Lecture 8: Intradomain routing

Anirudh Sivaraman

2017/10/10

Up until now, we have talked about the top two layers of the networking stack: the application and transport layers. This week we’ll start with the routing layer. This is the part of the network that gets packets from one end of the network to the other through a sequence of routers, called a path. The job of the routing layer is to find such a path between and source host and any destination host (routing), and then to send packets along this path (forwarding).

These two functions: routing and forwarding, correspond to the two planes that are present on every router. The control plane handles routing and finds paths between any two hosts/routers in the network. The data plane actually sends (or forwards) packets along these pre-computed paths when a new packet shows up at the router. The interface between these two planes is the forwarding (or routing) table, which is essentially a look-up table (or a dictionary or hashmap or map or hash table depending on which term you prefer).

Let’s look at the content of the look-up tables at any particular router. The keys in this look-up table are the destination addresses of the hosts in the network. The values in this look-up table are the output ports of this router on which a packet must be sent out so that the packet eventually finds its way to the destination. These output ports correspond to the next hop along the path from the source to the destination. Figure 1 shows the forwarding tables for four nodes connected in a straight line topology.

The control plane writes entries (destination addresses and next hops) into the forwarding tables, while the data plane reads these entries by looking up the next hop for the destination address carried by the incoming packet. These two planes operate at very different timescales. The control plane needs to update or rewrite the forwarding tables roughly every time the topology changes because a new router joined the topology or an old one failed. The data plane, on the other hand, needs to read the forwarding tables on every incoming packet at the router. The rate of topology changes is typically much lower than the packet rate at a router. For instance, for a router that supports 1 Tbit/s of aggregate forwarding capacity, this translates into 1B packets/s, assuming 1000 bit packets. That’s a packet every 1 ns in the data plane. Topology changes, on the other hand, are relatively infrequent. Even in large networks (e.g., the ones inside a large company like Google, Microsoft, or Facebook), topology changes happen at the rate of at most a 1000 times a second. That’s a change every 1 ms in the control plane. Hence the timescales differ by six orders of magnitude.

As a result, the data and control planes are implemented very differently. Because the data plane operates at such small time scales, it is built using dedicated hardware that is specialized for table lookups. This hardware is called a switching chip or a switching ASIC (Application Specific Integrated Circuit). The control plane is implemented using a general-purpose CPU like an Intel processor because that is sufficient for the rates required by the control plane.

Keeping with this control and data plane dichotomy, our discussion of the routing layer will also be divided into two parts. This week we’ll look at the control plane, and the week after the midterm, we’ll look at the data plane. Let’s begin with the control plane. The job of the control plane is to compute the forwarding tables shown in Figure 1 and write them into the router. The job of the control plane can be summarized by a routing algorithm, which computes paths between any two hosts/routers within a network. We’ll now discuss two specific routing algorithms.
In this lecture, we’ll be looking at routing algorithms within any one of the networks that constitute the Internet. These networks are typically owned and operated by a single autonomous entity, who typically has complete control over all devices within their network. Such networks are called domains (or) autonomous systems. \footnote{This is distinct from the use of the term domain in the context of the Domain Name System.} We’ll use the term domain here consistently.

Because a single autonomous entity owns and operates the entire network, it is free to do as it pleases when choosing among different alternative paths between a source and a destination host/router. Typically, intradomain routing tries to minimize some kind of path metric, which in turn is (usually) the sum of a link metric for every edge/link within a path. One reasonable choice for the link metric is the minimum latency (i.e., the propagation delay, which excludes any queueing delays) along a network link; correspondingly, the path metric is the minimum latency along a path.

There is a class of routing algorithms, colloquially termed dynamic routing algorithms, which incorporate the current queueing delay in the link metric. However, such algorithms are harder to reason about because their performance depends on the extent to which queues build up, which in turn is a function of the number and nature of applications using a network. We won’t be dealing with such algorithms in this course both because they are hard to reason about and they are not very widely deployed. Instead, we’ll concern ourselves with static routing algorithms, which depend on static (or at least relatively less dynamic) properties of the network, such as a link’s propagation delay or its capacity, or the network’s topology—in contrast to dynamic (or quickly changing) properties such as queueing delays, queue sizes, link utilization, end-host latencies, or end-host throughputs.

\section{Intradomian routing algorithms}

In this lecture, we’ll be looking at routing algorithms within any one of the networks that constitute the Internet. These networks are typically owned and operated by a single autonomous entity, who typically has complete control over all devices within their network. Such networks are called domains (or) autonomous systems. We’ll use the term domain here consistently.

Because a single autonomous entity owns and operates the entire network, it is free to do as it pleases when choosing among different alternative paths between a source and a destination host/router. Typically, intradomain routing tries to minimize some kind of path metric, which in turn is (usually) the sum of a link metric for every edge/link within a path. One reasonable choice for the link metric is the minimum latency (i.e., the propagation delay, which excludes any queuing delays) along a network link; correspondingly, the path metric is the minimum latency along a path.

