advertise to B that their route to D has cost INFINITY. Of course, they also advertise to each other that there is a route to D with cost 2; this advertisement is useful if link AB (or BC) were to fail, because A could then use the route via C to get to D (or C could use the route via A).

Now, suppose the link BD fails at some time $t < \tau$. Ideally, if B discovers the failure and sends a cost of INFINITY to A and C in its next update, all the nodes will have the correct cost to D, and there is no routing loop. Because of the split horizon scheme, B does not have to send its advertisement immediately upon detecting the failed link, but the sooner it does, the better, for that will enable A and C to converge sooner.

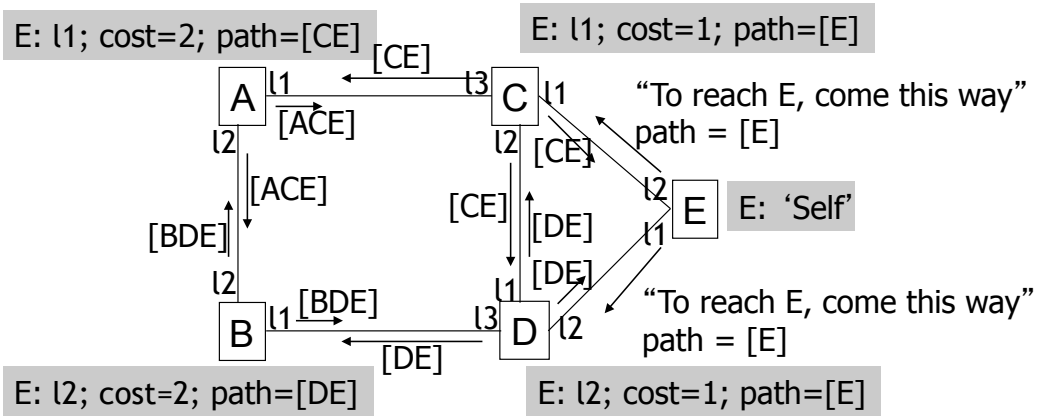


Figure 18-3: Path vector protocol example.

However, suppose B's routing advertisement with the updated cost to D (of INFINITY) reaches A, but is lost and doesn't show up at C. A now knows that there is no route of finite cost to D, but C doesn't. Now, in its next advertisement, C will advertise a route to D of cost 2 to A (and a cost of INFINITY to B because of poison reverse). In response, A will assume that C has found a better route than what A has (which is a "null" route with cost INFINITY), and integrate that into its table. In its next advertisement, A will advertise to B that it has a route of cost 3 to destination D, and B will incorporate that route at cost 4! It is easy to see now that when B advertises this route to C, it will cause C to increase its cost to 5, and so on. The count-to-infinity problem has shown up again!

Path vector routing is a good solution to this problem.

■ 18.8 Path-Vector Routing

The insight behind the path vector protocol is that a node needs to know when it is safe and correct to integrate any given advertisement into its routing table. The split horizon technique was an attempt that worked in only a limited way because it didn't prevent loops longer than two hops. The path vector technique extends the distance vector advertisement to include not only the cost, *but also the nodes along the best path from the node to the destination.* It looks like this:

[dest1 cost1 path1 dest2 cost2 path2 dest3 cost3 path3 ...]

Here, each "path" is the concatenation of the identifiers of the node along the path, with the destination showing up at the end (the opposite convention is equivalent, as long as all nodes treat the path consistently). Figure 18-3 shows an example.

The integration step at node n should now be extended to only consider an advertisement as long as n does not already appear on the advertised path. With that step, the rest of the integration step of the distance vector protocol can be used unchanged.

Given an initial state at $t = 0$ and a set of changes in $(0, \tau)$, and assuming that each link has a high-enough packet delivery probability, this path vector protocol eventually converges (with high probability) to the correct state without "counting to infinity". The

time it takes to converge when each node is interested in finding the minimum-cost path is proportional to the length of the longest minimum-cost path multiplied by the advertisement interval. The reason is as follows. Initially, each node knows nothing about the network. After one advertisement interval, it learns about its neighbors routing tables, but at this stage those tables have nothing other than the nodes themselves. Then, after the next advertisement, each node learns about all nodes two hops away and how to reach them. Eventually, after k advertisements, each node learns about how to reach all nodes k hops away, assuming of course that no packet losses occur. Hence, it takes d advertisement intervals before a node discovers routes to all the other nodes, where d is the length of the longest minimum-cost path from the node.

Compared to the distance vector protocol, the path vector protocol consumes more network bandwidth because now each node needs to send not just the cost to the destination, but also the addresses (or identifiers) of the nodes along the best path. In most large real-world networks, the number of links is large compared to the number of nodes, and the length of the minimum-cost paths grows slowly with the number of nodes (typically logarithmically). Thus, for large network, a path vector protocol is a reasonable choice.

We are now in a position to compare the link-state protocol with the two vector protocols (distance-vector and path-vector).

■ 18.9 Summary: Comparing Link-State and Vector Protocols

There is nothing either good or bad, but thinking makes it so.
—Hamlet, Act II (scene ii)

Bandwidth consumption. The total number of bytes sent in each link-state advertisement is quadratic in the number of links, while it is linear in the number of links for the distance-vector protocol.

The advertisement step in the simple distance-vector protocol consumes less bandwidth than in the simple link-state protocol. Suppose that there are n nodes and m links in the network, and that each [node pathcost] or [neighbor linkcost] tuple in an advertisement takes up k bytes (k might be 6 in practice). Each advertisement also contains a source address, which (for simplicity) we will ignore.

Then, for distance-vector, each node's advertisement has size kn. Each such advertisement shows up on every link *twice*, because each node advertises its best path cost to every destination on each of its link. Hence, the total bandwidth consumed is roughly $2knm/$`ADVERT_INTERVAL` bytes/second.

The calculation for link-state is a bit more involved. The easy part is to observe that there's a "origin_address" and sequence number of each LSA to improve the efficiency of the flooding process, which isn't needed in distance-vector. If the sequence number is ℓ bytes in size, then because each node broadcasts every other node's LSA once, the number of bytes sent is ℓn. However, this is a second-order effect; most of the bandwidth is consumed by the rest of the LSA. The rest of the LSA consists of k bytes of information *per neighbor*. Across the entire network, this quantity accounts for $k(2m)$ bytes of information, because the sum of the number of neighbors of each node in the network is $2m$. Moreover, each LSA is re-broadcast once by each node, which means that each LSA shows up *twice*

on every link. Therefore, the total number of bytes consumed in flooding the LSAs over the network to all the nodes is $k(2m)(2m) = 4km^2$. Putting it together with the bandwidth consumed by the sequence number field, we find that the total bandwidth consumed is $(4km^2 + 2\ell mn)/$`ADVERT_INTERVAL` bytes/second.

It is easy to see that there is no connected network in which the bandwidth consumed by the simple link-state protocol is lower than the simple distance-vector protocol; the important point is that the former is quadratic in the number of links, while the latter depends on the product of the number of nodes and number of links.

Convergence time. The convergence time of our distance vector and path vector protocols can be as large as the length of the longest minimum-cost path in the network multiplied by the advertisement interval. The convergence time of our link-state protocol is roughly one advertisement interval.

Robustness to misconfiguration. In a vector protocol, each node advertises costs and/or paths to all destinations. As such, an error or misconfiguration can cause a node to wrongly advertise a good route to a destination that the node does not actually have a good route for. In the worst case, it can cause all the traffic being sent to that destination to be hijacked and possibly "black holed" (i.e., not reach the intended destination). This kind of problem has been observed on the Internet from time to time. In contrast, the link-state protocol only advertises each node's immediate links. Of course, each node also re-broadcasts the advertisements, but it is harder for any given erroneous node to wreak the same kind of havoc that a small error or misconfiguration in a vector protocol can.

In practice, link-state protocols are used in smaller networks typically within a single company (enterprise) network. The routing between different autonomously operating networks in the Internet uses a path vector protocol. Variants of distance vector protocols that guarantee loop-freedom are used in some small networks, including some wireless "mesh" networks built out of short-range (WiFi) radios.

Problems and Questions

1. Why does the link-state advertisement include a sequence number?

2. What is the purpose of the hop limit field in packet headers? Is that field used in routing or in forwarding?

3. Describe clearly why the convergence time of our distance vector protocol can be as large as the length of the longest minimum-cost path in the network.

4. Suppose a link connecting two nodes in a network drops packets independently with probability 10%. If we want to detect a link failure with a probability of falsely reporting a failure of $\leq 0.1\%$, and the HELLO messages are sent once every 10 seconds, then how much time does it take to determine that a link has failed?

5. You've set up a 6-node connected network topology in your home, with nodes named $A, B, \ldots, F$. Inspecting A's routing table, you find that some entries have been mysteriously erased (shown with "?" below), but you find the following entries:

Destination	Cost	Next-hop
B	3	C
C	2	?
D	4	E
E	2	?
F	1	?

   Each link has a cost of either 1 or 2 and link costs are symmetric (the cost from X to Y is the same as the cost from Y to X). The routing table entries correspond to minimum-cost routes.

   (a) Draw a network topology with the *smallest number of links* that is consistent with the routing table entries shown above and the cost information provided. Label each node and show each link cost clearly.

   (b) You know that there could be other links in the topology. To find out, you now go and inspect D's routing table, but it is mysteriously empty. What is the smallest *possible* value for the cost of the path from D to F in your home network topology? (Assume that any two nodes may *possibly* be directly connected to answer this question.)

6. A network with N nodes and N bi-directional links is connected in a ring as shown in Figure 18-4, where N is an even number. The network runs a distance-vector protocol in which the advertisement step at each node runs when the local time is $T * i$ seconds and the integration step runs when the local time is $T * i + \frac{T}{2}$ seconds, $(i = 1, 2, \ldots)$. Each advertisement takes time δ to reach a neighbor. Each node has a separate clock and **time is not synchronized** between the different nodes.

   Suppose that at some time t after the routing has converged, node $N + 1$ is inserted into the ring, as shown in the figure above. Assume that there are no other changes

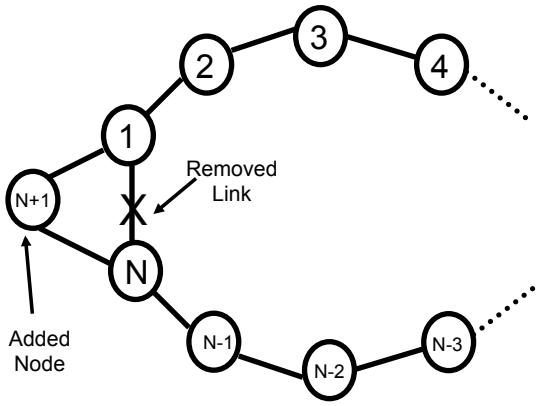


Figure 18-4: The ring network with N nodes (N is even).

in the network topology and no packet losses. Also assume that nodes 1 and N update their routing tables at time t to include node $N+1$, and then rely on their next scheduled advertisements to propagate this new information.

(a) What is the minimum time before every node in the network has a route to node $N+1$?

(b) What is the maximum time before every node in the network has a route to node $N+1$?

7. Alyssa P. Hacker manages MIT's internal network that runs link-state routing. She wants to experiment with a few possible routing strategies. Listed below are the names of four strategies and a brief description of what each one does.

(a) MinCost: Every node picks the path that has the smallest sum of link costs along the path. (This is the minimum cost routing you implemented in the lab).

(b) MinHop: Every node picks the path with the smallest number of hops (irrespective of what the cost on the links is).

(c) SecondMinCost: Every node picks the path with the second lowest sum of link costs. That is, every node picks the second best path with respect to path costs.

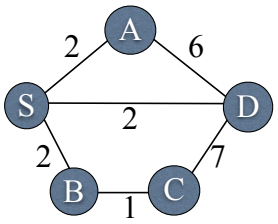
(d) MinCostSquared: Every node picks the path that has the smallest sum of squares of link costs along the path.

Assume that sufficient information is exchanged in the link state advertisements, so that every node has complete information about the entire network and can correctly implement the strategies above. You can also assume that a link's properties don't change, e.g., it doesn't fail.

(a) Help Alyssa figure out which of these strategies will work correctly, and which will result in routing with loops. In case of strategies that do result in routing loops, come up with an example network topology with a routing loop to convince Alyssa.

(b) How would you implement MinCostSquared in a distance-vector protocol? Specify what the advertisements should contain and what the integration step must do.

8. Alyssa P. Hacker implements the 6.02 distance-vector protocol on the network shown below. Each node has its own local clock, which may not be synchronized with any other node's clock. Each node sends its distance-vector advertisement every 100 seconds. When a node receives an advertisement, it immediately integrates it. The time to send a message on a link and to integrate advertisements is negligible. No advertisements are lost. There is no HELLO protocol in this network.



(a) At time 0, all the nodes **except** D are up and running. At time 10 seconds, node D turns on and immediately sends a route advertisement for itself to all its neighbors. What is the *minimum time* at which *each of the other nodes* is **guaranteed** to have a correct routing table entry corresponding to a minimum-cost path to reach D? Justify your answers.

(b) If every node sends packets to destination D, and to no other destination, which link would carry the most traffic?

Alyssa is unhappy that one of the links in the network carries a large amount of traffic when all the nodes are sending packets to D. She decides to overcome this limitation with Alyssa's Vector Protocol (AVP). In AVP, S *lies*, advertising a "path cost" for destination D that is *different* from the sum of the link costs along the path used to reach D. All the other nodes implement the standard distance-vector protocol, not AVP.

(c) What is the *smallest* numerical value of the cost that S should advertise for D along each of its links, to **guarantee** that only its own traffic for D uses its direct link to D? Assume that all advertised costs are integers; if two path costs are equal, one can't be sure which path will be taken.

9. Help Ben Bitdiddle answer these questions about the distance-vector protocol he runs on the network shown in Figure 18-5. The link costs are shown near each link. Ben is interested in minimum-cost routes to destination node D.

Each node sends a distance-vector advertisement to all its neighbors at times $0, T, 2T, \ldots$. Each node integrates advertisements at times $T/2, 3T/2, 5T/2, \ldots$. You may assume that all clocks are synchronized. The time to transmit an advertisement over a link is negligible. There are no failures or packet losses.

At each node, a route for destination D is **valid** if packets using that route will eventually reach D.

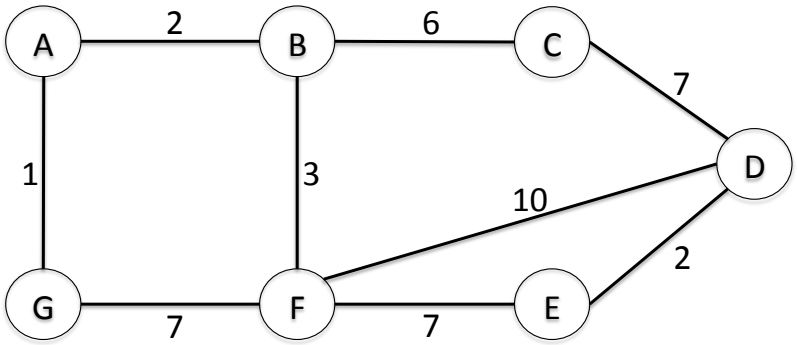


Figure 18-5: Time to converge = ?

At each node, a route for destination D is **correct** if packets using that route will eventually reach D along some minimum-cost path.

(a) At what time will **all** nodes have integrated a valid route to D into their routing tables? What node is the **last one** to integrate a valid route to D? Answer both questions.

(b) At what time will **all** nodes have integrated a correct (minimum-cost) route to D into their routing tables? What node is the **last one** to integrate a correct route to D? Answer both questions.

10. *Go Ahead, Make My Route:* Jack Ripper uses a minimum-cost distance-vector routing protocol in the network shown in Figure 18-6. Each link cost (not shown) is a **positive integer** and is the same in each direction of the link. Jack sets "infinity" to 32 in the protocol. After all routes have converged (breaking ties arbitrarily), F's routing table is as follows:

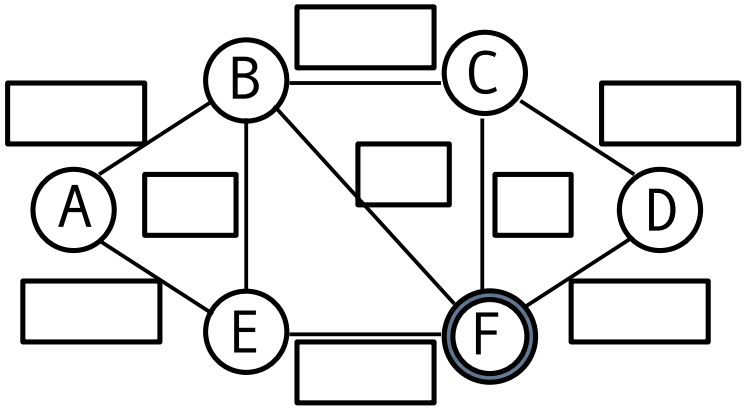


Figure 18-6: Distance vector topology in Jack Ripper's network.

Using the information provided, answer these questions:

(a) Fill in the two missing blanks in the table above.

Destination	Cost	Route
A	6	link $\langle FC \rangle$
B	4	link $\langle FC \rangle$
C	3	_____
D	5	link $\langle FD \rangle$
E	1	_____

(b) For **each link** in the picture, write the link's cost in the box near the link. Each cost is either a positive integer or an expression of the form "$< c, \leq c, \geq c$, or $> c$", for some integer c.

(c) Suppose link $\langle FE \rangle$ fails, but there are no other changes. When the protocol converges, what will F's **route** (not path) to E be? (If there is no route, say "no route".)

(d) Now suppose links $\langle BC \rangle$ and $\langle BF \rangle$ **also** fail soon after link $\langle FE \rangle$ fails. There are no packet losses. In the **worst case**, C and F enter a "count-to-infinity" phase. How many distinct route advertisements (with different costs) must C hear from F, before C determines that it does not have any valid route to node A?

11. Alyssa P. Hacker runs the **link-state routing protocol** in the network shown below. Each node runs Dijkstra's algorithm to compute minimum-cost routes to all other destinations, breaking ties arbitrarily.

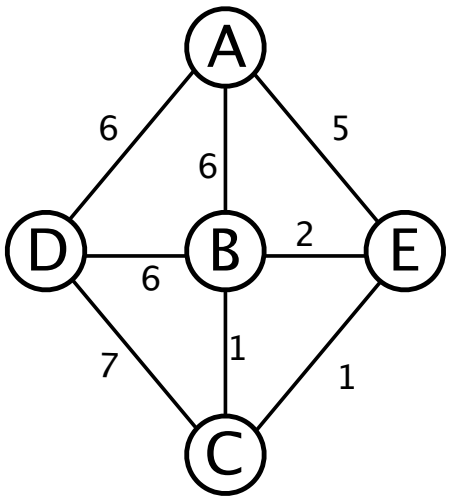


Figure 18-7: Network in Alyssa's link-state protocol.

The links in Alyssa's network are unreliable; on each link, any packet sent over the link is delivered with some probability, p, to the other end of the link, independent of all other events ($0 < p < 1$). Suppose links $\langle CE \rangle$ and $\langle BD \rangle$ fail.

Answer the following questions.

(a) How do C and E discover that the link has failed? How does the method work?

(b) Over this unreliable network, link state advertisements (LSAs) are lost according to the probabilities mentioned above. Owing to a bug in its software, E **does not originate any LSA of its own or flood them**, but all other nodes (except E) work correctly. Calculate the probability that A learns that link $\langle CE \rangle$ has failed from the **first** LSA that originates from C after C discovers that link $\langle CE \rangle$ has failed. Note that link $\langle BD \rangle$ has also failed.

(c) Suppose only link $\langle CE \rangle$ had failed, **but not** $\langle BD \rangle$, which like the other surviving links can delivery packets successfully with probability p. Now, would the answer to part (b) above **increase**, **decrease**, or **remain the same**? Why? (No math necessary.)

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: November 3, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 19
Reliable Data Transport Protocols

Packets in a *best-effort network* lead a rough life. They can be lost for any number of reasons, including queue overflows at switches because of congestion, repeated collisions over shared media, routing failures, and uncorrectable bit errors. In addition, packets can arrive out-of-order at the destination because different packets sent in sequence take different paths or because some switch en route reorders packets for some reason. They usually experience variable delays, especially whenever they encounter a queue. In some cases, the underlying network may even duplicate packets.

Many applications, such as Web page downloads, file transfers, and interactive terminal sessions would like a **reliable, in-order** stream of data, receiving exactly one copy of each byte in the same order in which it was sent. A **reliable transport protocol** does the job of hiding the vagaries of a best-effort network—packet losses, reordered packets, and duplicate packets—from the application, and provides it the abstraction of a reliable packet stream. We will develop protocols that also provide in-order delivery.

A large number of protocols have been developed that various applications use, and there are several ways to provide a reliable, in-order abstraction. This chapter will not discuss them all, but will instead discuss two protocols in some detail. The first protocol, called **stop-and-wait**, will solve the problem in perhaps the simplest possible way that works, but do so somewhat inefficiently. The second protocol will augment the first one with a **sliding window** to significantly improve performance.

All reliable transport protocols use the same powerful ideas: *redundancy to cope with packet losses* and *receiver buffering to cope with reordering*, and most use *adaptive timers*. The tricky part is figuring out exactly how to apply redundancy in the form of packet *retransmissions*, in working out exactly when retransmissions should be done, and in achieving good performance. This chapter will study these issues, and discuss ways in which a reliable transport protocol can achieve high throughput.

■ 19.1 The Problem

The problem we're going to solve is relatively easy to state. A sender application wants to send a stream of packets to a receiver application over a best-effort network, which

can drop packets arbitrarily, reorder them arbitrarily, delay them arbitrarily, and possibly even duplicate packets. The receiver wants the packets in exactly the same order in which the sender sent them, and wants exactly one copy of each packet.[1] Our goal is to devise mechanisms at the sending and receiving nodes to achieve what the receiver wants. These mechanisms involve rules between the sender and receiver, which constitute the protocol. In addition to correctness, we will be interested in calculating the throughput of our protocols, and in coming up with ways to maximize it.

All mechanisms to recover from losses, whether they are caused by packet drops or corrupted bits, employ *redundancy*. We have already studied *error-correcting codes* such as linear block codes and convolutional codes to mitigate the effect of bit errors. In principle, one could apply similar coding techniques over packets (rather than over bits) to recover from packet losses (as opposed to bit corruption). We are, however, interested not just in a scheme to reduce the effective packet loss rate, but to eliminate their effects altogether, and recover all lost packets. We are also able to rely on *feedback* from the receiver that can help the sender determine what to send at any point in time, in order to achieve that goal. Therefore, we will focus on carefully using *retransmissions* to recover from packet losses; one may combine retransmissions and error-correcting codes to produce a protocol that can further improve throughput under certain conditions. In general, experience has shown that if packet losses are not persistent and occur in bursts, and if latencies are not excessively long (i.e., not multiple seconds long), retransmissions by themselves are enough to recover from losses and achieve good throughput. Most practical reliable data transport protocols running over Internet paths today use only retransmissions on packets (individual links usually use the error correction methods, such as the ones we studied earlier, and may also augment them with a limited number of retransmissions to reduce the link-level packet loss rate.

