Forwarding Sometimes Referred to as Routing

Introduction

With this module, we get into the heart of what networking is all about. These next two modules are probably the most important in the course. This lecture will cover what is generally called routing. Next, we will look at the particulars of the Network Layer, or the first layer where we really consider relaying, and how routing is used to get PDUs to their destinations. Finally, we will cover Naming and Addressing, which is by far the most important aspect of networking and the least understood.

Wait a Minute! There Was Relaying in Ethernet!

Yes, there was and later there was limited routing. First, the networking field was trying to follow Dijkstra's idea from operating systems that once a function was performed, it didn't need to be repeated in higher layers. *(Of course, we have seen that functions are repeated in layers of different scope.)* As for relaying, there was relaying in every layer. And Ethernet demonstrated that neatly: Hubs in the Physical Layer; Bridges in the Data Link Layer; and Routers in the Network Layer. Relaying was necessary to increase the scope of layers, so it was ignored. *(Note that this follows the principle we just mentioned but that was not recognized at the time.)* Initially, routing was only done at the Network Layer, *(which fit with the theory that functions didn't repeat.)* Those working on the Network Layer issues staked out routing as a topic that was theirs. However, as we saw in the last module, as larger Ethernets were deployed it was clear that for increased efficiency (and falling hardware costs), it wasn't necessary for every Ethernet PDU to be seen by the entire Ethernet. So, in the 80s, Learning Bridges were created. This was more filtering than routing, so it didn't raise any issues. Also, as Ethernets were deployed, redundant links were introduced to increase the reliability of the network, but this led to PDUs looping indefinitely. To prevent that, it became necessary for the bridges to exchange information to create a spanning tree. (This was an interesting case. As we will see shortly, the crux of one of the routing algorithms is to compute a spanning tree rooted at each node in the network, but it was not yet in use in the Network Layer, so this did not raise many issues either. But it was becoming a slippery slope. This algorithm was used to define a spanning tree for suppressing loops, not for routing. Yes, it is a fine line.

Wasn't this routing? Yes.

The Network Layer groups didn't object? If they did, not much. There is some evidence that while those working on the details of the adoption of Link State routing knew it was computing a spanning tree and Ethernet bridges were doing something very similar, most of those further removed did not realize they were the same. Finally in the 2000s, IEEE 802 gave up all pretense, and adopted a variation of the IS-IS Network Layer routing protocol with some very interesting modifications.

In describing the architecture, the group working on working out the structure of the Network

Layer ran into a bit of a problem. They knew they had an overlay Internet Layer over multiple technology-dependent networks. There seemed to be two cases:

1. The traditional technology-dependent network, such as X.25, ATM, MPLS, etc. which defined specific Physical, Data Link and Network Layers; and
2. Ethernet LANs which were entirely in the Data Link Layer.

They finally said that these were not really “layers” but “roles” and in the case of LANs, the Network Layer aspects were null. One could argue that Ethernet does follow the principles and would have looked like any other technology-specific network, if not for the global scope of the flat 48-bit MAC address. To see that lets do a little gedanken experiment. Lets see what might have happened if the MAC address wasn’t 48-bits. What might have been the layers and address spaces of Ethernet?

1. The Data Link Layer or Media-Access Layer is a multi-access broadcast layer of limited length. (Initially, the maximum length was about 1km and got shorter as speeds went up.) How large should its address space be? Probably 9 or 10 bits would be sufficient. An Ethernet segment with 500 or 1000 stations would be pretty heavily loaded! (The only exception here, of course, would be Ethernet Hubs that relay at the physical layer, which would be the same Ethernet domain and might argue for a longer address. But hub failures tend to jam the entire Ethernet domain, so one would want to limit the size to limit the impact on the organization.)
2. The Network Layer – would be Ethernet bridges that relay traffic between Ethernet segments. Clearly here we need a larger address, but let’s assume the technology is still intended to be “local”, an address space of 16, 24, or 32 bits. The Network Layer could use spanning trees, learning bridge, OSPF, or IS-IS, and might not be all that “local.” (IEEE 802 is currently using IS-IS for shortest-path routing. The only limit on this is that 48-bit MAC addresses are not location-dependent. We will explain more about this later in this module.)
3. The Internet Layer – would use IP either v4 or v6 with BGP for routing.

This shows that Ethernet is not a special case, obeys the principles were developed and fits nicely in the model. It also avoids the security issue that the MAC address has.

The big disadvantage to this, of course, is it requires an enrollment procedure for both the Data Link and Network Layers to assign addresses. The 48-bit address was chosen primarily to make enrollment and address management simple. At the time, it seemed like a good idea (I certainly thought so!). But it was early when the 48-bit address was developed and the vagaries of using a device-id as an address only surfaced later.

As the bandwidth of the physical layer is increased, the length of an Ethernet segment becomes shorter. An Ethernet segment would be able to support fewer interfaces, so a smaller address field could be used (Less overhead). (Note: that because this is the Data Link Layer, its scope is quite limited. Hence, a smaller (or no) address field could be used without impacting interoperability. There is no requirement that all Data Link Layers have the same size address fields.) The bandwidth of the media can increase to the point, e.g., Gigabit Ethernet, where it becomes a point-to-point technology and no address fields are necessary.

Is this now a special case? No, it is just the traditional case we find with routers connected by point-to-point lines. The

Data Link Protocol might do some error checking and could dispense with address fields altogether. Note that all of this is just leveraging what we learned before about separating mechanism and policy.

Should we re-work Ethernet to fit more closely? I wouldn't unless all of this yielded a huge advantage in cost and/or functionality. For example, we mentioned in the last module that it can be shown that WiFi and VLANS are basically the same thing and could be unified. That alone wouldn't be sufficient but a few things like that might make an argument for an Ethernet II. But current Ethernet isn't that far off the model, and it integrates easily and there is a lot of stuff out there with MAC addresses.

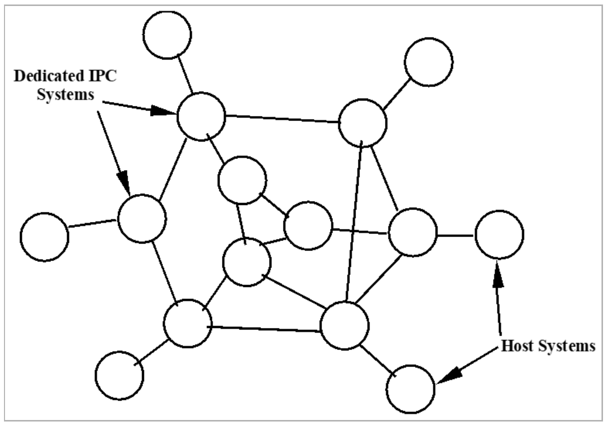
After that Digression on to the Network Layer

We learned in the last lecture was that when a layer has more than two members, an identifier is needed to distinguish which "end" a PDU is destined for. Now in this lecture, we will refine that concept and see when it is advantageous to embed hints in this identifier to indicate "nearness", so that from a distance, the same routes can be used for destinations that are near each other. In other words, these identifiers are treated as addresses, i.e., are location dependent. They have the same property that a mailing address has that from a distance one doesn't need to know precisely where the destination is but only the general direction the PDU should be forwarded.