There is a class of routing algorithms, colloquially termed dynamic routing algorithms, which incorporate the current queueing delay in the link metric. However, such algorithms are harder to reason about because their performance depends on the extent to which queues build up, which in turn is a function of the number and nature of applications using a network. We won’t be dealing with such algorithms in this course both because they are hard to reason about and they are not very widely deployed. Instead, we’ll concern ourselves with static routing algorithms, which depend on static (or at least relatively less dynamic) properties of the network, such as a link’s propagation delay or its capacity, or the network’s topology—in contrast to dynamic (or quickly changing) properties such as queueing delays, queue sizes, link utilization, end-host latencies, or end-host throughputs.

\section{Link-state routing algorithms}

The first class of routing algorithms we’ll discuss are called link-state algorithms. To build some intuition for this, let’s assume you’re an omniscient network operator looking down into the network from above. Because you’re omniscient, you know everything there’s to know about the network: its entire topology (i.e., who is connected to whom), the link capacities on each of the links/edges, and the propagation delays on each of the links. We’ll also assume you’re supplied with a formula to compute the link’s metrics from the link’s properties such as its propagation delay and capacity. Finally, we’ll assume that the path metric is the sum of the link metrics along the path, and you’re interested in minimizing the path metric.
How would you solve this problem assuming you’re the omniscient network operator? At its core, this problem is no different from computing the shortest path on a graph, where you initialize the graph’s edges with weights corresponding to the link metrics. Once you have done that, you need some way to compute the shortest path between every pair of nodes in a graph. For this, you could use a standard shortest path algorithm from CSCI-UA.0310 such as Dijkstra’s algorithm, the Bellman-Ford algorithm, or the Floyd-Warshall algorithm.

OK, now how do we do this in a network where you don’t have the benefit of omniscience? In other words, each router can only see its own local neighborhood: who it is connected to, what the link capacities to each of its neighbors is, and what the propagation delay to each of its neighbors is. But, can the routers cooperate with each other to exchange information about their neighborhoods so that every router has an omniscient or global view of the network after this exchange? In other words, can we replace the centralized graph computation algorithm with a distributed algorithm, where no router has global visibility of the network when it boots up?

This is the essence of link-state algorithms. Each router first collects information about its local neighborhood by sending probe packets out of all its output ports to see who it is connected to and what the properties of the connecting link (propagation delay and capacity) are. This is the link state because it captures the current state of the router’s links. The routers then exchange this link state information with each other. They do so by ensuring that each router’s link state information is broadcasted to the entire network so that at the end of the broadcast process, every router has every router’s (including itself) link-state information.

This broadcast works by having each router forward any link-state advertisements (LSA) (a packet containing information about the neighborhood for a particular router) that it receives to its neighbors. These neighbors then forward the LSAs to their neighbors, and so on, until the LSAs reach the edges of the network. Some care must be taken to ensure that each LSA received at a router is forwarded to a neighbor of the router only once, and we’ll look at this in the assignment. Without this, the LSAs can keep getting forwarded in the network and the broadcast process will never stop.

Once the broadcast process has completed, each router has the LSA from every router (including itself). These LSAs correspond to the adjacency list for each router in the network, and the combination of all these LSAs gives us the adjacency list representation of the network’s graph. With this representation available, each router can independently run a single-source shortest path algorithm, such as Dijkstra’s algorithm to calculate shortest paths to every destination address from itself. The output of Dijkstra’s algorithm can be used to determine the next hop along the shortest path to each destination. With this information, the router can fill in its forwarding table with an entry for each destination.

3 Distance vector algorithms

An alternative to link-state algorithms is the class of distance-vector algorithms. In link-state algorithms, until the broadcast process is completed, no router has computed routes or next hops to any destination. In other words, the link-state algorithm sequences routing into two parts: an information gathering phase where every router accumulates enough information to reconstruct the network’s graph and a route computation phase where routers actually compute routes based on the graph that they have accumulated.

The distance vector algorithm is more incremental in the sense that the information gathering phase and route computation phases are interleaved without being sequenced one after the other. The consequence of this is that as time progresses a router has computed the shortest paths for an expanding frontier of other routers/end hosts around it. At the beginning of time, each router knows shortest paths (as measured by the path metric) whose path lengths (as measured by the number of hops) are at most 1. As time progresses, each router knows shortest paths with path lengths at most 2, at most 3, and so on. So, a router learns shortest paths to nearby routers quickly and farther routers slowly.

How does the algorithm actually work? The principal idea underlying distance vector algorithms is this: a router’s shortest path to a destination can be decomposed into an edge to one of the router’s neighbors concatenated with a shortest path to that destination from that neighbor. Why is this? If a router’s shortest path to \( D \) consisted of a non-shortest path to \( D \) through a particular neighbor, the non-shortest path could be
replaced with the neighbor’s shortest path to $D$, yielding a shorter path to $D$ in the process. This argument is at the core of why Dijkstra’s algorithm works as well, so reviewing material from CSCI-UA.0310 may be helpful.