We will develop the key ideas for two kinds of reliable data transport protocols: **stop-and-wait** and **sliding window with a fixed window size**. We will use the word "sender" to refer to the sending side of the transport protocol and the word "receiver" to refer to the receiving side. We will use "sender application" and "receiver application" to refer to the processes (applications) that would like to send and receive data in a reliable, in-order manner.

■ 19.2 Stop-and-Wait Protocol

The high-level idea in this protocol is simple. The sender attaches a *transport-layer header* to every data packet, which includes a *unique identifier* for the data packet (the transport-layer header is distinct from the *network-layer* packet header that contains the destination address, hop limit, and header checksum discussed in Chapters 17 and 18). Ideally, this unique identifier will never be reused for two different packets on the same stream.[2] The

[1] The reason for the "exactly one copy" requirement is that the mechanism used to solve the problem will end up retransmitting packets, so duplicates may occur that need to be filtered out. In some networks, it is possible that some links may end up duplicating packets because of mechanisms they employ to improve the packet delivery probability or bit-error rate over the link.

[2] In an ideal implementation, such reuse will never occur. In practice, however, a transport protocol may use a sequence number field whose width is not large enough and sequence numbers may wrap-around. In this case, it is important to ensure that two distinct unacknowledged data packets never have the same

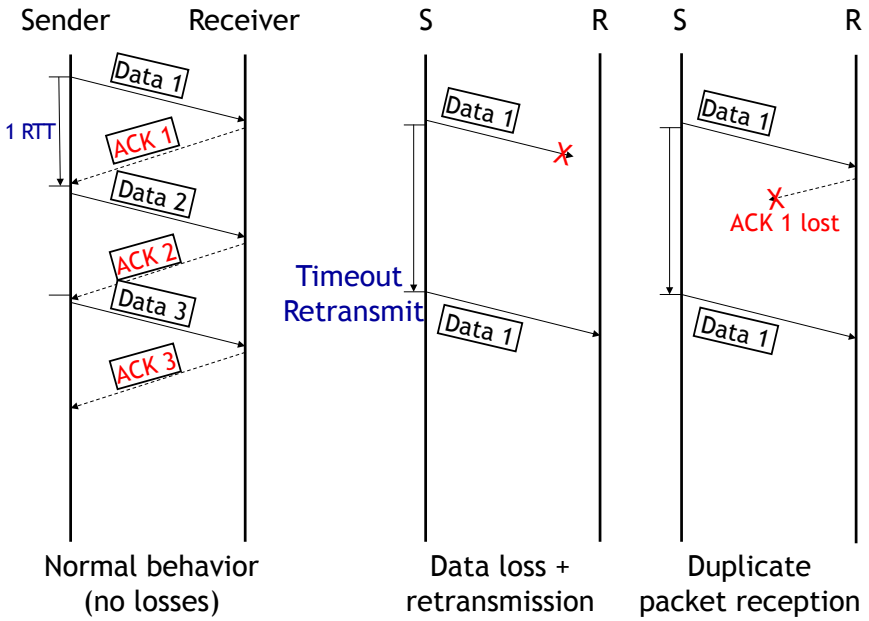


Figure 19-1: The stop-and-wait protocol. Each picture has a sender timeline and a receiver timeline. Time starts at the top of each vertical line and increases moving downward. The picture on the left shows what happens when there are no losses; the middle shows what happens on a data packet loss; and the right shows how duplicate packets may arrive at the receiver because of an ACK loss.

receiver, upon receiving the data packet with identifier k, will send an *acknowledgment* (ACK) to the sender; the header of this ACK contains k, so the receiver communicates "I got data packet k" to the sender. Both data packets and ACKs may get lost in the network.

In the stop-and-wait protocol, the sender sends the next data packet on the stream if, and only if, it receives an ACK for k. If it does not get an ACK within some period of time, called the *timeout*, the sender *retransmits* data packet k.

The receiver's job is to deliver each data packet it receives to the receiver application. Figure 19-1 shows the basic operation of the protocol when packets are not lost (left) and when data packets are lost (right).

Three properties of this protocol bear some discussion:

1. how to pick unique identifiers,

2. why this protocol may deliver duplicate data packets to the receiver application, and how the receiver can prevent that from occurring, and

3. how to pick the timeout.

We discuss each of these in turn below.

sequence number.

19.2.1 Selecting Unique Identifiers: Sequence Numbers

The sender may pick any unique identifier for a data packet. In most transport protocols, a convenient and effective choice of unique identifier is to use an *incrementing sequence number*. The simplest way to achieve this goal is for the sender and receiver to agree on the initial value of the identifier (which for our purposes will be taken to be 1), and then increment the identifier by 1 for each subsequent *new* data packet sent. Thus, the data packet sent after the ACK for k is received by the sender will have identifier $k + 1$. These incrementing identifiers are called **sequence numbers**.

In practice, transport protocols like TCP (Transmission Control Protocol), the standard Internet protocol for reliable data delivery, devote considerable effort to picking a good initial sequence number to avoid overlaps with previous instantiations of reliable streams between the same communicating processes. We won't worry about these complications in this chapter, except to note that establishing and properly terminating these streams (aka connections) reliably is a non-trivial problem. TCP also uses a sequence number that identifies the *starting byte offset* of the packet in the stream, to handle variable packet sizes.

19.2.2 Semantics of this Stop-and-Wait Protocol

It is easy to see that the stop-and-wait protocol achieves reliable data delivery as long as each of the links along the path have a non-zero packet delivery probability. However, it does not achieve *exactly once* semantics; its semantics are *at least once*—i.e., each packet will be delivered to the receiver application either once or *more than once*.

One reason is that the network could drop ACKs, as shown in Figure 19-1 (right). A data packet may have reached the receiver, but the ACK doesn't reach the sender, and the sender will then timeout and retransmit the data packet. The receiver will get multiple copies of the data packet, and deliver both to the receiver application. Another reason is that the sender might have timed out, but the original data packet may not actually have been lost. Such a retransmission is called a *spurious retransmission*, and is a waste of bandwidth. The sender may strive to reduce the number of spurious retransmissions, but it is impossible to eliminate them in general.

Preventing duplicates: The solution to the problem of duplicate data packets arriving at the receiver is for the receiver to keep track of the last *in-sequence* data packet it has delivered to the application. At the receiver, let us maintain the sequence number of the last in-sequence data packet in the variable `rcv_seqnum`. If a data packet with sequence number less than or equal to `rcv_seqnum` arrives, then the receiver sends an ACK for the packet and discards it. Note that the only way a data packet with sequence number *smaller* than `rcv_seqnum` can arrive is if there were reordering in the network and the receiver gets an old data packet; for such packets, the receiver can safely not send an ACK because it knows that the sender knows about the receipt of the packet and has sent subsequent packets. This method prevents duplicate packets from being delivered to the receiving application.

If a data packet with sequence number `rcv_seqnum` + 1 arrives, then the receiver sends an ACK to the sender, delivers the data packet to the application, and increments `rcv_seqnum`. Note that a data packet with sequence number greater than `rcv_seqnum`

+ 1 should never arrive in this stop-and-wait protocol because that would imply that the sender got an ACK for `rcv_seqnum` + 1, but such an ACK would have been sent only if the receiver got the corresponding data packet. So, if such a data packet were to arrive, then *there must be a bug in the implementation* of either the sender or the receiver in this stop-and-wait protocol.

With this modification, the stop-and-wait protocol guarantees exactly-once delivery to the application.[3]

19.2.3 Setting Timeouts

The final design issue that we need to nail down in our stop-and-wait protocol is setting the value of the timeout. How soon after the transmission of a packet should the sender conclude that the data packet (or the ACK) was lost, and go ahead and retransmit? One approach might be to use some constant, but then the question is what it should be set to. Too small, and the sender may end up retransmitting data packets before giving enough time for the ACK for the original transmission to arrive, wasting network bandwidth. Too large, and one ends up wasting network bandwidth and simply idling before retransmitting.

The natural time-scale in the protocol is the time between the transmission of a data packet and the arrival of the ACK for the packet. This time is called the **round-trip time**, or **RTT**, and plays a crucial role in all reliable transport protocols. A good value of the timeout must clearly depend on the RTT; it makes no sense to use a timeout that is not bigger than the mean RTT (and in fact, it must be quite a bit bigger than the average, as we'll see).

The other reason the RTT is an important concept is that the throughput (in packets per second) achieved by the stop-and-wait protocol is inversely proportional to the RTT (see Section 19.4). In fact, the throughput of the sliding window protocol also depends on the RTT, as we will see.

The next section describes a procedure to estimate the RTT and set sender timeouts. This technique is general and applies to a variety of protocols, including both stop-and-wait and sliding window.

19.3 Adaptive RTT Estimation and Setting Timeouts

The RTT experienced by packets is variable because the delays in a best-effort network are variable. An example is shown in Figure 19-2, which shows the RTT of an Internet path between two hosts (blue) and the packet loss rate (red), both as a function of the time-of-day. The "rtt median-filtered" curve is the median RTT computed over a recent window of samples, and you can see that even that varies quite a bit. Picking a timeout equal to simply the mean or median RTT is not a good idea because there will be many RTT samples that are larger than the mean (or median), and we don't want to timeout prematurely and send *spurious retransmissions*.

[3] We are assuming here that the sender and receiver nodes and processes don't crash and restart; handling those cases make "exactly once" semantics considerably harder than described here and require stable storage that persists across crashes.

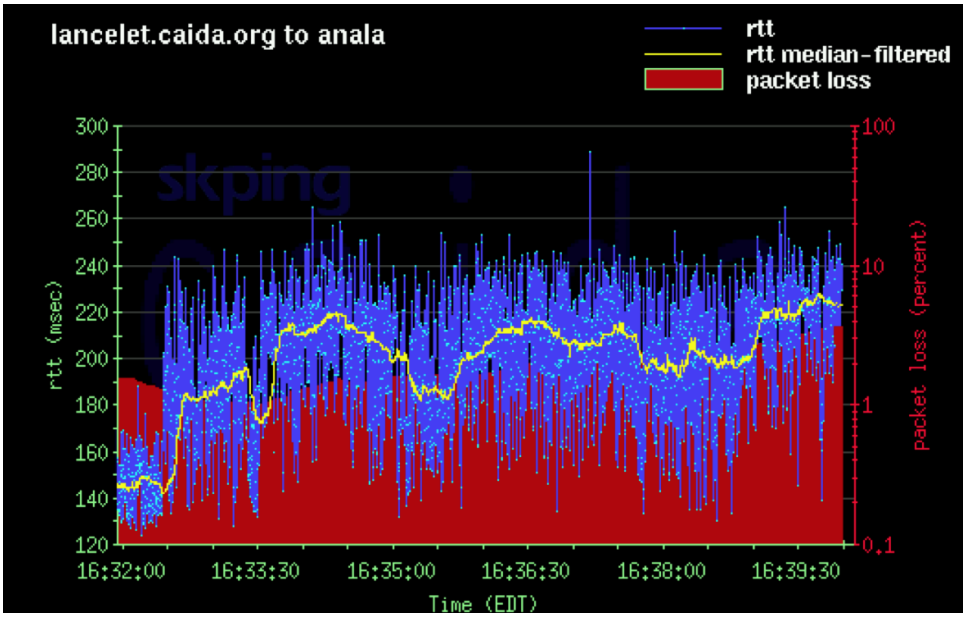


Figure 19-2: RTT variations are pronounced in many networks.

A good solution to the problem of picking the timeout value uses two tools we have seen earlier in the course: *probability distributions* (in our case, of the RTT estimates) and *a simple filter design*.