As we have seen, all layers do the same functions, just over different ranges of the problem and different scope. That attitude is beginning to change. IEEE 802 now uses one of the routing protocols discussed in this module with some innovative differences.

As we saw in the IPC Model, the Network Layer can introduce errors, although from different sources and with different characteristics. While Tanenbaum's textbook includes traffic management and some aspects of congestion management in the Network Layer chapter, we will postpone that to the next Module. Because the Internet did congestion control in the Transport Layer, where it doesn't belong, we will hold off until we have covered the particulars of error and flow control over relays to cover congestion control and traffic management all at once.

N Systems with Relays (On the Cheap)

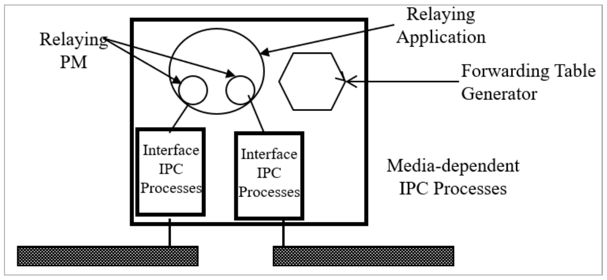


By dedicating systems to IPC, reduce the number of lines required and even out usage by recognizing that not everyone talks to everyone else the same amount.

This is where we are in development of the IPC model. We are doing communication on the cheap, with dedicated relays, sometimes called *routers*. A wire from a host no longer connects directly to just one destination. To get to a destination, a PDU must be relayed by one or more routers. Therefore, it will be necessary to tell the relays where the PDUs are going.

Dest Addr	Src Addr	Dest-port	Src-port	Op	Seq #	CRC	Data

Common Relaying and Multiplexing Application Header



Before we consider how we determine what outgoing interface, the PDU should be sent on, let's look in detail at one of these relay systems. It is difficult to draw the picture because we are interested in what goes on inside the box. Keep in mind that these relays will generally have more than two interfaces (lines connecting other relays or hosts). Each router consists of one of the IPC Facilities we developed in the previous steps for each interface. These constitute as many *1-layers* as there are interfaces. Each IPC Facility is a member of a point-to-point 1-layer consisting of two members. (See the network figure above.) Within each IPC Facility of these 1-layers, there is a protocol operating that coordinates the transmission of PDUs from the sending 1-IPC Facility to the receiving 1-IPC Facility.

Now how does a PDU get to a router? Clearly, we are going to need a 2-layer that includes all of the hosts and all of the routers operating over all of the 1-layers. There will be a 2-data transfer protocol operating in this 2-layer to carry the data. Unlike the 1-layer that interfaces to a wire, we didn't need to tell the IPC Process the destination of the PDU. We could rely on the fact that if it went in at one end, it could only come out at the other! Here we have a layer with "more than 2 ends." The multiple ends are created because we are relaying within the layer. The 2-IPC Process in a host is going to have to label each PDU with an identifier for the destination of the PDU. (How were they assigned? We will assume they were for now. But later we will see how these names are assigned during the Enrollment Phase.)

The protocol for 2-layer will need these source and destination names added to its PCI, indicating who is sending the PDU and who is the destination.

The Application will pass an SDU to the 2-IPC Process, which will encapsulate it in a 2-PDU and deliver it to the 1-IPC Facility, so that it is sent across the wire to a router.

When a PDU arrives at the 1-IPC Facility, the 1-PCI is stripped off and the "Data" is handed up to the 2-IPC Process as "Data" Of course, this "Data" is a 2-PDU, which after SDU Protection has computed its error code and determined it is okay is handed to the 2-Relaying Task. The IPC Process inspects the address: if it is its address, it must hand it some other process (currently outside our scope). If it is not its address, it must consult a forwarding table to determine which interface to relay the PDU on.

This lecture is about generating that forwarding table. Suffice it to say here that the forwarding table maps the destination identifier to the correct 1-port, not to the next 1-IPC Process. Notice that it is only at this late stage that we are finally talking about forwarding-ids or addresses. We haven't needed them before. The most we needed was a local identifier to distinguish that there were multiple lines/interfaces.

“But we talked about addresses in the last lecture!” You are, of course, right. We did. We had a layer with more than two ends (multi-access media), but no relaying. This is in some regards a degenerate case. Every member of the layer saw every PDU. All it had to do was recognize which address was its address.

It also had one other characteristic that we won’t get to until later. The “addresses” weren’t really addresses. We could have a flat address space, i.e., they were assigned by enumeration (as in the ARPANET) or at the factory like a serial number (as with Ethernet). Intuitively, an address is location-dependent and route-independent. In other words, it indicates where but not how to get there.

Flat addresses are “good enough” for small networks. But as often happens as time goes on, “good enough” turns out to be not “good enough.” As we will see, the other subtlety here is that “location” is not physical location but “location” relative to the graph of the layer.

From this we learn that anytime a layer has more than two ends, we’re going to need forwarding identifiers or addresses. With relays, all destinations don’t see all PDUs, so we need some way of keeping track of what is where. For “small” networks, we have algorithms for doing that. That is the subject of the rest of this lecture.

Also, this shows that there’s more to a layer than just a protocol for transferring data. This is the first evidence we have that layers aren’t just protocols, but there is layer management as well. Notice that the address is not naming the application. It is naming the node, the IPC Process that is doing the relaying.

In this lecture, we are going to cover these topics: the optionality principle, Shortest Path Routing, Flooding, Distance Vector routing, Link State, etc.

Why Do We Do Routing?

Oddly enough, we don’t do routing to determine the route that a PDU follows. You will find that people talk about it that way, but that's not really what’s going on. We do it to build a forwarding table of next hops or, as we just saw, the next interface to forward the PDU on. Here’s what we do:

Each router exchanges routing information with others (which others depends on the algorithm).

They all execute the same algorithm with what we *hope* is the same or nearly the same data. We *hope*.

They all get the same results for the optimal route through the network for some value of optimal.

Then the routers build a forwarding table (which includes who they route to next) based on where they are in that solution. If all goes well, the PDU does follow the optimal route.

If not, very weird things can happen.