Now, onto the algorithm itself. The algorithm is a distributed version of the Bellman-Ford algorithm. Each router maintains its current estimate of the best path to the destination. Let’s call this $d(v)$, where $d$ represents the router’s current estimate of the shortest distance to destination $v$. This is called a distance vector because it is a vector of distances to each destination in the network. Whenever the distance vector of a router changes, the router exchanges the distance vector with its neighbors alone. In particular, unlike link-state algorithms, it does not broadcast the distance vector to the entire network.

When a router receives a distance vector $d_N$ from one of its neighbors, it incrementally updates its own distance vector $d_R$ for every destination $v$, as follows:

$$d_R(v) = \min(d_R(v), d_N(v) + \text{link\_metric}_{R,N})$$  \hspace{1cm} (1)

This follows from the earlier intuition: any shortest path can be broken up into an edge to a neighbor concatenated with a shortest path from the neighbor.

That’s it. That’s the entire algorithm. When do we stop this algorithm? The beautiful thing about the algorithm is that it stops automatically because if the distance vector does not change, the router does not send it out to its neighbors. We can prove that when the algorithm quiesces (i.e., there are no more distance vectors flying around), the distance vectors would be at their correct values. These are the values computed by a shortest-path algorithm such as Dijkstra’s algorithm.

So when does the algorithm stop? This depends on the maximum length of a shortest path in the network. This is because, as we remarked earlier, each router builds up an expanding frontier of routers to which it knows shortest paths. Hence, the farthest router determines how long the algorithm takes to stop (also called convergence time in the routing literature).

One detail we have omitted is how the best next hop to a destination is maintained at each router. In the incremental update step, if a router chooses to update its shortest path to go through its neighbor, it updates its next hop to that neighbor. Again, when the algorithm quiesces, we can prove that the next hop will be at its correct value. A good illustration of the distance-vector algorithm, courtesy of Prof. Nick Feamster at Princeton, is available at [https://www.youtube.com/watch?v=x9WIQbaVPzY](https://www.youtube.com/watch?v=x9WIQbaVPzY).

### 4 Comparing link-state and distance-vector algorithms

One difference we have already alluded to: link-state separates out the information gathering and route computation phases, while distance-vector interleaves them. As a result, the distance-vector algorithm incrementally builds up shortest paths as opposed to the link-state algorithm, which doesn’t have any valid shortest path during the information gathering phase.

Assuming the computational steps (running Dijkstra’s algorithm and performing the incremental updates) in the two algorithms are instantaneous, the decision between the two comes down to the size of the network.

For a network with a small number of nodes, the LSAs can be broadcast across the network quite quickly, after which the route computation can be run quite quickly at each router. For a network with a larger number of nodes, distance vector is preferable, because the broadcast step starts taking longer. Further, every change to the network requires us to rebroadcast LSAs for link-state routing. For distance vector, the amount of network traffic generated in response to a change in the network is roughly proportional to the change in the shortest paths that is induced by the network change. In summary, link-state algorithms are preferable in a small network because broadcast is cheap and quick. It is also easier to reason about link-state algorithms when routers fail.

---

3 This includes the time at which the router just boots up and its distance vector is first initialized. A freshly initialized distance vector at a router has the entry $\infty$ for any router that is not directly connected to the router and the link metric for any router that is directly connected.

4 The computation step just needs to be much faster than the time it takes to broadcast an LSA across the whole network or send a distance vector to a router’s neighbor.

5 This oversimplifies distance vector, which needs some important modifications to handle router failures. We won’t get into failure handling in this course.
5 What we haven't covered

This section provides a brief summary of what we have left out in the topic of routing. This won’t be tested unless I end up covering it in detail later, but is nonetheless useful if you want to understand routing at the next level of detail.

1. We don’t look at how we handle churn, i.e., router failures and router additions, for both the link-state and distance-vector algorithms. All our discussions have been focused on what happens at the beginning of time when the network just boots up. Dealing with churn is probably among the most painful parts of routing as far as a network operator is concerned.

2. We have only looked at routing based on destination address, where the destination’s IP address determines the packet’s path through the network. There is a rich class of routing algorithms that deal with policy routing, which goes beyond the destination address field in the packet header and uses other packet headers to inform its routing decisions. For instance, a cloud provider may want to ensure that one tenant’s traffic does not enter a particular part of the network that is being used by another tenant for security reasons. It could make this decision based on the source IP address of the tenant.

3. Another issue that is important operationally is scaling to large networks. At some point the size of the forwarding tables becomes a concern because high-speed memory (required for data plane lookups) is limited. The common solution to this problem is hierarchy and the separation of routing into intra and interdomain routing (which we’ll discuss next lecture) is the simplest example of this. Routing protocols also make use of the hierarchical structure of IP addresses themselves to significantly reduce the number of forwarding table entries they need to store.