Suppose we are interested in estimating a good timeout *post facto*: i.e., suppose we run the protocol and collect a sequence of RTT samples, how would one use these values to pick a good timeout? We can take all the RTT samples and plot them as a probability distribution, and then see how any given timeout value will have performed in terms of the probability of a spurious retransmission. If the timeout value is T, then this probability may be estimated as the area under the curve to the right of "T" in the picture on the left of Figure 19-3, which shows the histogram of RTT samples. Equivalently, if we look at the cumulative distribution function of the RTT samples (the picture on the right of Figure 19-3, the probability of a spurious retransmission may be assumed to be the value of the y-axis corresponding to a value of T on the x-axis.

Real-world distributions of RTT are not actually Gaussian, but an interesting property of all distributions is that if you pick a threshold that is a sufficient number of standard deviations greater than the mean, the tail probability of a sample exceeding that threshold can be made arbitrarily small. (For the mathematically inclined, a useful result for arbitrary distributions is Chebyshev's inequality, which you might have seen in other courses already (or soon will): $P(|X - \mu| \geq k\sigma) \leq 1/k^2$, where μ is the mean and σ the standard deviation of the distribution. For Gaussians, the tail probability falls off *much faster* than $1/k^2$; for instance, when $k = 2$, the Gaussian tail probability is only about 0.05 and when $k = 3$, the tail probability is about 0.003.)

The protocol designer can use past RTT samples to determine an RTT cut-off so that only a small fraction f of the samples are larger. The choice of f depends on what spurious retransmission rate one is willing to tolerate, and depending on the protocol, the cost of such an action might be small or large. Empirically, Internet transport protocols tend to

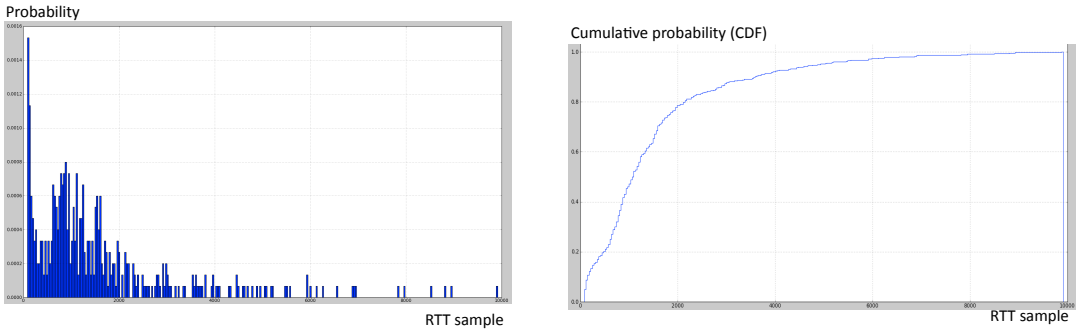


Figure 19-3: RTT variations on a wide-area cellular wireless network (Verizon Wireless's 3G CDMA Rev A service) across both idle periods and when data transfers are in progress, showing extremely high RTT values and high variability. The x-axis in both pictures is the RTT in milliseconds. The picture on the left shows the histogram (each bin plots the total probability of the RTT value falling within that bin), while the picture on the right is the cumulative distribution function (CDF). These delays suggest a poor network design with excessively long queues that do nothing more than cause delays to be very large. Of course, it means that the timeout method must adapt to these variations to the extent possible. (Data collected in November 2009 in Cambridge, MA and Belmont, MA.)

be conservative and use $k = 4$, in an attempt to make the likelihood of a spurious retransmission very small, because it turns out that the cost of doing one on an already congested network is rather large.

Notice that this approach is similar to something we did earlier in the course when we estimated the bit-error rate from the probability density function of voltage samples, where values above (or below) a threshold would correspond to a bit error. In our case, the "error" is a spurious retransmission.

So far, we have discussed how to set the timeout in a post-facto way, assuming we knew what the RTT samples were. We now need to talk about two important issues to complete the story:

1. How can the sender obtain RTT estimates?

2. How should the sender estimate the mean and deviation and pick a suitable timeout?

Obtaining RTT estimates. If the sender keeps track of when it sent each data packet, then it can obtain a sample of the RTT when it gets an ACK for the packet. The RTT sample is simply the difference in time between when the ACK arrived and when the data packet was sent. An elegant way to keep track of this information in a protocol is for the sender to include the current time in the header of each data packet that it sends in a "timestamp" field. The receiver then simply echoes this time in its ACK. When the sender gets an ACK, it just has to consult the clock for the current time, and subtract the echoed timestamp to obtain an RTT sample.

Calculating the timeout. As explained above, our plan is to pick a timeout that uses both the average and deviation of the RTT sample distribution. The sender must take two factors into account while estimating these values:

1. It must not get swayed by infrequent samples that are either too large or too small. That is, it must employ some sort of "smoothing".

2. It must weigh more recent estimates higher than old ones, because network conditions could have changed over multiple RTTs.

Thus, what we want is a way to track changing conditions, while at the same time not being swayed by sudden changes that don't persist.

Let's look at the first requirement. Given a sequence of RTT samples, $r_0, r_1, r_2, \ldots, r_n$, we want a sequence of smoothed outputs, $s_0, s_1, s_2, \ldots, s_n$ that avoids being swayed by sudden changes that don't persist. This problem sounds like a *filtering problem*, which we have studied earlier. The difference, of course, is that we aren't applying it to frequency division multiplexing, but the underlying problem is what a *low-pass filter* (LPF) does.

A simple LPF that provides what we need has the following form:

$$s_n = \alpha r_n + (1 - \alpha) s_{n-1}, \tag{19.1}$$

where $0 < \alpha < 1$.

To see why Eq. (19.1) is a low-pass filter, let's write down the frequency response, $H(\Omega)$. We know that if $r_n = e^{j\Omega n}$, then $s_n = H(\Omega)e^{j\Omega n}$. Letting $z = e^{j\Omega}$, we can rewrite Eq. (19.1) as

$$H(\Omega)z^n = \alpha z^n + (1 - \alpha)H(\Omega)z^{(n-1)},$$

which then gives us

$$H(\Omega) = \frac{\alpha z}{z - (1 - \alpha)}, \tag{19.2}$$

This filter has a single real pole, and is stable when $0 < \alpha < 1$. The peak of the frequency response is at $\Omega = 0$.

What does α do? Clearly, large values of α mean that we are weighing the current sample much more than the existing s estimate, so there's little memory in the system, and we're therefore letting higher frequencies through more than a smaller value of α. What α does is determine the rate at which the frequency response of the LPF tapers: small α makes lets fewer high-frequency components through, but at the same time, it takes more time to react to persistent changes in the RTT of the network. As α increases, we let more higher frequencies through. Figure 19-4 illustrates this point.

Figure 19-5 shows how different values of α react to a sudden non-persistent change in the RTT, while Figure 19-6 shows how they react to a sudden, but persistent, change in the RTT. Empirically, on networks prone to RTT variations due to congestion, researchers have found that α between 0.1 and 0.25 works well. In practice, TCP uses $\alpha = 1/8$.

The specific form of Equation 19.1 is very popular in many networks and computer systems, and has a special name: **exponential weighted moving average (EWMA)**. It is a "moving average" because the LPF produces a smoothed estimate of the average behavior. It is "exponentially weighted" because the weight given to older samples decays

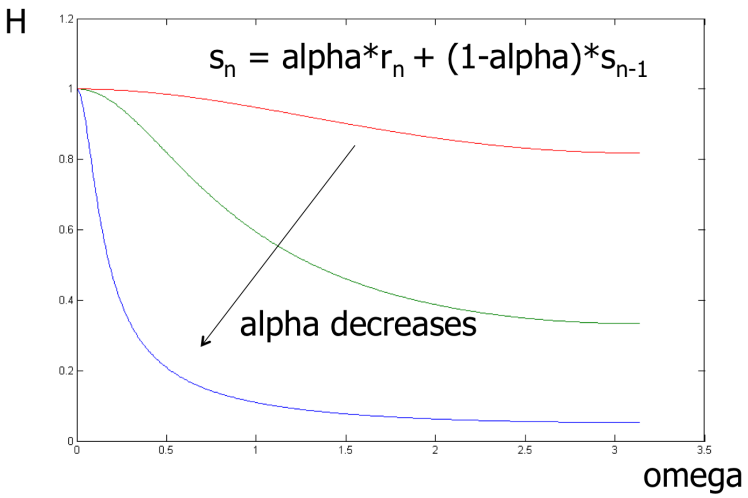


Figure 19-4: Frequency response of the exponential weighted moving average low-pass filter. As α decreases, the low-pass filter becomes even more pronounced. The graph shows the response for $\alpha = 0.9, 0.5, 0.1$, going from top to bottom.

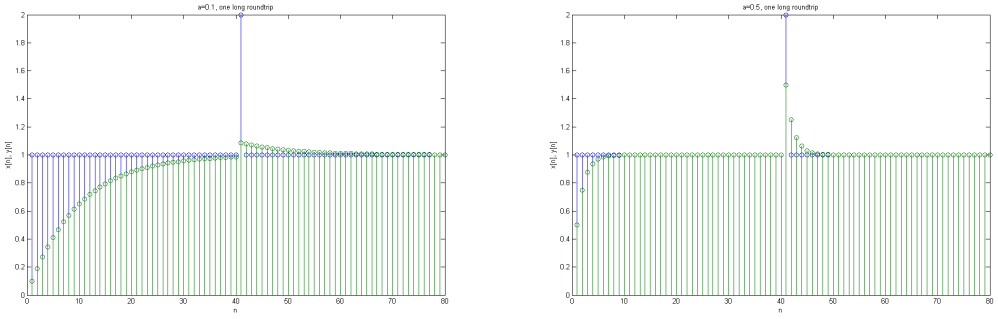


Figure 19-5: Reaction of the exponential weighted moving average filter to a non-persistent spike in the RTT (the spike is double the other samples). The smaller α (0.1, shown on the left) doesn't get swayed by it, whereas the bigger value (0.5, right) does. The output of the filter is shown in green, the input in blue.

geometrically: one can rewrite Eq. 19.1 as

$$s_n = \alpha r_n + \alpha(1-\alpha)r_{n-1} + \alpha(1-\alpha)^2 r_{n-2} + \ldots + \alpha(1-\alpha)^{n-1} r_1 + (1-\alpha)^n r_0, \qquad (19.3)$$

observing that each successive older sample's weight is a factor of $(1-\alpha)$ "less important" than the previous one's.

With this approach, one can compute the smoothed RTT estimate, `srtt`, quite easily using the pseudocode shown below, which runs each time an ACK arrives with an RTT estimate, r.

$$\texttt{srtt} \leftarrow \alpha r + (1-\alpha)\texttt{srtt}$$

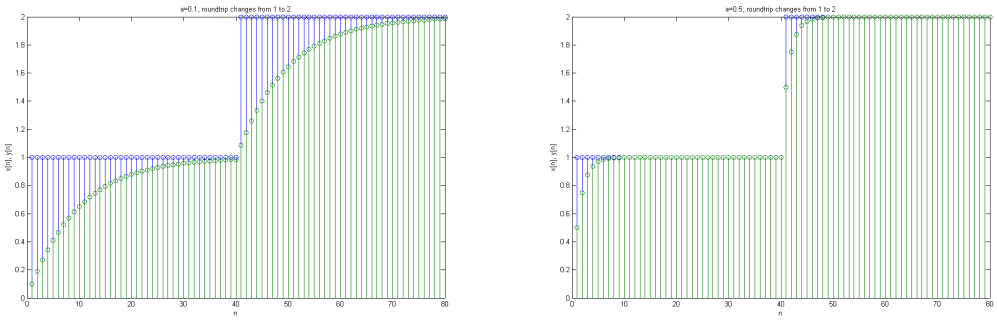


Figure 19-6: Reaction of the exponential weighted moving average filter to a persistent change (doubling) in the RTT. The smaller α (0.1, shown on the left) takes much longer to track the change, whereas the bigger value (0.5, right) responds much quicker. The output of the filter is shown in green, the input in blue.