Let’s Take Routing Apart

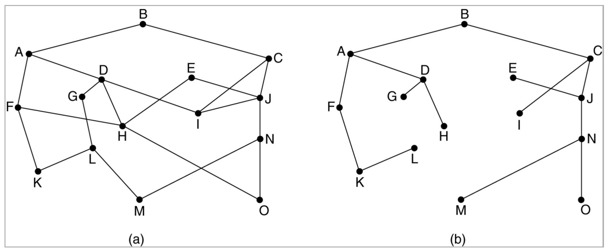
All routing protocols confuse, combining four separate problems:

1. When updating multiple copies of a database, exchanging routing information. Multiple sources have different information, and they need to form a consistent picture of the graph of the layer. (How many members of the layer are involved in this depends on the routing algorithm.)
2. Determining the connectivity and finding the routes.
3. Choosing the routes based on some metric: what we want to optimize or delay, propagation time, etc. We can weight the routes according to various criteria.
4. Creating a forwarding table.

For now, there are basically three types of routing:

1. *Link State*, which requires complete knowledge of the graph,
2. *Distance Vector*, which has partial knowledge and exchanges information with its nearest neighbors only, and
3. *Hierarchical routing*, which tries to organize, is only good for hierarchical networks. There are some other techniques that are being explored, but nothing really has taken hold.

The Optimality Principle



(a) A subnet. (b) A sink tree for router B.

Any subpath of an optimal path is optimal.

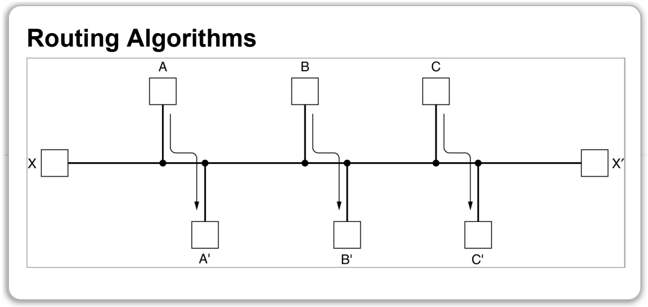
The key thing here is the optimal quality principle. This is where we pull out the graph theory. The result we need is that a subpath of any optimal path is optimal.

What we are doing, at least for some kinds of routing algorithms, is this:

- Each router is computing the optimal path from it to everything else,
- From this, each router knows for every destination what is the next hop on which it should send a PDU.
- It creates a table of these next hops and uses it to forward arriving PDUs for which it is not the destination.
- At the next hop, that router does the same thing: It has computed the optimal path from it to every destination and determines what its next hop is and forwards incoming PDUs accordingly.

Because the path to each next hop is a subpath of an optimal path, the sum of the subpaths is an optimal path.

If they don't use the same data, weird things can happen, like loops. One can get routing loops that will last for hours or days.



Efficiency, Optimization, and Fairness. One condition we want is to make sure that whatever algorithms we come up with are fair. We want optimality and fairness, but we also need stability, correctness, and simplicity.

This implies an algorithm that is stable and, once stable, tends to stay there in the face of failures. There are lots of algorithms that are not stable and will not converge no matter how long they run. An algorithm is required that converges quickly and stays at the operating point. An algorithm needs to converge quickly because it takes some time after convergence for the network to quiesce around the new stable configuration and there can be problems. Providing Quality of Service requires being unfair. Some flows are treated differently than other flows by definition. To maintain some QoS parameters may mean giving higher priority to some traffic over other traffic. However, if the traffic with different QoS is within their bounds, then it is fair. Then fairness becomes what is fair within a QoS class.

Interactions among QoS classes can also exhibit unintended unfairness and instabilities. Furthermore, since this is an internet, adjacent networks may have different routing policies. We must ensure that the interactions among them must also be stable. One of the concerns with IPv6 and the much larger addresses is that while the algorithms may converge quickly enough, the time for a network to quiesce at a new operating point will not occur before it is time for another routing update. Consequently, the update could be using transient data, which could lead to instabilities.

Fairness and efficiency may seem obvious but are far from it. Being fair may require being less efficient. For example, in this figure, it might be very efficient for X to blast away to X' but unfair to soak up all the bandwidth, so no other traffic can get from B to B', C to C'. We want this to be a fair, so that X A, B and C all get their fair share of capacity between the nodes, and none are starved out.

Routing Algorithms

Before we start, a little background on routing is in order. Many early networks used fixed routing that was configured by an operator and was changed manually when there were changes in the network. The ARPANET was the first to use a distributed routing algorithm based on what is called Distance Vector or the Bellman-Ford algorithm (which we will cover next). As the ARPANET became the Internet, Distance Vector revealed some problems with convergence. A move was made to a Link-state algorithm. As we will see, while Link State converges faster and doesn’t have other problems, it has the drawback of requiring complete knowledge of the graph. Hence, the Internet has adopted a two-tiered approach: Link-state is used for intra-domain or network routing, while a variation of distance-vector, now called path vector, is used for inter-domain or internet routing.

With that, let us look at how the Dijkstra Algorithm works.

The Dijkstra Algorithm for Computing a Minimal Spanning Tree

One of the major routing algorithms is based on a graph theory algorithm developed by Edgar Dijkstra for finding the optimal spanning tree of a graph with weighted arcs.

First, we will go through the algorithm, and then we will use the algorithm to do an example. Pay careful attention because this gets complicated. (Actually, the form of this algorithm turns up in a number of graph-like algorithms. Understand this one and you will be halfway to understanding others when you encounter them.) First, the definitions:

Let

- N = set of vertices in network
- s = source vertex (starting point)
- T = set of vertices so far incorporated
- $Tree$ = spanning tree for vertices in T, including edges on least-cost path from s to each vertex in T
- $w(i, j)$ = link cost from vertex i to vertex j
  - $w(i, i) = 0$
  - $w(i, j) = \infty$ if i, j not directly connected by a single edge
  - $w(i, j) \geq 0$ of i, j directly connected by single edge
- $L(n)$ = cost of least-cost path from s to n currently known
  - At termination, this is least-cost path from s to n

Then we have the algorithm itself:

- Initialization

  $T = Tree = s$ - only source is incorporated so far
  $L(n) = w(s, n)$ for $n \neq s$ - initial path cost to neighbors are link costs

- Get next vertex

  Find $x \notin T \ni L(x) = \min L(j), j \notin T$
  Add x to T and $Tree$
  Add edge to T incident on x and has least cost
  Last hop in path

- Update least-cost paths

  $L(n) = \min[L(n), L(x) + w(x, n)] \forall n \notin T$
  If latter term is minimum, path from s to n is now path from s to x concatenated with edge from x

to n

- Terminate when all vertices added to T

  This requires $|V|$ iterations.

- At termination

  $L(x)$ associated with each vertex is cost of least cost path from s to x
  Tree is a spanning tree

    Defines least-cost path from s to each other vertex

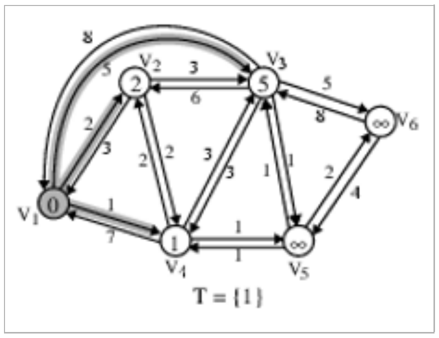
- Running time order of $|V|^2$

Now let’s do an example.

An Example of the Dijkstra Algorithm

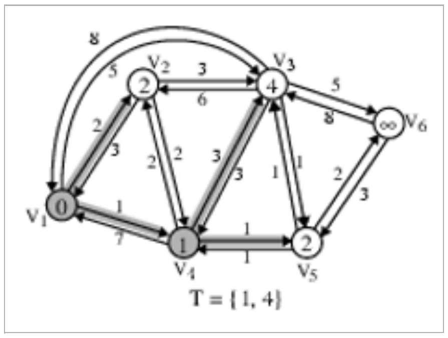
This example is from Stallings. In the figures, the numbers in the circles (nodes) in each step are the distance they are from the source according to the Dijkstra algorithm.

Dijkstra’s Algorithm: Example Graph T={1}



We start with vertex 1, V_1. It is the only thing in our solution {1}.

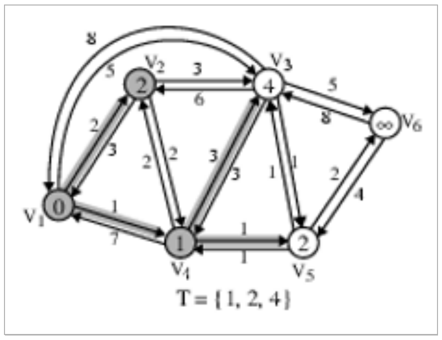
Dijkstra’s Algorithm: Example Graph T={1,4}



We want to find the minimum next hop. There are three possibilities:
V_4 is 1,
V_2 is 2, and
V_3 is 5.
(The shaded arcs are the candidates in this step.)
The minimum is V_4. So, we include V_4 in our solution: {1,4}.

Dijkstra’s Algorithm: Example Graph T={1,2,4}

Now we repeat the procedure at V_4.
V_2 is 2.
V_5 is one, for a total length of 2.



V_3

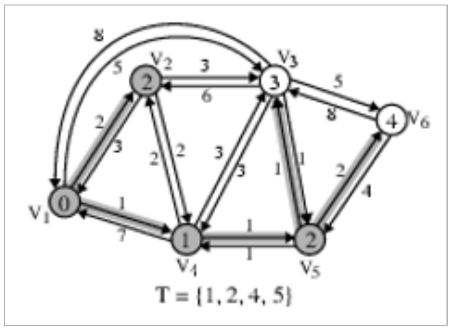
is 3, for a total length of 4.

(The shaded arcs are the candidates in this step.)

The minimum is V_2. So we include V_2 in our solution: {1, 2, 4}.

So, now we go to next step.

Dijkstra's Algorithm: Example Graph T={1,2,4,5}



From before, we know that

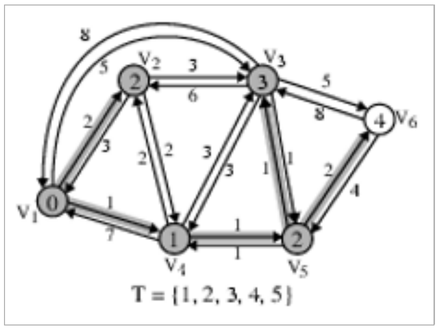
V_3 is 3, for a total length of 5 (from V_2),

V_3 is 3, for a total length of 4 (from V_4), and

V_5 is one, for a total length of 2.

The minimum is V_5. So, we include V_5 in our solution: {1, 2, 4, 5}.

Dijkstra's Algorithm: Example Graph T={1,2,3,4,5}



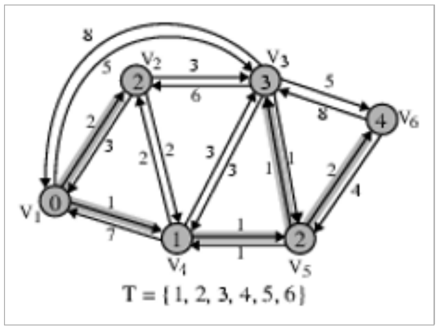
We have two possibilities from V_5.

V_3 is 1, for total length of 3.

V_6 is 2, for total length of 4.

The minimum is V_3. So, we include V_5 in our solution: {1, 2, 3, 4, 5}.

Dijkstra's Algorithm: Example Graph T={1,2,3,4,5,6}



The last node to add is V_6.

So, we include V_5 in our solution: {1, 2, 3, 4, 5, 6}.

All the nodes are included. The arcs that are shaded are the spanning tree.

Now we can build our forwarding table:

If we have a PDU going to V_2, then forward it on the link to V_2. Otherwise, forward it on the link going to V_4.

Every node in this graph will have done the same algorithm, hopefully, with the same data with similar results. If V_1 is sending a PDU to V_6, it will send it toward V_4.

When V_4 receives the PDU, it wills see that the PDU is addressed to V_6 and consult its forwarding table will say send it to V_5 and so on.

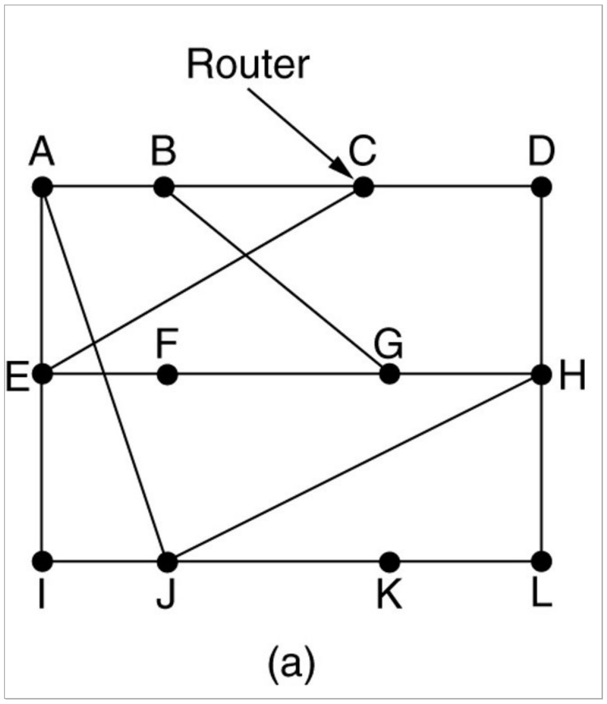
Obviously, this has limited utility because it requires complete knowledge of the graph. We can do this locally and, believe it or not, this will work up to about 100,000 nodes. Beyond that, it's going to blow up on us. Remember, this requires order V^2 iterations.

Distance Vector or Bellman-Ford Algorithm

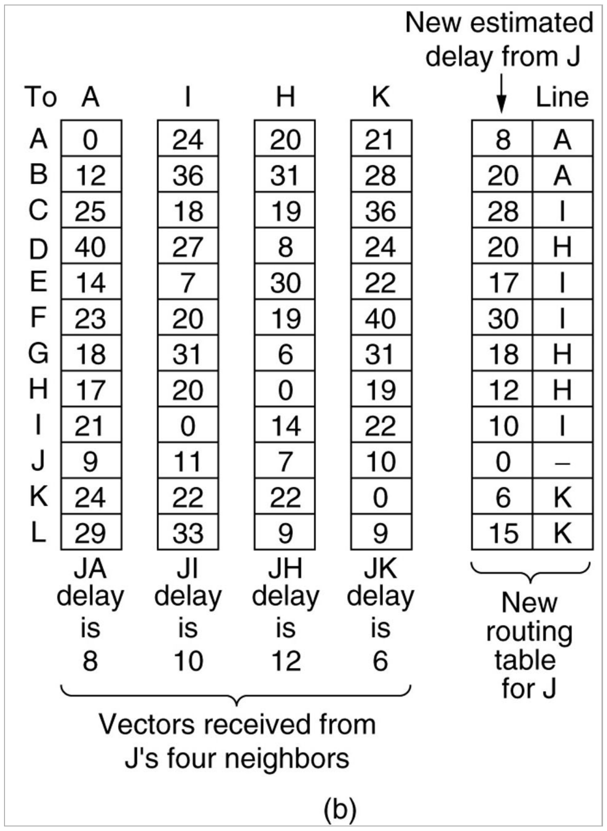
Let's consider Distance Vector or Bellman-Ford. As we just said, this is the algorithm the ARPANET started with. It has a very appealing property. Remember I have said that we were enthusiastic about creating a decentralized distributed routing network. The Bellman-Ford algorithm fit that criterion. It only exchanges routing updates with nearest neighbors. Very decentralized! But it turned out to have some problems as well.

First, let's see how it works. We have the graph above. We assume that every node has sent its nearest neighbors what they believe is the distance to all of the other nodes, i.e., distance vectors (where "distance" is the routing metric).

Distance Vector Routing



(a) A subnet



To	A	I	H	K	New estimated delay from J	Line
A	0	24	20	21	8	A
B	12	36	31	28	20	A
C	25	18	19	36	28	I
D	40	27	8	24	20	H
E	14	7	30	22	17	I
F	23	20	19	40	30	I
G	18	31	6	31	18	H
H	17	20	0	19	12	H
I	21	0	14	22	10	I
J	9	11	7	10	0	–
K	24	22	22	0	6	K
L	29	33	9	9	15	K
	JA delay is 8	JI delay is 10	JH delay is 12	JK delay is 6	New routing table for J	

Vectors received from J's four neighbors
(b)

(b) Input from A, I, H, K, and the new router table for J.

In addition, each node knows its distance to its neighbors. For the node J, we have:
The distance from J to A is 8; from J to I, 10; from J to H, 12; from J to K, 6.

From this we can construct the forwarding table for a node.

J receives the vectors above from its neighbors.
J starts by inspecting each next hop (nearest neighbor) as a possible path:

> To go to A, the distance is 8 and the entry is 0. Distance = 8;
> To go to A via I is 10+24 = 34;
> To go to A via H is 12+20=22;
> To go to A via K is 6+21=27.

A is the shortest. We put A in the forwarding table (the two-column table on the right).

Now let's do B.

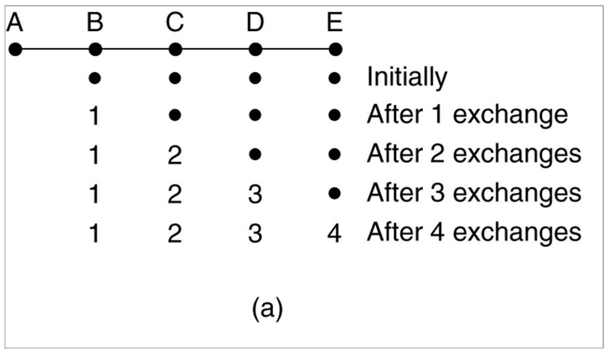
> To go to B via A is 8 + 12= 20;
> To go to B via I is 10+36=46;
> To go to B via H is 12+31=43;
> To go to B via K is 6 +28=34.

The shortest path is via A. We put 20 in the forwarding table. And we do that for all of them. This becomes our routing table.

The Count-to-Infinity Problem

Distance Vector has a problem, the “count-to-infinity” problem. It learns bad news quickly but good news slowly and hence is slow to converge.

Distance Vector Routing


A B C D E
● ● ● ● Initially
1 ● ● ● After 1 exchange
1 2 ● ● After 2 exchanges
1 2 3 ● After 3 exchanges
1 2 3 4 After 4 exchanges
(a)

The count-to-infinity problem.

Consider this example: initially, A is down and comes up.

Everyone finds out fairly quickly.

Then A goes back down.

B wants to send A, but knows that its line to A is down.

C says, “I get to A!” But C’s path to A is through B.

B tells C it can’t get there.

Then, D says, “I get to A!” But D’s path to A is through C.

C tells D it can’t get there.

And so on.

There have been attempts to fix this. But the fundamental problem is that if a neighbor says it has a path to a destination, there is no way to know if you are on the path.

Link State Routing

Read

Read Tanenbaum Chapter 5, Section 5.2.5, pp. 377–382.

We know how the Dijkstra algorithm works, now let’s see how it is used in link state routing and what is necessary to go around it.

Each router must do the following things:

- Discover its neighbors and learn their network address.
- Measure the delay or costs to each one.
- Construct a packet telling what it just learned and send this packet to all the other routers. And then compute the shortest path to every other router.

An Aside

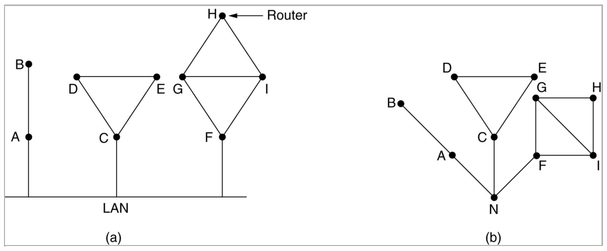
We are doing routing because we are relaying PDUs to the addresses we assigned them. Normally, we think of addresses as being location-dependent, and route independent. They tell you where something is without telling you how to get there.

When I go to my local post office and mail a letter to 40 rue de Rivoli, Paris, France, my local post office doesn't have to know where 40 rue de Rivoli, Paris, France, really is. They just know to put it in a bag going East. Notice that neither of these algorithms use that property of addresses. They just use them as labels to keep track of which nodes have been visited by the algorithm.

Learning About the Neighbors

When a router comes up, it has to learn about its neighbors. It sends a query to each point-of-attachment to announce itself and to get an estimate of delay to that point of attachment. There are some special cases that have to be handled as illustrated in the figure. The network diagram on the left has a LAN. The LAN or any broadcast or multiaccess media will be modeled as a single node.

Learning About the Neighbors

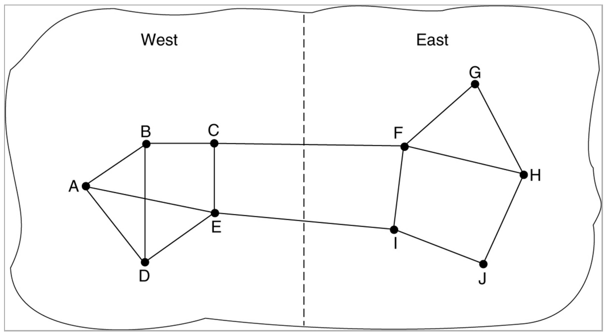


(a) Nine routers and a LAN. (b) A graph model of (a).

Setting the Costs of the Link

This is critical and the wrong choices can cause instabilities in the network. Various metrics are used. A common approach is to set cost to be inversely proportional to the data rate to make high-capacity lines appear less expensive than low-capacity lines. It is not a good idea to include load in the cost. First, it can change rather quickly, and second, it can lead to “router flap.”

Measuring Line Cost



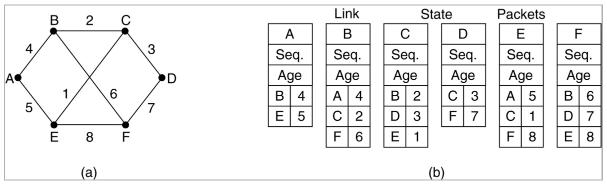
Most link state implementations use delay as a metric, which may be measured by computing round trips, queue length, etc. Also, we have to be careful about choosing the best one, or we can get oscillations.

Consider the figure. If load is used as the metric, and the algorithm is choosing the “best” route, then it might happen that the load on the northern route is low and so it appears to be the best. All traffic is routed to the northern route, and it becomes congested and slow. At the next router update, the southern route appears lightly loaded and it is chosen, so now all traffic is routed to the southern route. It would continue to flip back and forth. Load doesn’t make a good estimate of cost. Often, delay is a better metric, and then consider load separately. Assigning metrics that would lead to this sort of instability needs to be avoided. Of course, this is more difficult when the metric is determined by the conditions in the network. The important property is that the metric be . . . well, a metric. It must obey the properties of a metric, especially the triangle inequality.

Building the Link State Packets

This is the critical step in the process. Every router is going to update every other router in the network. To do that it must flood the network. It is going to be impossible for a router to avoid receiving duplicate updates. Routing protocols go to great lengths to ensure that all of the routers have a consistent database. Most of a “routing protocol” is really more about updating multiple copies of a database.

Building the Link State Packets



A	B	C	D	E	F
Seq.	Seq.	Seq.	Seq.	Seq.	Seq.
Age	Age	Age	Age	Age	Age
B 4	A 4	B 2	C 3	A 5	B 6
E 5	C 2	D 3	F 7	C 1	D 7
	F 6	E 1		F 8	E 8

(a) A subnet. (b) The link state packets for this subnet.

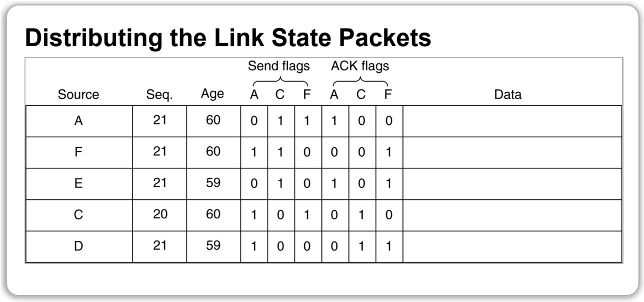
Each router builds a packet with the information it has learned and sends it to all of the other routers. The packets have both a sequence number to eliminate duplicates and age. Age is decremented with time and the information discarded when it reaches zero. New information is forwarded on all links except the one it arrived on. Duplicates are discarded. All updates are ack'ed. Another problem arises because of sequence number rollover. This could cause new information to be mistaken as old information (when 254 < 35). The Age field is included to distinguish old information from new information.

Notice that most of the discussion in the book is not about determining routes but about getting reliable data. "Routing Protocols" are not about routing, but about loosely replicated databases, a distributed database problem. This solution was very much affected by the resource constraints at the time. There weren't resources or the time for real database synchronization among all parties. There still isn't time, especially when responding to a failure. Recently, there have been new ideas that also make the algorithm loop-free. There are still questions about how these techniques scale.

Constructing the Database

The critical part of this algorithm is to distribute the information to all other routers without causing a broadcast storm.

When an update arrives, it is set aside for a while to see if others arrive from the same source. Each router maintains a table like the one in the figure to keep track of the processing of arriving updates. This figure refers to the network in the figure above or in the textbook Figure 5-12a for node B. This keeps track of where the update came from, so the information is only forwarded to other nodes. It has columns for each of its nearest neighbors, A, C, and F, to indicate where it is to be sent and who is to be ack'ed.


Distributing the Link State Packets

			Send flags			ACK flags			
Source	Seq.	Age	A	C	F	A	C	F	Data
A	21	60	0	1	1	1	0	0	
F	21	60	1	1	0	0	0	1	
E	21	59	0	1	0	1	0	1	
C	20	60	1	0	1	0	1	0	
D	21	59	1	0	0	0	1	1	

To ensure that they use the right information and to curb flooding they exchange this kind of information with every other router. The routing update is forwarded on every link that a copy didn't arrive on.

Let's go through the table above:

- First row: a PDU from A. Must be Ack'ed to A, and forwarded to C and F (flag bits).
- Second row: a PDU from F. Must be Ack'ed to F, and forwarded to A and C.
- Third row: a PDU from E: it is a duplicate, once from A and once from F. Must be Ack'ed to A and F, and sent to C, dropping one of the PDUs.
- Fourth row: a PDU from C. Must be Ack'ed to C, and forwarded to A and F.
- Fifth row: a PDU from D. Must be Ack'ed to C and F, and forwarded to A.
- And so on.

However, Link State doesn't scale. We can't have every router in the Internet exchanging link state information with every other router in the Internet. That clearly is not going to work.

Consequently, there's a two-tier scheme. We use link state within autonomous systems, i.e., subnets, and do something like distance vector among networks.

The Important Points from This

Link State converges faster than Distance Vector and to the same answer. Link State avoids the count-to-infinity problem and is less likely to develop loops.

Notice that neither of these algorithms use the location-dependence property of “addresses,” i.e., “nearness.” They are only used as labels to keep track of nodes touched by the algorithm.

However, Link State doesn’t scale. We can’t have every router in the Internet exchanging Link State PDUs with every other router!

There is a two-tiered routing scheme:

1. Link State is used within autonomous systems (subnets), called *intra-domain*, e.g., network routing.
2. A variation of Distance Vector is used among networks, called *inter-domain*, e.g., internet routing.

Hierarchical Routing…That Isn’t

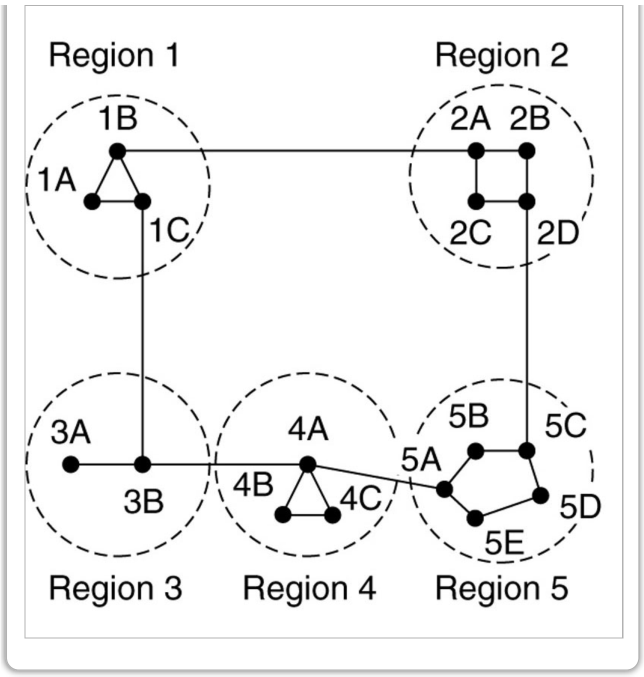
Read

Read Tanenbaum Chapter 5, Section 5.2.6, pp. 382–384.

If the address space is flat, then every address requires a router table entry. As the network grows so does the table. Any network will, by its nature, coalesce into interconnected subnets. Look at the ARPANET graph: there was a Boston subnet, a Washington, DC subnet, a Bay Area subnet, and a southern California subnet, with a few sites outside that. A corporate network will do the same, by site, by building, by floor of a building, by departments, etc. Addressing can take advantage of this.

At the periphery, these subnets follow organizational or physical constraints as just illustrated. As we move toward the backbone of the network, these subnets serve as aggregations of capacity and ranges of QoS. Multiple levels of hierarchy are of course possible.

Hierarchical Routing



The figure here shows the advantage of aggregating routes to take advantage of nearness. For the network on the left, the first table is the full table for router 1A, while the second table give the routes for the routers local to 1A and then to the aggregate subnets. It is clear how much shorter its table is.

Section 5.2.6 of the textbook mistakes this for hierarchical routing. No, this is **addressing**. Addresses are location-dependent and route-independent. This is simply making the address location dependent. Let's say I am sitting Foxboro, Massachusetts, and I address a letter by writing 526 E. Main St, Springfield, IL. I put it in the mail. At the post office, when they sort the mail, do they compute a route to 526 E. Main St, Springfield, IL, and then say, "ah, my next hop is to put this in a bag going west"? Of course not. They look at the address, see "IL" and know it goes in a bag going west. They are using the location properties of the address and what else it is near. That letter might go through sorting again in Boston, Albany, Cleveland, etc. They would all do the same thing. When it gets to Chicago, they will say, "This is Illinois, so (next is?) Springfield. It goes in a bag going south." When it gets to Springfield, the post office will look at it and determine which delivery route covers the 500 block of E. Main. Only the postman making the delivery will look at 526 to figure out which house.

But the important property you expect to have is that the building next door is either 524 or 528 E. Main, not 4385 W. Pine St. It is the "nearness property" that is important, not the hierarchy. The hierarchy is merely an artifact of serializing "nearness."[1]

The important thing here is to create an address space with a topological structure that is an abstraction of the graph of the layer, i.e., that has a useful "nearness" property. (Note: "topological" is used here in the original mathematical sense of the word. Not as it is abused by most network experts to inflate their egos by using it to be synonymous for "graph.") The study of the topological structure of addresses is beyond the scope of the course, but it can be very important for making routing much more efficient.

Is the network hierarchical? Maybe, maybe not. It is all about how nodes are grouped with respect to nearness that the routes to them are common.

For hierarchical networks, the graph of the network is a tree. A strict hierarchy often leads to inefficiencies in path length. The question that comes up is, how many levels is optimal for router table size? It turns out that for a network with N nodes, the optimal table size is (ln N). That yields router tables requiring (e ln N) entries per router, where N is the number of routers per subnet. However, there are other factors, as noted above, that determine the nature of the subnets. This can be further modified to add short cuts, which, of course, make the graph no longer a tree but avoid traffic going up the tree to a common node and then back down again. These are easily detected and interpreted if addresses are assigned wisely. The fit is best if the network is roughly hierarchical.

---

1. This can be a real-world problem. In the UK, it had been reasonably common outside of major cities for homes and building to have names and not addresses. Several years ago, the overhead this caused in sorting the mail to become too great and they had to move to normal street addresses. Flat address spaces don't scale.

Broadcast Routing

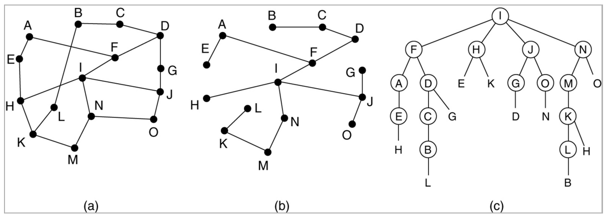
Read

Read Tanenbaum Chapter 5, Section 5.2.7, pp. 384–386.

Sometimes, it may be necessary to send the same PDU to all nodes on a network. The brute force approach would be to simply send N PDUs to N nodes. But that is very wasteful of capacity. Another way would be to have a special PDU type with a list of addresses it is to be delivered to. It works, but it would create very long PCI and there might not much room left for data. Besides it is a special case and special cases are ugly.

A simple approach is to do Reverse Path Forwarding that is more efficient. If a PDU arrived on the line from where it should have come from if it was coming from the source, then forward to everyone else. If it arrived on any other line, then this probably didn't come from the source and is a duplicate—discard it. It isn't perfect, but it's simple.

Broadcast Routing



Reverse path forwarding. (a) A subnet. (b) a sink tree. (c) The tree built by reverse path forwarding.

The figure above tries to illustrate this: a) is the network; b) shows a "sink tree" or spanning tree rooted at I; and c) shows the same thing but more as a tree. The router knows where it sits in the graph of the network. (We just did an algorithm that created a minimal spanning tree rooted at a given node.) The PDU is sent to the Broadcast Address. At each node where the tree branches, the router copies the PDU and sends one down each branch.

If this network is using link-state routing, then each node knows the entire graph of the network; can compute a spanning tree for the source; and knows where it is on the graph, what node to expect PDUs from, and where to forward the PDUs.

