*MIT 6.02* **DRAFT** *Lecture Notes*
*Fall 2011 (Last update: November 27, 2011)*
*Comments, questions or bug reports?*
*Please contact hari at mit.edu*

# CHAPTER 19
# 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.

## 19.1 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, $\tau$. 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 $\tau$ 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.

### 19.1.1 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. Obviously, packets sent along this path to $D$ will be stuck in the network forever, unless other mechanisms are put in place to "flush" such packets from the network (see Section 19.2).

### 19.1.2 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.

## 19.2 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.

## ■ 19.3 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` - $\delta$, `HELLO_INTERVAL` + $\delta$], 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 $\ell^k$, where $\ell$ 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.

## 19.4 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.

## 19.5 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 $\tau$ 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

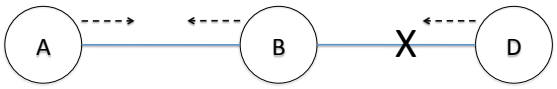



**Figure 19-1: Distance-vector protocol showing the "count-to-infinity" problem (see Section 19.6 for the explanation).**

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.

## ■ 19.6 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 19-1). 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 protocol in elegant ways.

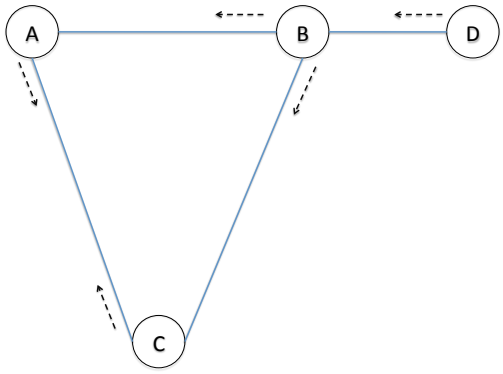



**Figure 19-2: Split horizon (with or without poison reverse) doesn't prevent routing loops of three or more hops. The dashed arrows show the routing advertisements for destination $D$. If link $BD$ fails, as explained in the text, it is possible for a "count-to-infinity" routing loop involving $A$, $B$, and $C$ to ensue.**

## ■ 19.7 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 19-2. 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.

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 $D$

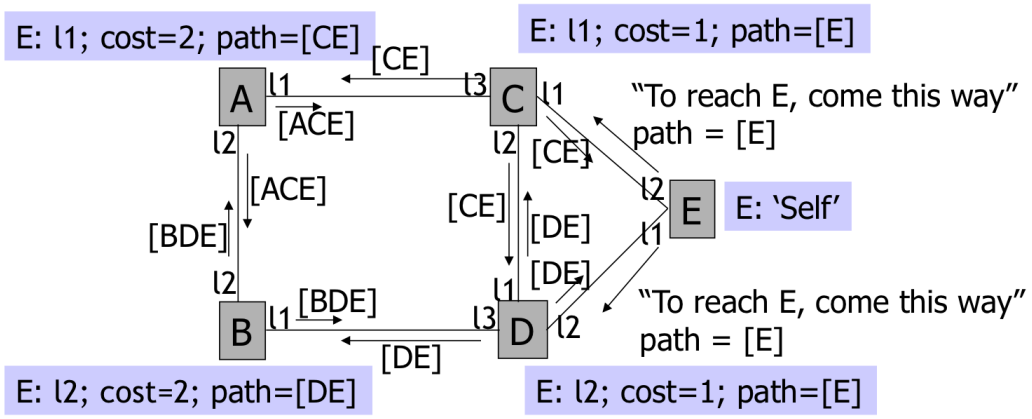



**Figure 19-3: Path vector protocol example.**

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.

## ■ 19.8 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 19-3 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).

## ■ 19.9 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 $\ell$ bytes in size, then because each node broadcasts every other node's LSA once, the number of bytes sent is $\ell n$. 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.

## ■ Acknowledgments

Thanks to Sari Canelake for several useful comments and to Kerry Xing for spotting editing errors.

## ■ 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 19-4, 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 $\delta$ 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

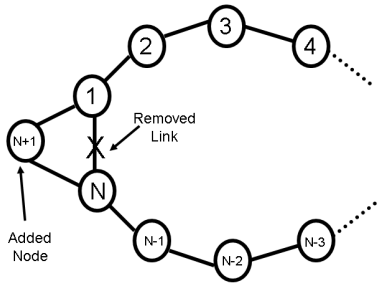



**Figure 19-4: The ring network with $N$ nodes ($N$ is even).**

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.

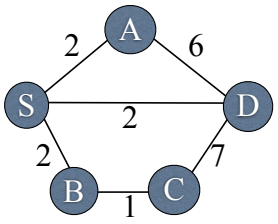
(d) MinCostSquared: Every node picks the path that has the smallest sum of squares of link costs along the path.

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.

Node S:

Node A:

Node B:

Node C:

(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.