What about the deviation? Ideally, we want the sample standard deviation, but it turns out to be a bit easier to compute the mean *linear deviation instead*.[4] The following elegant method performs this task:

$$\texttt{dev} \leftarrow |r - srtt|$$
$$\texttt{rttdev} \leftarrow \beta \cdot \texttt{dev} + (1 - \beta) \cdot \texttt{rttdev}$$

Here, $0 < \beta < 1$, and we apply an EWMA to estimate the linear deviation as well. TCP uses $\beta = 0.25$; again, values between 0.1 and 0.25 have been found to work well.

Finally, the timeout is calculated very easily as follows:

$$\texttt{timeout} \leftarrow \texttt{srtt} + 4 \cdot \texttt{rttdev}$$

This procedure to calculate the timeout runs every time an ACK arrives. It does a great deal of useful work essential to the correct functioning of any reliable transport protocol, and it can be implemented in less than 10 lines of code in most programming languages! The reader should note that this procedure does not depend on whether the transport protocol is stop-and-wait or sliding window; the same method works for both.

Exponential back-off of the timeout. When a timeout occurs and the sender retransmits a data packet, it might be lost again (or its ACK might be lost). In that case, it is possible (in networks where congestion is the main reason for packet loss) that the network is heavily congested. Rather than using the same timeout value and retransmitting, it would be prudent to take a leaf from the exponential back-off idea we studied earlier with contention MAC protocols and double the timeout value. Eventually, when the retransmitted data packet is acknowledged, the sender can revert to the timeout value calculated from the mean RTT and its linear deviation. Most reliable transport protocols use an adaptive timer with such an exponential back-off mechanism.

[4]The mean linear deviation is always at least as big as the sample standard deviation, so picking a timeout equal to the mean plus k times the linear deviation has a tail probability no larger than picking a timeout equal to the mean plus k times the sample standard deviation.

19.4 Throughput of Stop-and-Wait

We now show how to calculate the throughput of the stop-and-wait protocol. Clearly, the maximum throughput occurs when there are no packet losses. The sender sends one packet every RTT, so the maximum throughput is exactly that.

We can also calculate the throughput of stop-and-wait when the network has a packet loss rate of ℓ. For convenience, we will treat ℓ as the *bi-directional* loss rate; i.e., the probability of any given packet *or* its ACK getting lost is ℓ.[5] We will assume that the packet loss distribution is independent and identically distributed. What is the throughput of the stop-and-wait protocol in this case?

The answer clearly depends on the timeout that's used. Let's assume that the retransmission timeout is RTO, which we will assume to be a constant for simplicity (i.e., it is the same throughout the connection and the sender doesn't use any exponential back-off). These assumptions mean that the calculation below may be viewed as a (good) upper bound on the throughput.

Let T denote the expected time taken to send a data packet and get an ACK for it. Observe that with probability $1-\ell$, the data packet reaches the receiver and its ACK reaches the sender. On the other hand, with probability ℓ, the sender needs to time out and retransmit a data packet. We can use this property to write an expression for T:

$$T = (1-\ell)\cdot \text{RTT} + \ell(\text{RTO} + T), \tag{19.4}$$

because once the sender times out, the expected time to send a data packet and get an ACK is exactly T, the number we want to calculate. Solving Equation (19.4), we find that $T = \text{RTT} + \frac{\ell}{1-\ell}\cdot$ RTO.

The expected throughput of the protocol is then equal to $1/T$ packets per second.[6]

The good thing about the stop-and-wait protocol is that it is very simple, and should be used under two circumstances: first, when throughput isn't a concern and one wants good reliability, and second, when the network path has a small RTT such that sending one data packet every RTT is enough to saturate the bandwidth of the link or path between sender and receiver.

On the other hand, a typical Internet path between Boston and San Francisco might have an RTT of about 100 milliseconds. If the network path has a bit rate of 1 megabit/s, and we use a data packet size of 10,000 bits, then the maximum throughput of stop-and-wait would be only 10% of the possible rate. And in the face of packet loss, it would be much lower than that.

The next section describes a protocol that provides considerably higher throughput. It

[5] In general, we will treat the loss rate as a probability of loss, so it is a unit-less quantity between 0 and 1; it is not a "rate" like the throughput. A better term might be the "loss probability" or a "loss ratio" but "loss rate" has become standard terminology in networking.

[6] The careful reader or purist may note that we have only calculated T, the *expected time* between the transmission of a data packet and the receipt of an ACK for it. We have then assumed that the expected value of the reciprocal of X, which is a random variable whose expected value is T, is equal to $1/T$. In general, however, $1/E[X]$ is not equal to $E[1/X]$. But the formula for the expected throughput we have written does in fact hold. Intuitively, to see why, define $Y_n = X_1 + X_2 + \ldots X_n$. As $n \to \infty$, one can show using the Chebyshev inequality that the probability that $|Y_n - nT| > \delta n$ goes to 0 or any positive δ. That is, when viewed over a long period of time, the random variable X looks like a constant—which is the only distribution for which the expected value of the reciprocal is equal to the reciprocal of the expectation.

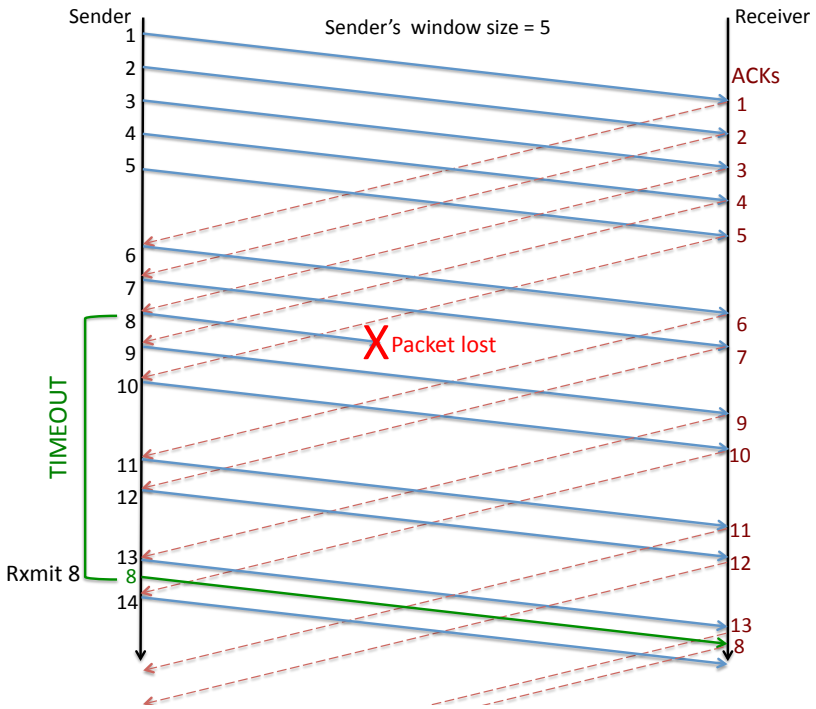


Figure 19-7: The sliding window protocol in action ($W = 5$ here).

builds on all the mechanisms used in the stop-and-wait protocol.

■ 19.5 Sliding Window Protocol

The idea is to use a *window* of data packets that are *outstanding* along the path between sender and receiver. By "outstanding", we mean "unacknowledged". The idea then is to overlap data packet transmissions with ACK receptions. For our purposes, a window size of W data packets means that the sender has at most W outstanding data packets at any time. Our protocol will allow the sender to pick W, and the sender will try to have W outstanding data packets in the network at all times. The receiver is almost exactly the same as in the stop-and-wait case, except that it must also buffer data packets that might arrive out-of-order so that it can deliver them in order to the receiving application. This enhancement makes the receiver more complicated than before, but this complexity is worth the improvement in throughput in most situations.

The key idea in the protocol is that the window *slides* every time the sender gets an ACK. The reason is that the receipt of an ACK is a positive signal that one data packet left the network, and so the sender can add another to replenish the window. This plan is shown in Figure 19-7 that shows a sender (top line) with $W = 5$ and the receiver (bottom line) sending ACKs (dotted arrows) whenever it gets a data packet (solid arrow). Time moves from left to right here.

There are at least two different ways of defining a window in a reliable transport protocol. Here, we will use the following:

A window size of W means that the maximum number of outstanding (unacknowledged) data packets between sender and receiver is W.

When there are no packet losses, the operation of the sliding window protocol is fairly straightforward. The sender transmits the next in-sequence data packet every time an ACK arrives; if the ACK is for data packet k and the window is W, the data packet sent out has sequence number $k + W$. The receiver ACKs each data packet echoing the sender's timestamp and delivers packets in sequence number order to the receiving application. The sender uses the ACKs to estimate the smoothed RTT and linear deviations and sets a timeout. Of course, the timeout will only be used if an ACK doesn't arrive for a data packet within that duration.

We now consider what happens when a packet is lost. Suppose the receiver has received data packets 0 through $k - 1$ and the sender doesn't get an ACK for data packet k. If the subsequent data packets in the window reach the receiver, then each of those packets triggers an ACK. So the sender will have the following ACKs assuming no further packets are lost: $k + 1, k + 2, \ldots, k + W - 1$. Moreover, upon the receipt of each of these ACKs, an additional new data packet will get sent with an even higher sequence number. But somewhere in the midst of these new data packet transmissions, the sender's timeout for data packet k will occur, and the sender will retransmit that packet. If that data packet reaches, then it will trigger an ACK, and if that ACK reaches the sender, yet another new data packet with a new sequence number one larger than the last sent so far will be sent.

Hence, this protocol tries hard to keep as many data packets outstanding as possible, *but not exceeding the window size*, W. If ℓ data packets or ACKs get lost, then the effective number of outstanding data packets reduces to $W - \ell$, until one of them times out, is retransmitted and received successfully by the receiver, and its ACK received successfully at the sender.

We will use a *fixed size* window in our discussion in this chapter. The sender picks a maximum window size and does not change that during a stream. In practice, most practical transport protocols on the Internet should implement a *congestion control* strategy to adjust the window size to prevailing network conditions (level of congestion, rate of data delivery, packet loss rates, round-trip times, etc.)

■ 19.5.1 Sliding Window Sender

We now describe the salient features of the sender side of this fixed-size sliding window protocol. The sender maintains `unacked_pkts`, a buffer of unacknowledged data packets. Every time the sender is called (by a fine-grained timer, which we assume fires each slot), it first checks to see whether any data packets were sent greater than "timeout" seconds ago (assuming time is maintained in seconds). If so, the sender retransmits each of these data packets, and takes care to change the packet transmission time of each of these packets to be the current time. For convenience, we usually maintain the time at which each packet was last sent in the packet data structure, though other ways of keeping track of this information are also possible.

After checking for retransmissions, the sender proceeds to see whether any new data packets can be sent. To properly check if any new packets can be sent, the sender maintains a variable, `outstanding`, which keeps track of the current number of outstanding data packets. If this value is smaller than the maximum window size, the sender sends a new

data packet, setting the sequence number to be `max_seq` + 1, where `max_seq` is the highest sequence number sent so far. Of course, we should remember to update `max_seq` as well, and increment `outstanding` by 1.

Whenever the sender gets an ACK, it should remove the acknowledged data packet from `unacked_pkts` (assuming it hasn't already been removed), decrement outstanding, and call the procedure to calculate the timeout (which will use the timestamp echoed in the current ACK to update the EWMA filters and update the timeout value).

We would like `outstanding` to keep track of the number of unackowledged data packets between sender and receiver. We have described the method to do this task as follows: increment it by 1 on each new data packet transmission, and decrement it by 1 on each ACK that was not previously seen by the sender, corresponding to a packet the sender had previously sent that is being acknowledged (as far as the sender is concerned) for the first time. The question now is whether `outstanding` should be adjusted when a *retransmission* is done. A little thought will show that it should not be. The reason is that it is precisely on a timeout of a data packet that the sender believes that the packet was actually lost, and in the sender's view, the packet has left the network. But the retransmission immediately adds a data packet to the network, so the effect is that the number of outstanding packets is exactly the same. Hence, no change is required in the code.

Implementing a sliding window protocol is sometimes error-prone even when one completely understands the protocol in one's mind. Three kinds of errors are common. First, the timeouts are set too low because of an error in the EWMA estimators, and data packets end up being retransmitted too early, leading to spurious retransmissions. In addition to keeping track of the sender's smoothed round-trip time (srtt), RTT deviation, and timeout estimates,[7] it is a good idea to maintain a counter for the number of retransmissions done for each data packet. If the network has a certain total loss rate between sender and receiver and back (i.e., the bi-directional loss rate), p_l, the number of retransmissions should be on the order of $\frac{1}{1-p_l} - 1$, assuming that each packet is lost independently and with the same probability. (It is a useful exercise to work out why this formula holds.) If your implementation shows a much larger number than this prediction, it is very likely that there's a bug in it.

Second, the number of outstanding data packets might be larger than the configured window, which is an error. If that occurs, and especially if a bug causes the number of outstanding packets to grow unbounded, delays will increase and it is also possible that packet loss rates caused by congestion will increase. It is useful to place an assertion or two that checks that the outstanding number of data packets does not exceed the configured window.

Third, when retransmitting a data packet, the sender must take care to modify the time at which the packet is sent. Otherwise, that packet will end up getting retransmitted repeatedly, a pretty serious bug that will cause the throughput to diminish.

■ 19.5.2 Sliding Window Receiver

At the receiver, the biggest change to the stop-and-wait case is to maintain a list of received data packets that are out-of-order. Call this list `rcvbuf`. Each data packet that arrives is added to this list, assuming it is not already on the list. It's convenient to store this list

[7] In our lab, this information will be printed when you click on the sender node.

in increasing sequence order. Then, check to see whether one or more contiguous data packets starting from `rcv_seqnum` + 1 are in `rcvbuf`. If they are, deliver them to the application, remove them from `rcvbuf`, and remember to update `rcv_seqnum`.

■ 19.5.3 Throughput

What is the throughput of the sliding window protocol we just developed? Clearly, we send W data packets per RTT when there are no data packet or ACK losses, so the throughput in the absence of losses is W/RTT packets per second. So the question one should ask is, what should we set W to in order to maximize throughput, at least when there are no data packet or ACK losses? After answering this question, we will provide a simple formula for the throughput of the protocol in the absence of losses, and then finally consider packet losses.

Setting W

One can address the question of how to choose W using Little's law. Think of the entire bi-directional path between the sender and receiver as a single queue (in reality it's more complicated than a single queue, but the abstraction of a single queue still holds). W is the number of (unacknowledged) packets in the system and RTT is the mean delay between the transmission of a data packet and the receipt of its ACK at the sender (upon which the sender transmits a new data packet). We would like to maximize the processing rate of this system. Note that this rate cannot exceed the bit rate of the slowest, or *bottleneck*, link between the sender and receiver (i.e., the rate of the *bottleneck link*) . If that rate is B packets per second, then by Little's law, setting $W = B \times$ RTT will ensure that the protocol comes close to achieving a thoroughput equal to the available bit rate.

But what should the RTT be in the above formula? After all, the definition of a "RTT sample" is the time that elapses between the transmission of a data packet and the receipt of an ACK for it. As such, it depends on other data using the path. Moreover, if one looks at the formula $B = W/$ RTT, it suggests that one can simply increase the window size W to any value and B may correspondingly just increase. Clearly, that can't be right!

Consider the simple case when there is only one connection active over a network path. Observe that the RTT experienced by a packet P sent on the connection may be broken into two parts: one part that does not depend on any queueing delay (i.e., the sum of the propagation, transmission, and processing delays of the packet and its ACK), and one part that depends on how many other packets were ahead of P in the bottleneck queue. (Here we are assuming that ACKs experience no queueing, for simplicity.) Denote the RTT in the absence of queuing as $\text{RTT}_{\min}$, the minimum possible round-trip time that the connection can experience.

Now, suppose the RTT of the connection is equal to $\text{RTT}_{\min}$. That is, there is no queue building up at the bottleneck link. Then, the throughput of the connection is $W/\text{RTT} = W/\text{RTT}_{\min}$. We would like this throughput to be the bottleneck link rate, B. Setting $W/\text{RTT}_{\min} = B$, we find that W should be equal to $B \cdot \text{RTT}_{\min}$.

This quantity—$B \cdot \text{RTT}_{\min}$—is an important concept for sliding window protocols (all sliding window protocols, not just the one we have studied). It is called the **bandwidth-delay product** of the connection and is a property of the bi-directional network path between sender and receiver. When the window size is strictly smaller than the bandwidth-

delay product, the throughput will be strictly smaller than the bottleneck rate, B, and the queueing delay will be non-existent. In this phase, the connection's throughput *linearly increases* as we increase the window size, W, assuming no other traffic intervenes. The smallest window size for which the throughput will be equal to B is the bandwidth-delay product.

This discussion shows that for our sliding window protocol, setting $W = B \times \text{RTT}_{\min}$ achieves the maximum possible throughput, B, *in the absence of any data packet or ACK losses*. When packet losses occur, the window size will need to be higher to get maximum throughput (utilization), because we need a sufficient number of unacknowledged data packets to keep a $B \times \text{RTT}_{\min}$ worth of packets even when losses occur. A smaller window size will achieve sub-optimal throughput, linear in the window size, and inversely proportional to $\text{RTT}_{\min}$.

But once W exceeds $B \times \text{RTT}_{\min}$, the RTT experienced by the connection includes queueing as well, and the RTT will *no longer be a constant independent of* W! That is, increasing W will cause RTT to also increase, but the rate, B, will no longer increase. What is the throughput in this case?

We can answer this question by applying Little's law *twice*. Once at the bottleneck link's queue, and once on the entire network path. We will show the intuitive result that if $W > B \times \text{RTT}_{\min}$, then the throughput is B packets per second.

First, let the average number of packets at the queue of the bottleneck link be Q. By Little's law applied to this queue, we know that $Q = B \cdot \tau$, where B is the rate at which the queue drains (i.e., the bottleneck link rate), and τ is the average delay in the queue, so $\tau = Q/B$.

We also know that

$$\text{RTT} = \text{RTT}_{\min} + \tau = \text{RTT}_{\min} + Q/B. \tag{19.5}$$

Now, consider the window size, W, which is the number of unacknowledged packets. We know that all these packets, by conservation of packets, must either be in the bottleneck queue, or in the non-queueing part of the system. That is,

$$W = Q + B \cdot \text{RTT}_{\min}. \tag{19.6}$$

Finally, from Little's law applied to the entire bi-directional network path,

$$\begin{aligned} \text{Throughput} &= \frac{W}{\text{RTT}} && (19.7)\\ &= \frac{B \cdot \text{RTT}_{\min} + Q}{\text{RTT}_{\min} + (Q/B)} && (19.8)\\ &= B && (19.9) \end{aligned}$$

Thus, we can conclude that, in the absence of any data packet or ACK losses, the connection's throughput is as shown schematically in Figure 19-8.

Throughput of the sliding window protocol with packet losses

Assuming that one sets the window size properly, i.e., to be large enough so that $W \geq B \times \text{RTT}_{\min}$ always, even in the presence of data or ACK losses, what is the maximum

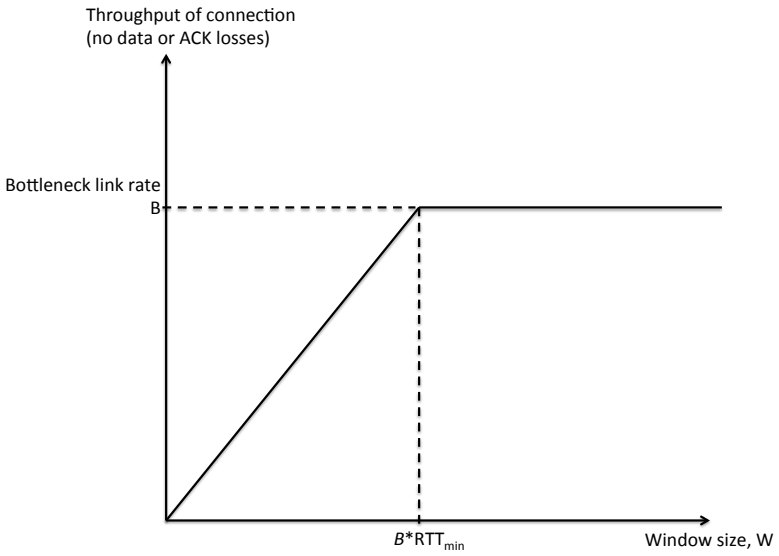


Figure 19-8: Throughput of the sliding window protocol as a function of the window size in a network with no other traffic. The bottleneck link rate is B packets per second and the RTT without any queueing is $\text{RTT}_{\min}$. The product of these two quantities is the bandwidth-delay product.

throughput of our sliding window protocol if the network has a certain probability of packet loss?

Consider a simple model in which the network path loses any packet—data or ACK—such that the probability of either a data packet being lost *or* its ACK being lost is equal to ℓ, and the packet loss random process is independent and identically distributed (the same model as in our analysis of stop-and-wait). Then, the utilization achieved by our sliding window reliable transport protocol is at most $1 - \ell$. Moreover, for a large-enough window size, W, our sliding window protocol comes close to achieving it.

The reason for the upper bound on utilization is that in this protocol, a data packet is acknowledged only when the sender gets an ACK explicitly for that packet. Now consider the number of transmissions that any given data packet must incur before its ACK is received by the sender. With probability $1 - \ell$, we need one transmission, with probability $\ell(1 - \ell)$, we need two transmissions, and so on, giving us an *expected number of transmissions* of $\frac{1}{1-\ell}$. If we make this number of transmissions, one data packet is successfully sent and acknowledged. Hence, the utilization of the protocol can be at most $\frac{1}{\frac{1}{1-\ell}} = 1 - \ell$. In fact, it turns out the $1 - \ell$ is the *capacity* (i.e., upper-bound on throughput) for *any* channel (network path) with packet loss rate ℓ.

If the sender picks a window size sufficiently larger than the bandwidth-minimum-RTT product, so that at least bandwidth-minimum-RTT packets are in transit (unacknowledged) even in the face of data and ACK losses, then the protocol's utilization will be close to the maximum value of $1 - \ell$.

Is a good timeout important for the sliding window protocol?

Given that our sliding window protocol always sends a data packet every time the sender gets an ACK, one might reasonably ask whether setting a good timeout value, which under even the best of conditions involves a hard trade-off, is essential. The answer turns out to be subtle: it's true that the timeout can be quite large, because data packets will continue to flow as long as some ACKs are arriving. However, as data packets (or ACKs) get lost, the effective window size keeps falling, and eventually the protocol will stall until the sender retransmits. So one can't ignore the task of picking a timeout altogether, but one can pick a more conservative (longer) timeout than in the stop-and-wait protocol. However, the longer the timeout, the bigger the stalls experienced by the receiver application—even though the receiver's transport protocol would have received the data packets, they can't be delivered to the application because it wants the data to be delivered *in order*. Therefore, a good timeout is still quite useful, and the principles discussed in setting it are widely useful.

Secondly, we note that the longer the timeout, the bigger the receiver's buffer has to be when there are losses; in fact, in the worst case, there is no bound on how big the receiver's buffer can get. To see why, think about what happens if we were unlucky and a data packet with a particular sequence number kept getting lost, but everything else got through.

The two factors mentioned above affect the throughput of the transport protocol, but the biggest consequence of a long timeout is the effect on the *latency* perceived by applications (and users). The reason is that data packets are delivered in-order by the protocol to the application, which means that a missing packet with sequence number k will cause the application to stall, even though data packets with sequence numbers larger than k have arrived and are in the transport protocol's receiver buffer. Hence, an excessively long timeout hurts interactivity and degrades the user's experience.

■ 19.6 Summary

This chapter described the key concepts in the design on a reliable data transport protocol. The big idea is to use redundancy in the form of careful retransmissions, for which we developed the idea of using sequence numbers to uniquely identify data packets and acknowledgments for the receiver to signal the successful reception of a data packet to the sender. We discussed how the sender can set a good timeout, balancing between the ability to track a persistent change of the round-trip times against the ability to ignore non-persistent glitches. The method to calculate the timeout involved estimating a smoothed mean and linear deviation using an exponential weighted moving average, which is a single real-zero low-pass filter. The timeout itself is set at the mean + 4 times the deviation to ensure that the tail probability of a spurious retransmission is small. We used these ideas in developing the simple stop-and-wait protocol.

We then developed the idea of a sliding window to improve performance, and showed how to modify the sender and receiver to use this concept. Both the sender and receiver are now more complicated than in the stop-and-wait protocol, but when there are no losses, one can set the window size to the bandwidth-delay product and achieve high throughput in this protocol. We also studied how increasing the window size increases the throughput linearly up to a point, after only the (queueing) delay increases, and not the throughput of

the connection.

■ Problems and Questions

1. Consider a best-effort network with variable delays and losses. In such a network, Louis Reasoner suggests that the receiver does not need to send the sequence number in the ACK in a correctly implemented stop-and-wait protocol, where the sender sends data packet $k+1$ *only after* the ACK for data packet k is received. Explain whether he is correct or not.

2. The 802.11 (WiFi) link-layer uses a stop-and-wait protocol to improve link reliability. The protocol works as follows:

   (a) The sender transmits data packet $k+1$ to the receiver as soon as it receives an ACK for the data packet k.
   (b) After the receiver gets the entire data packet, it computes a checksum (CRC). The processing time to compute the CRC is T_p and you may assume that it does not depend on the packet size.
   (c) If the CRC is correct, the receiver sends a link-layer ACK to the sender. The ACK has negligible size and reaches the sender instantaneously.

   The sender and receiver are near each other, so you can ignore the propagation delay. The bit rate is $R = 54$ Megabits/s, the smallest data packet size is 540 bits, and the largest data packet size is 5,400 bits.

   What is the maximum processing time T_p that ensures that the protocol will achieve a throughput of *at least 50%* of the bit rate of the link in the absence of data packet and ACK losses, *for any data packet size*?

3. Alyssa P. Hacker sets up a wireless network in her home to enable her computer ("client") to communicate with an Access Point (AP). The client and AP communicate with each other using a stop-and-wait protocol.

   The data packet size is 10000 bits. The total round-trip time (RTT) between the AP and client is equal to 0.2 milliseconds (that includes the time to process the packet, transmit an ACK, and process the ACK at the sender) **plus** the transmission time of the 10000 bit packet over the link.

   Alyssa can configure two possible transmission bit rates for her link, with the following properties:

Bit rate	**Bi-directional packet loss probability**	**RTT**
10 Megabits/s	$1/11$	_____
20 Megabits/s	$1/4$	_____

   Alyssa's goal is to select the bit rate that provides the higher throughput for a stream of packets that need to be delivered reliably between the AP and client using stop-and-wait. For both bit rates, the **retransmission timeout (RTO) is 2.4 milliseconds**.

(a) Calculate the round-trip time (RTT) for each bit rate?

(b) For each bit rate, calculate the **expected time**, in milliseconds, to successfully deliver a packet and get an ACK for it. **Show your work.**

(c) Using the above calculations, which bit rate would you choose to achieve Alyssa's goal?

4. Suppose the sender in a reliable transport protocol uses an EWMA filter to estimate the smoothed round trip time, srtt, every time it gets an ACK with an RTT sample r.

   srtt $\rightarrow \alpha \cdot r + (1-\alpha) \cdot$ srtt

   We would like every data packet in a window to contribute a weight of at least 1% to the srtt calculation. As the window size increases, should α increase, decrease, or remain the same, to achieve this goal? (You should be able to answer this question without writing any equations.)

5. TCP computes an average round-trip time (RTT) for the connection using an EWMA estimator, as in the previous problem. Suppose that at time 0, the initial estimate, srtt, is equal to the true value, r_0. Suppose that immediately after this time, the RTT for the connection increases to a value R and remains at that value for the remainder of the connection. You may assume that $R >> r_0$.

   Suppose that the TCP retransmission timeout value at step n, $RTO(n)$, is set to $\beta \cdot$ srtt. Calculate the number of RTT samples before we can be sure that there will be no spurious retransmissions. Old TCP implementations used to have $\beta = 2$ and $\alpha = 1/8$. How many samples does this correspond to before spurious retransmissions are avoided, for this problem? (As explained in Section 19.3, TCP now uses the mean linear deviation as its RTO formula. Originally, TCP didn't incorporate the linear deviation in its RTO formula.)

6. Consider a sliding window protocol between a sender and a receiver. The receiver should deliver data packets reliably and in order to its application.

   The sender correctly maintains the following state variables:
   `unacked_pkts` – the buffer of unacknowledged data packets
   `first_unacked` – the lowest unacked sequence number (undefined if all data packets have been acked)
   `last_unacked` – the highest unacked sequence number (undefined if all data packets have been acked)
   `last_sent` – the highest sequence number sent so far (whether acknowledged or not)

   If the receiver gets a data packet that is strictly larger than the next one in sequence, it adds the packet to a buffer if not already present. We want to ensure that the size of this buffer of data packets awaiting delivery *never exceeds* a value $W \geq 0$. Write down the check(s) that the sender should perform before sending a new data packet in terms of the variables mentioned above that ensure the desired property.

7. Alyssa P. Hacker measures that the network path between two computers has a round-trip time (RTT) of 100 milliseconds. The queueing delay is negligible. The

rate of the bottleneck link between them is 1 Mbyte/s. Alyssa implements the reliable sliding window protocol studied in 6.02 and runs it between these two computers. The data packet size is fixed at 1000 bytes (you can ignore the size of the acknowledgments). There is no other traffic.

(a) Alyssa sets the window size to 10 data packets. What is the resulting maximum utilization of the bottleneck link? Explain your answer.

(b) Alyssa's implementation of a sliding window protocol uses an 8-bit field for the sequence number in each data packet. Assuming that the RTT remains the same, what is the smallest value of the bottleneck link bandwidth (in Mbytes/s) that will cause the protocol to stop working correctly when packet losses occur? Assume that the definition of a window in her protocol is the difference between the last transmitted sequence number and the last in-sequence ACK.

(c) Suppose the window size is 10 data packets and that the value of the sender's retransmission timeout is 1 second. A data packet gets lost before it reaches the receiver. The protocol continues *and no other data packets or acks are lost*. The receiver wants to deliver data to the application in order.

What is the maximum size, in packets, that the buffer at the receiver can grow to in the sliding window protocol? Answer this question for the two different definitions of a "window" below.

i. When the window is the maximum difference between the last transmitted data packet and the last in-sequence ACK received at the sender:

ii. When the window is the maximum number of unacknowledged data packets at the sender:

8. In the reliable transport protocols we studied, the receiver sends an acknowledgment (ACK) saying "I got k" whenever it receives a data packet with sequence number k. Ben Bitdiddle invents a different method using **cumulative ACKs**: whenever the receiver gets a data packet, whether in order or not, it sends an ACK saying "I got every data packet up to and including ℓ", where ℓ is the **highest, in-order** data packet received so far.

The definition of the window is the same as before: a window size of W means that the maximum number of unacknowledged data packets is W. Every time the sender gets an ACK, it may transmit one or more data packets, within the constraint of the window size. It also implements a timeout mechanism to retransmit data packets that it believes are lost using the algorithm described in these notes. The protocol runs over a best-effort network, but *no data packet or ACK is duplicated at the network or link layers*.

The sender sends a stream of new data packets according to the sliding window protocol, and in response gets the following cumulative ACKs from the receiver:

1 2 3 4 4 4 4 4 4 4

(a) Now, suppose that the sender times out and retransmits the first unacknowledged data packet. When the receiver gets that retransmitted data packet, what can you say about the ACK, a, that it sends?

i. $a = 5$.
ii. $a \geq 5$.
iii. $5 \leq a \leq 11$.
iv. $a = 11$.
v. $a \leq 11$.

(b) Assuming no ACKs were lost, what is the *minimum* window size that can produce the sequence of ACKs shown above?

(c) Is it possible for the given sequence of cumulative ACKs to have arrived at the sender even when no data packets were lost en route to the receiver when they were sent?

(d) A little bit into the data transfer, the sender observes the following sequence of cumulative ACKs sent from the receiver:

21 22 23 25 28

The window size is 8 packets. What data packet(s) should the sender transmit upon receiving each of the above ACKs, if it wants to maximize the number of unacknowledged data packets?

On getting ACK #	→ Send ??	On getting ACK #	→ Send ??
21	→	22	→
23	→	25	→
28	→		

9. Give one example of a situation where the cumulative ACK protocol described in the previous problem gets higher throughput than the sliding window protocol described in this chapter.

10. A sender S and receiver R communicate reliably over a series of links using a sliding window protocol with some window size, W packets. The path between S and R has one bottleneck link (i.e., one link whose rate bounds the throughput that can be achieved), whose data rate is C packets/second. When the window size is W, the queue at the bottleneck link is always **full**, with Q data packets in it. The round trip time (RTT) of the connection between S and R during this data transfer with window size W is T seconds, *including the queueing delay*. There are no data packet or ACK losses in this case, and there are no other connections sharing this path.

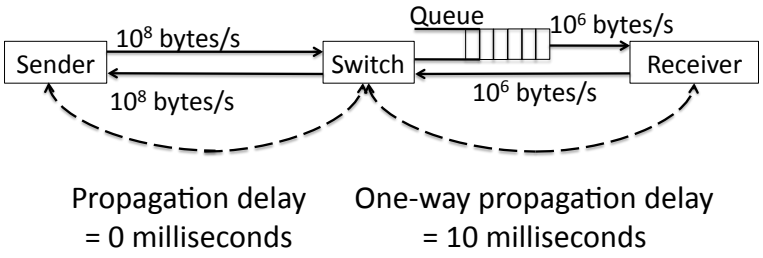
(a) Write an expression for W in terms of the other parameters specified above.

(b) We would like to reduce the window size from W and still achieve high utilization. What is the minimum window size, W_{min}, which will achieve 100% utilization of the bottleneck link? Express your answer as a function of C, T, and Q.

(c) Now suppose the sender starts with a window size set to W_{min}. If all these data packets get acknowledged and no packet losses occur in the window, the sender increases the window size by 1. The sender keeps increasing the window size

in this fashion until it reaches a window size that causes a data packet loss to occur. What is the smallest window size at which the sender observes a data packet loss caused by the bottleneck queue overflowing? Assume that no ACKs are lost.

11. Ben Bitdiddle decides to use the sliding window transport protocol described in these notes on the network shown in Figure 19-9. The receiver sends **end-to-end ACKs** to the sender. The switch in the middle simply forwards packets in best-effort fashion.



Max queue size = 100 packets
Packet size = 1000 bytes
ACK size = 40 bytes
Initial sender window size = 10 packets

Figure 19-9: Ben's network.