Multicast Routing

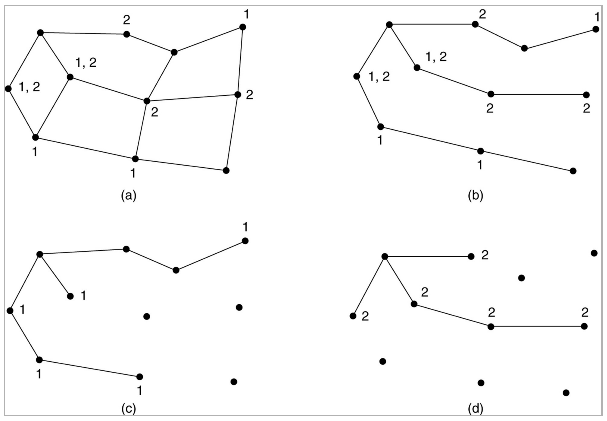
Read

Read Tanenbaum Chapter 5, Section 5.2.8, pp. 386–389.

Broadcast is actually seldom used, i.e., sending a PDU to every node in a network. It is more common to send the same PDU to a subset of nodes, e.g., subscribers to an application of some sort. This is referred to as *multicast*.

The first problem is to define the members of the group that the PDUs are to be delivered to. Tanenbaum starts this discussion with determining the set of routers. In fact, it actually starts by defining the set of applications that constitute the group. It may also include procedures for dynamically joining and leaving the group.

Multicast Routing



(a) A network. (b) A spanning tree for the leftmost router. (c) A multicast tree for group 1. (d) A multicast tree for group 2.

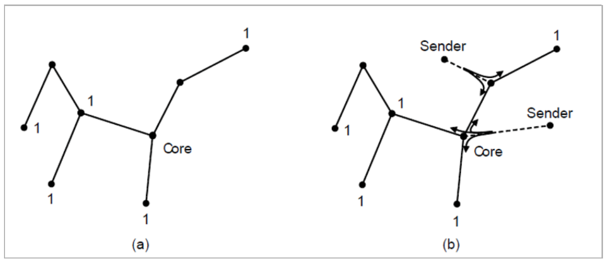
To make multicast efficient, the whole idea is to create a spanning tree of the IPC Processes that constitute the group. A spanning tree is a sub-graph of a graph that includes every node in a set just once with no loops. In the figure above, a) is the network, b) is a spanning tree for the whole network, c) is a spanning tree for Group 1, and d) is a spanning tree for Group 2.

Most of these algorithms work by forming a spanning tree from the sender and then pruning the branches that don't go to a member of the group. Both MOSPF and DVMRP work this way.

Core-Based Trees (Multicast With Multiple Senders)

These are primarily for groups with a single sender and multiple receivers. An alternate approach called *Core-Based Trees* are used when there are multiple senders.

Multicast Routing



(a) Core-based tree for group 1.

(b) Sending to group 1.

The solutions still involve spanning trees, but spanning tree is rooted at the “center-of-gravity” of the group. PDUs are sent in toward the root and back out. The drawback to these schemes is that the root is determined *a priori* before the group is formed, rather than dynamically given the members of the group.

Anycast Routing

Read

Read Tanenbaum Chapter 5, Section 5.2.9, pp. 389–390.

Anycast is in a sense the flip side of multicast. We can characterize a multicast address as the name of a set of addresses, such that when the name is resolved, a PDU is sent to all members of the set. Anycast generalizes that a bit and could be defined as the name of a set of addresses, with a rule such that when the name is resolved, the rule is applied and a PDU is sent to the selected member.

This is to cover the case of a service that wants to send a PDU to the “nearest” member of the set or the “least heavily loaded,” etc. The existing Distance Vector or Link State algorithms can be used to do this, so nothing new is needed.

However, one aspect that should be noted is that all sorts of things have been considered for the rule. One must consider these carefully and ask if this information should be kept by routers. For example, “nearest” might well be something a router could and should know, but consider a number of application services wanting to refer requests to the “least loaded” member of the service. A router knowing the loads at all members of these services or other attributes of these services is not a good idea. That would imply that anycast routing is, in these cases, part of a distributed application.

The Truth About Multicast (and Anycast)

Multicast has little or no benefit to users. It turns out that this observation has been missed by most of the networking community. One often hears laments that multicast failed to catch on in the 90s because the advantage was not made clear enough to the users. But the user’s see no benefit from multicast. The user has an application that *must* send the same data to multiple subscribers/members/customers. The user has no choice.

The user gives the provider the PDU to determine how to get the PDU delivered to all of them. The primary benefit is to the provider. Multicast saves capacity in the network so the network can carry more data. By using multicast, the provider can save capacity to be used by its other customers. Whether or not the provider chooses better margins or to pass the savings on to the user is the provider's choice.

From the provider's point of view, the question is, is it worth the trouble? It requires a parallel routing scheme to generate the spanning tree. Providers have been concerned about unexpected interactions between unicast and multicast routing. The answer until recently has been, not if there is a fiber glut — just send n copies of the PDU. But that is changing as more streaming services are appearing.

The user need never be aware of multicast. The interface interaction appear the same as a unicast interaction. To do this, it is important to distinguish **creating** a multicast group (enrollment) from **joining** a multicast group (allocation). If it is done right, it is straightforward. No current approaches do it right.

Multicast addresses aren't really addresses. As we saw, addresses should be location-dependent and route-independent. It is difficult for the name of a set (the group address) where members may be all over the network to be location-dependent in any meaningful way in general, but there are a few interesting exceptions.

In the Internet, multicast addressing used Ethernet as a model and have defined a multicast address as an ambiguous address. In Ethernet, PDUs travel down the cable passing all stations. When an Ethernet PDU travels down the cable, it is copied off by every station assigned the multicast address. For networks with relaying, this practically requires flooding (which is inherent to Ethernet) or a parallel system, such as a spanning tree, which reflects the set definition. However, this is changing.

As noted above, a multicast "address" is the name of a set such that referencing the set yields all members of the set. Anycast is the flip side of multicast. Hence,

Multicast is $\forall$, while *anycast* is $\exists$ (existential operators).

Anycast is defined by a set, a name for the set, and a rule, such that when the name is referenced, the rule is applied to select one element of the set.

This begs the question: what if the rule returns something between **all and one**? We ignore the problem by generalizing the whole concept so that both are examples of the general case: *Whatevercast*!

Whatevercast is the name of a set of application names or addresses and a rule. Resolving the name causes the rule to be evaluated and returns one or more elements of the set. When a forwarding table is generated, the rule is evaluated relative to this router and the resulting list is the entry in the forwarding table for this name. Unicast is merely a list with one entry.

- Multicast – the rule returns all members.
- Anycast – the rule returns one member.
- But the rule could also return something in between, which could represent multicast or anycast.