MIT 6.02 **DRAFT** *Lecture Notes*
Spring 2009 (Last update: April 22, 2009)
Comments, questions or bug reports?
Please contact Hari Balakrishnan (hari@mit.edu) or 6.02-staff@mit.edu

LECTURE 20
Network Routing - I (Without Any Failures)

This lecture 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.
2. Forwarding.
3. Distributed routing protocols: *distance-vector* and *link-state* protocols. Understanding how routing protocols handle failures.

■ 20.1 The Problem

As explained in earlier lectures, 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. A network topology, such as the one shown at the top of Figure 20-1 is the result of these interconnections. 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.

This problem is challenging for the following reasons:

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

1. **Distributed information:** Each node only knows about its local connectivity—what its immediate neighbors are. 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 of packets. For concreteness, we will assume that links have costs (these costs could model link latency), and that we are interested in finding a path between any source and destination that minimizes the total cost. 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 adapting 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 (see Lecture 17), 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 continuosly, 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*.

■ 20.2 Addressing and Forwarding

Clearly, to send packets to some end point, we need a way to identify it uniquely. Such identifiers are also called *names* 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. Later, when we discuss how to design large networks, we will constrain how addresses

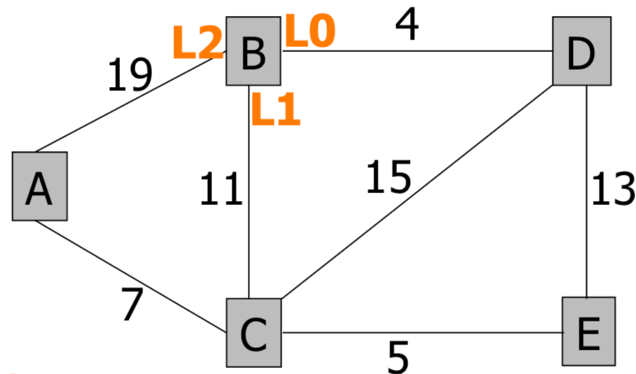

Table @ B

Destination	Link (next-hop)	Cost
A	ROUTE L1	18
B	'Self'	0
C	L1	11
D	L0	4
E	L1	16

Figure 20-1: 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.

are assigned, and also draw the distinction between the unique identifier (name) of a node and its addresses. The distinction will allow us to use an address to refer to the network interface on a node; since a node may have multiple links connected to it, its unique name is distinct from the addresses of its interfaces.

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 called *forwarding*. The selected link is called the *outgoing link*. The table is called the *routing table*. 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 20-1 shows a routing table and routes at a node in a simple network.

One of our goals is to ensure that packets don't remain stuck in the network forever

because of routing glitches. For example, under certain conditions, it is possible for a *routing loop* to occur, in which packets destined for a destination D traverse a sequence of two or more switches $S_1, S_2, \ldots, S_n$, where S_1 and S_n are the same. Even when the routing protocol is implemented correctly, it may take a while for the procedure to untangle this situation and produce a *loop-free path*.

To combat 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 that's (much) 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).

Finally, because data may be corrupted when sent over a link or due to bugs in switch implementations, it is customary to include a checksum that covers the packet's header, and possibly also the data being sent. The forwarding process needs to make sure that the checksum is adjusted to reflect the decrement done to the hop-limit field.

In summary, the basic steps done while forwarding a packet are:

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".

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 and adjust the checksum(s). Enqueue the packet in the queue corresponding to the outgoing link returned by the route lookup procedure. When this packet reaches the front of the queue, the switch will send the packet on the link.

■ 20.3 Routing Overview

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.

2. *Sending advertisements*: Each node sends *routing advertisements* periodically to its neighbors. These advertisements summarize useful information about the network topology.

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, 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, but because the ability to refresh the route means that the system can adapt by "routing around" link and node failures.

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. The first class of protocols are *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 destination. For example, in the simplest form of a vector protocol, n advertises its *cost* to reach each destination. In the integration step, each recipient of the advertisement can use the advertised cost from each neighbor, together with some other information 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*.

The second class of protocols are *link-state protocols*. Here, each node advertises information about the link to its current neighbors on all its links, and each recipient simply resends 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.

The next two sections discuss the essential details of distance-vector and link-state protocols. But before that, we need to talk about how nodes determine the current set of neighbors. This step is generally the same in both classes of protocols, and usually has a name: the *HELLO protocol*.

■ 20.3.1 HELLO protocol

The HELLO protocol is very 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 question is when a node should 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 the node whose HELLO packets we didn't hear 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).

20.4 A Simple Distance-Vector Protocol

The best way to understand a 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.

20.4.1 Distance-vector protocol advertisements

Each node (switch) in the protocol runs the "sending advertisement" step periodically, every `ADVERT_INTERVAL` seconds on average. The advertisement is simple, consisting of:

[(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. From this format, it should be clear why these protocols are called "vector" protocols—the advertisement consists of a vector of information, in this case cost, one per destination. Historically, they were first developed for a "distance" cost metric, which was simply the hop count, and then extended to other kinds of costs such as latency. The historic name, "distance vector" has stuck, though one might consider "cost vector" to be a more accurate moniker. Bowing to history, we will call the protocol "distance vector" even though the costs may have nothing to do with any distance metric.

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

20.4.2 Distance-vector protocol: Integration step

The key idea 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, then 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, the lowest-cost path to use would be via the neighbor j that satisfies:

$$j = \arg\min_i (c_i + l_i). \tag{20.1}$$

That is, choose the neighbor (link) such that the advertised cost from that neighbor plus the cost of the link from n to that neighbor is smallest.