(a) The sender's window size is 10 packets. At what approximate rate (in packets per second) will the protocol deliver a multi-gigabyte file from the sender to the receiver? Assume that there is no other traffic in the network and packets can only be lost because the queues overflow.

i. Between 900 and 1000.
ii. Between 450 and 500.
iii. Between 225 and 250.
iv. Depends on the timeout value used.

(b) You would like to double the throughput of this sliding window transport protocol running on the network shown on the previous page. To do so, you can apply **one** of the following techniques alone:

i. Double the window size.
ii. Halve the propagation time of the links.
iii. Double the rate of the link between the Switch and Receiver.

For each of the following sender window sizes, list which of the above techniques, **if any, can approximately double the throughput**. If no technique does the job, say "None". There might be more than one answer for each window size, in which case you should list them all. Each technique works in isolation.

1. $W = 10$: _____
2. $W = 50$: _____
3. $W = 30$: _____

12. Eager B. Eaver starts MyFace, a next-generation social networking web site in which the only pictures allowed are users' faces. MyFace has a simple request-response interface. The client sends a request (for a face), the server sends a response (the face). Both request and response fit in one packet (the faces in the responses are small pictures!). When the client gets a response, it immediately sends the next request. The size of the largest packet is $S = 1000$ bytes.

    Eager's server is in Cambridge. Clients come from all over the world. Eager's measurements show that one can model the typical client as having a 100 millisecond round-trip time (RTT) to the server (i.e., the network component of the request-response delay, not counting the additional processing time taken by the server, is 100 milliseconds).

    If the client does not get a response from the server in a time τ, it resends the request. It keeps doing that until it gets a response.

    (a) Is the protocol described above "at least once", "at most once", or "exactly once"?

    (b) Eager needs to provision the link bandwidth for MyFace. He anticipates that at any given time, the largest number of clients making a request is 2000. What minimum outgoing link bandwidth from MyFace will ensure that the link connecting MyFace to the Internet will not experience congestion?

    (c) Suppose the probability of the client receiving a response from the server for any given request is p. What is the expected time for a client's request to obtain a response from the server? Your answer will depend on p, RTT, and τ.

13. Lem E. Tweetit is designing a new protocol for Tweeter, a Twitter rip-off. All tweets in Tweeter are 1000 bytes in length. Each tweet sent by a client and received by the Tweeter server is immediately acknowledged by the server; if the client does not receive an ACK within a timeout, it re-sends the tweets, and repeats this process until it gets an ACK.

    Sir Tweetsalot uses a device whose data transmission rate is 100 Kbytes/s, which you can assume is the bottleneck rate between his client and the server. The round-trip propagation time between his client and the server is 10 milliseconds. Assume that there is no queueing on any link between client and server and that the processing time along the path is 0. You may also assume that the ACKs are very small in size, so consume neglible bandwidth and transmission time (of course, they still need to propagate from server to client). Do not ignore the transmission time of a tweet.

    (a) What is the smallest value of the timeout, in *milliseconds*, that will avoid spurious retransmissions?

(b) Suppose that the timeout is set to 90 milliseconds. Unfortunately, the probability that a given client transmission gets an ACK is only 75%. What is the *utilization* of the network?

14. A sender A and a receiver B communicate using the stop-and-wait protocol studied in this chapter. There are n links on the path between A and B, each with a data rate of R bits per second. The size of a data packet is S bits and the size of an ACK is K bits. Each link has a physical distance of D meters and the speed of signal propagation over each link is c meters per second. The total processing time experienced by a data packet *and* its ACK is T_p seconds. ACKs traverse the same links as data packets, except in the opposite direction on each link (the propagation time and data rate are the same in both directions of a link). There is no queueing delay in this network. Each link has a packet loss probability of p, with packets being lost independently.

    What are the following four quantities in terms of the parameters given?

    (a) Transmission time for a data packet *on one link* between A and B: _____.

    (b) Propagation time for a data packet across n links between A and B: _____.

    (c) Round-trip time (RTT) between A and B? _____.
    (The RTT is defined as the elapsed time between the start of transmission of a data packet and the completion of receipt of the ACK sent in response to the data packet's reception by the receiver.)

    (d) Probability that a data packet sent by A will reach B: _____.

15. Ben Bitdiddle gets rid of the timestamps from the packet header in this chapter's stop-and-wait transport protocol running over a best-effort network. The network may lose or reorder packets, but it never duplicates a packet. In the protocol, the receiver sends an ACK for each data packet it receives, echoing the sequence number of the packet that was just received.

    The sender uses the following method to estimate the round-trip time (RTT) of the connection:

    1. When the sender transmits a packet with sequence number k, it stores the time on its machine at which the packet was sent, t_k. If the transmission is a retransmission of sequence number k, then t_k is updated.
    2. When the sender gets an ACK for packet k, if it has not already gotten an ACK for k so far, it observes the current time on its machine, a_k, and measures the RTT sample as $a_k - t_k$.

    If the ACK received by the sender at time a_k was sent by the receiver in response to a data packet sent at time t_k, then the RTT sample $a_k - t_k$ is said to be correct. Otherwise, it is incorrect.

    State **True** or **False** for the following statements, with an explanation for your choice.

(a) If the sender never retransmits a data packet during a data transfer, then all the RTT samples produced by Ben's method are correct.

(b) If data and ACK packets are never reordered in the network, then all the RTT samples produced by Ben's method are correct.

(c) If the sender makes no spurious retransmissions during a data transfer (i.e., it only retransmits a data packet if all previous transmissions of data packets with the same sequence number did in fact get dropped before reaching the receiver), then all the RTT samples produced by Ben's method are correct.

16. Opt E. Miser implements this chapter's stop-and-wait reliable transport protocol with one modification: being stingy, he replaces the sequence number field with a 1-bit field, deciding to reuse sequence numbers across data packets. The first data packet has sequence number 1, the second has number 0, the third has number 1, the fourth has number 0, and so on. Whenever the receiver gets a packet with sequence number $s (= 0 \text{ or } 1)$, it sends an ACK to the sender echoing s. The receiver delivers a data packet to the application if, and only if, its sequence number is different from the last one delivered, and upon delivery, updates the last sequence number delivered.

    He runs this protocol over a best-effort network that can lose packets (with probability < 1) or reorder them, and whose delays are variable. Explain whether the modified protocol always provides reliable, in-order delivery of a stream of packets.

17. Consider a reliable transport connection using this chapter's sliding window protocol on a network path whose RTT in the absence of queueing is $\text{RTT}_{\min} = 0.1$ seconds. The connection's bottleneck link has a rate of $C = 100$ packets per second, and the queue in front of the bottleneck link has space for $Q = 20$ packets.

    Assume that the sender uses a sliding window protocol with fixed window size. There is no other traffic on the path.

    (a) If the window size is 8 packets, then what is the throughput of the connection?

    (b) If the window size is 16 packets, then what is the throughput of the connection?

    (c) What is the smallest window size for which the connection's RTT exceeds $\text{RTT}_{\min}$?

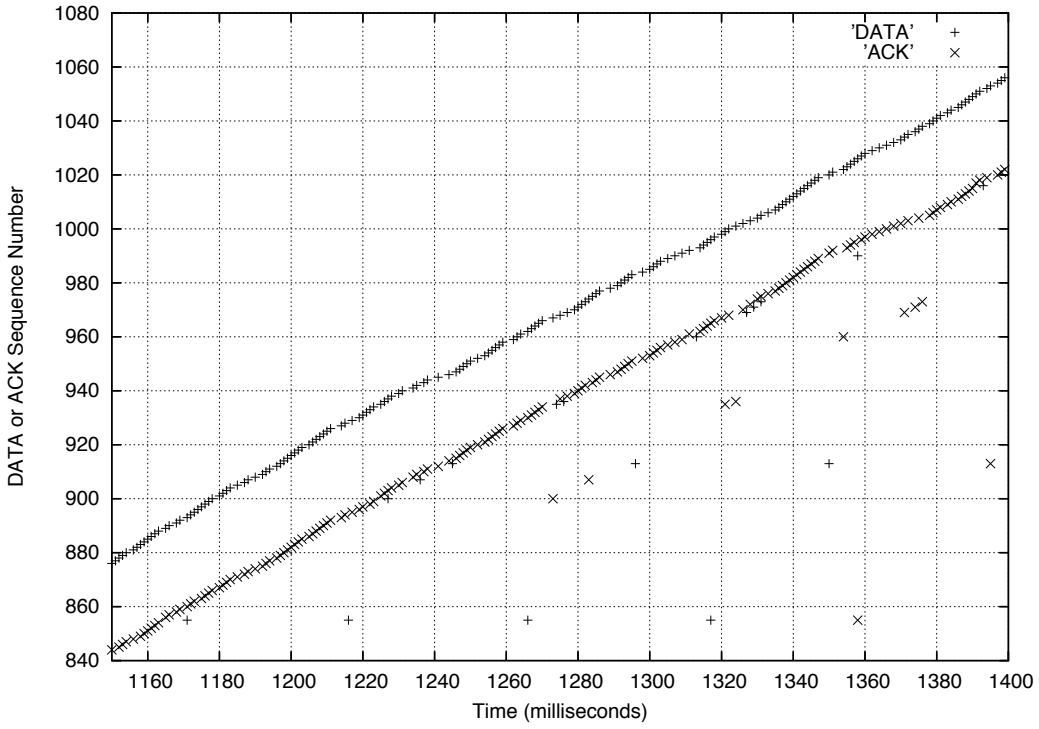
    (d) What is the largest value of the sender window size for which no packets are lost due to a queue overflow?

18. Annette Werker correctly implements the fixed-size sliding window protocol described in this chapter. She instruments the sender to store the time at which each DATA packet is sent and the time at which each ACK is received. A snippet of the DATA and ACK traces from an experiment is shown in the picture below. Each $+$ is a DATA packet transmission, with the x-axis showing the transmission time and the y-axis showing the sequence number. Each $\times$ is an ACK reception, with the x-axis showing the ACK reception time and the y-axis showing the ACK sequence number. All DATA packets have the same size.

    Answer the following questions, providing a brief explanation for each one.

    (a) Estimate any one sample round-trip time (RTT) of the connection.

    (b) Estimate the sender's retransmission timeout (RTO) for this trace.



(c) On the picture, circle DATA packet retransmissions for four different sequence numbers.

(d) Some DATA packets in this trace may have incurred more than one retransmission? On the picture, draw a square around one such retransmission.

(e) What is your best estimate of the sender's window size?

(f) What is your best estimate of the throughput in packets per second of the connection?

(g) Considering only sequence numbers > 880, what is your best estimate of the packet loss rate experienced by DATA packets?

19. Consider the same setup as the previous problem. Suppose the window size for the connection is equal to twice the bandwidth-delay product of the network path.

For each change to the parameters of the network path or the sender given below, explain if the connection's throughput (not utilization) will increase, decrease, or remain the same. In each statement, nothing other than what is specified in that statement changes.

(a) The packet loss rate, ℓ, decreases to $\ell/3$.

(b) The minimum value of the RTT, R, increases to $1.8R$.

(c) The window size, W, decreases to $W/3$.

20. Annette Werker conducts tests between a server and a client using the sliding window protocol described in this chapter. There is no other traffic on the path and no packet loss. Annette finds that:

- With a window size $W_1 = 50$ packets, the throughput is 200 packets per second.
- With a window size $W_2 = 100$ packets, the throughput is 250 packets per second.

Annette finds that even this small amount of information allows her to calculate several things, assuming there is only one bottleneck link. Calculate the following:

(a) The minimum round-trip time between the client and server.

(b) The average queueing delay at the bottleneck when the window size is 100 packets.

(c) The average queue size when the window size is 100 packets.