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:

1. A node should update its cost and route only if the new cost is not greater than the current estimate. The question 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". For now, we can assume that infinity is some large number. 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 it just needs to be larger than the longest shortest path (weighted by the costs) in the network.

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

3. If a node n is currently using a route sent by a neighbor m for some destination d, and m stops advertising a cost for d, then n should assume the worst and conclude that m no longer has a route for d. (There are ways to design and implement the protocol so that this worst-case conclusion is not called for, but those are more complicated than our simple protocol.) At this point, n doesn't have a route for d and should stop advertising a route.

4. If a node does not have a route to a destination, then it should (or could) advertise that destination with a cost of "infinity". If this is done carefully, then the mechanism described in the previous step may not be necessary. The designer may pick one or both mechanisms in the protocol.

■ 20.4.3 Performance

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 as described the size of the advertisements grows linearly with the size of the network, and because these advertisements are periodic, the bandwidth consumed also grows linearly. There are several ways to reduce this consumption and avoid the periodic refreshes of the full routing state in each advertisement to scale such protocols to larger networks. In addition, large networks use hierarchical design and addressing to avoid sending one advertisement per destination. We will get back to these issues later in the term.

The more serious problem with the simple distance-vector protocol is how it handles link and node failures. When failures occur, it will turn out that this protocol behaves poorly. We will study these problems and develop solutions in the next lecture. Unfortunately, it will turn out that these solutions are a two-edged sword: they will solve the problem, but do so in a way that does not work as the size of the network grows. As a result, such a 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.

■ 20.5 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 *all* its neighbors and the link costs to them in the advertisement step. Moreover, upon receiving the advertisement, a node *re-broadcasts* the advertisement along all its links. This process is termed *flooding*.

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

■ 20.5.1 Flooding link-state advertisements

Each node uses the HELLO protocol (Section 20.3.1) 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 neighbor whose HELLO packets have been received in the past $k\cdot$ `HELLO_INTERVAL` time units by the originating node, and the "linkcost" refers to the cost of the corresponding link.

In addition, the LSA has a sequence number 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.

■ 20.5.2 Integration step: Dijkstra's shortest path algorithm

The final step in the link-state routing protocol is to compute the shortest (minimum-cost) paths from each node to every destination in the network. Each node independently performs this computation on its version of the 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).

We model the map as a *graph*, consisting of vertices (nodes) and edges (links). Over the past few decades, a large number of algorithms for computing various properties over graphs have been developed. In particular, there are many ways to compute the shortest path between any two nodes on a map. For instance, one might use the Bellman-Ford method developed in Section 20.4. 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 map 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, and bears his name. 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 works as follows. It 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 Y t Z instead of the original sub-path, and then continue from Z to Y. By 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 one 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 20-2 shows an example of the algorithm in operation.

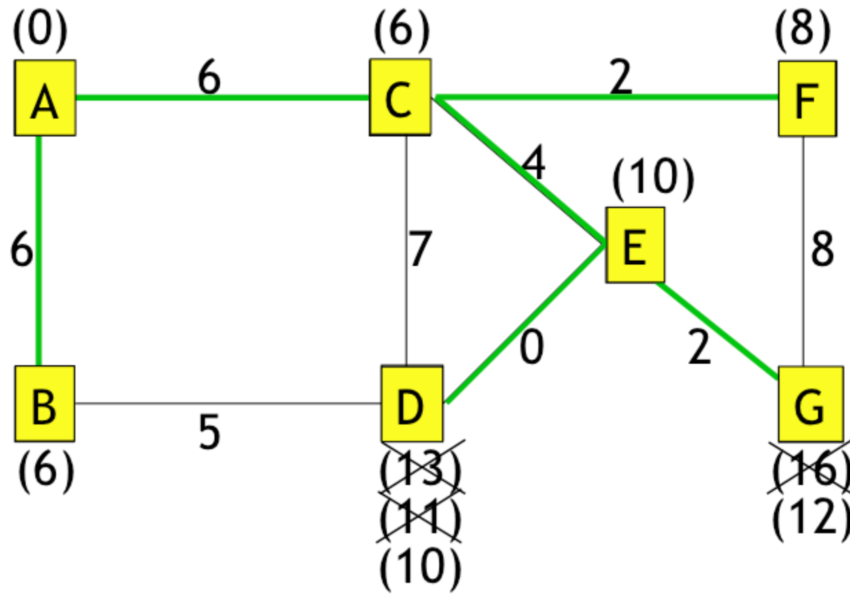

Figure 20-2: Dijkstra's shortest paths algorithm in operation, finding paths from A to all the other nodes. Initially, the set S of nodes to which the algorithm knows the shortest path is empty. Nodes are added to it in non-decreasing order of shortest path costs, with ties broken arbitrarily. In this 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.

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, `spcost`(v), for each node v. Initially, `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 `spcost`(w) is larger than `spcost`(u) + `linkcost`(uw). If it is, then update `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 all 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 by setting it to w's parent and relying on the second step above to determine the link, we will have solved the problem.

   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.

We will continue our discussion of routing protocols in recitation and in the next lecture...

■ 20.6 A Preliminary Comparison Between Distance-Vector and Link-State Protocols

We are now in a position to assess the relative merits and drawbacks of these two protocols, but only in a preliminary way. A more thorough comparison will wait for the next lecture, where we will discuss how each protocol handles failures and "churn" in the network. For now, we will compare the protocols along the following metrics:

1. Bandwidth consumption: 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 sequence number, we find that the total bandwidth consumed is $(4km^2 + 2km)/$`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.

2. Integration step assumptions: