MITOCW | 20. Network routing (with failures)

PROFESSOR: So we're going to talk about routing protocols. And today we're going to talk about how routing protocols handle failures. So I'm going to-- I want to bring everybody back onto the same page with respect to where we are in the story. So in terms of routing protocols, if you imagine a network topology like this, and let's say that links have costs associated with them. So I'm just going to make up some numbers here.

We studied two classes of routing protocols. And we looked at these in the absence of any failures. The first is the distance vector protocol, or more generally, vector protocols where, in an advertisement sent from one node to another, each node sends a subset of its routing table. In other words, it sends two columns from its routing table.

Recall that the routing table at every node contains a destination, a route, which is a link, the name of the link, and a cost. And in a distance vector protocol, you send these two columns. You send these tuples for all of the destinations that you know. And that's what's spread out to all of the other nodes.

In contrast-- so this is distance vector. In contrast, in a link state protocol, what you're sending to all of the other nodes is information about all of the links that you have. And in particular, you send information about the neighbor in your topology together with the link cost to that neighbor, whereas here it's the cost of the route to the destination that you send.

In the distance vector protocol, the computation of the routes is distributed. So every node kind of computes and updates its routing table using the Bellman-Ford update step, which essentially updates the route to a destination if you find a better route to the destination. Or if you find that the route to the destination has-- that you currently have "has a changed cost," in which case, you update your routing table.

In a link state protocol, the computation is not distributed. What is distributed is the process of flooding this information of the local links. That's why it's called a link state protocol. And the computation of the routes themselves are centralized. Each node just runs a shortest path computation-- for example, Dijkstra's shortest path computation.

And in a link state protocol, as long as all of the nodes run exactly the same computation-- in other words, they all try to minimize or optimize the same metric-- then you're guaranteed that all of the nodes end up computing the correct routes, and that when you send a packet from one node, it will reach the destination, assuming there's a path in the network.

And in the problem set-- maybe not in the problems. Certainly at the back of the chapter, there are problems on what happens if some nodes run one protocol and another node runs another protocol. For example, you might have-- imagine a routing protocol where one of the nodes is doing minimum cost routes and another node is doing shortest paths-- in terms of the number of hops, not costs.

And if you have that, you might end up in a situation where, in fact, the routing doesn't work correctly. But assuming that all of the nodes are on the same optimization in a link state protocol, they'll all get the correct answer. And similarly, in a distance vector protocol, actually the update rule can do anything, as long as the nodes believe that this route-- and they perform some sort of an update to the routing table entry consistent with the information that they hear, then the routing will work.

Now, what happens when there are failures? And how does it work? So let's say that you run this protocol and this topology. And you give it enough time. Once you hear all of the advertisements and compute the routes of the different nodes, if you look at this node over here, C four destination D would have a route-- let's call this link L0. It will have a route, L0. And the cost of that route would be three.

And similarly at B here, once it converges you'd have-- for destination D, let's imagine this link is called L0, as well, at node B. You might have L0 and the cost would be four, and similarly at the other nodes. Now, let's say what happens is that some sort of a failure occurs. And failures could be one of-- a variety of failures could occur that cause the routing to get screwed up.

So one of the failures-- the easiest form of failure-- is that a packet could be lost. In particular, an advertisement could be lost. The second thing that could happen is that a link could fail. I mean, if it's a wire, maybe a backhoe runs over it, which is actually more common than you think. In fact, you could have undersea cables where sharks bite them and they get destroyed. I mean, lots of things could happen. So links could fail.

And the third thing that could happen, which is more common than you might think, is that entire switches could fail or nodes could fail. Since we actually don't know how to write bug-free software, software bugs may cause things to fail. They may cause things to crash.

In fact, they may cause things to fail in mysterious ways where it looks like it's not failed, but in fact, there's a fault in the software causing you to send bad advertisements. And for this course, we won't worry about this. We'll actually worry about only a class of failures that we think of as fail stop.

What I mean by that is if it fails, it just stops, as opposed to it fails and then it gives you wrong information, which is actually a lot harder to deal with, because it's much better for you to fail and just stop rather than for the node to fail and pretend that it's correct and send you bad information. That's a lot harder to deal with. We'll worry about that later, not in 6.02.

Anyway, these things could happen. So let's concretely assume that you have this topology here. And what happens is this link fails. Now, if you did nothing and you had that link fail, what would happen? Now, let's look at it node by node. What would happen here is that C would end up with no route to the destination, because what would happen is assuming that C has some way of discovering that this fault has happened, it would have no route to the destination.

But if it doesn't discover the fault had happened, it has this route to the destination. But if it sends packets on that link, it wouldn't reach the other side. Packets would be lost. But assuming that it discovers, has a way of discovering that this link has failed, it has no route to the destination. Now, B has a route to the destination. It has this link. But in fact, the next advertisement from C, if C were to make another advertisement, would tell it that it had no route to the destination.

But if it did nothing, if you did nothing in the protocol, B thinks it has a route to the destination. But in fact, it reaches C and it's actually a dead end because it reaches C and C doesn't know what to do with it. It just drops the packet. The technical term for this is a dead end.

You send packets. You think you're reaching the destination. But it's actually getting dropped. What about S? Does S have a route to the destination? Well, S doesn't have a route to the destination, either, because the right route to the destination would probably have been this link over here because the cost was 2 plus 1 plus 3. And that wouldn't have a route to the destination.

It would have a route, but the route would lead to a dead end at C. And in fact, in this particular example, no node would have a route to the destination in terms of the routing table itself. No node would have a route that actually worked, in that the route wouldn't correspond to an actual working path.

But if you look at the picture, clearly there are other ways to get to the destination. I mean, we designed this topology presumably because we had some sort of redundancy. So if this failed, what you would like to have is for C to use one of the other paths. C might use this path or C might use this path. Or C might-- yeah, those are the two possible paths it could use.

And similarly, S might use-- should use that path, and so forth. So what you want is a routing protocol that converges to the new correct answer, assuming there is a new correct answer. And if there's no new correct answer, it converges to whatever the best possible answer is. So to some destinations, you could have a route, and to some destinations, you don't have a route.

So what you want is, as long as there is a connected path, there is a path between a source and a destination, you would like that source to end up with a route that corresponds to some good working path, and in particular, converges to the new minimum cost working path between the source and the destination. So that's the statement of the problem.

And we're going to solve that problem today for both distance vector and link scale. And interestingly, the idea that we're going to use the same idea in both cases. And the idea is, just like we built a redundant topology by having alternate paths between places, we're just going to repeat advertisements. And we're going to repeat the process of processing these advertisements. It's a very simple idea.

And the general plan-- there are three steps to the plan. The first step in the plan is to periodically check-- every neighbor is responsible for checking the health of-- every node is responsible for checking the health of its neighbors. So that's the step, which we're going to call neighbor aliveness.

And the protocol we're going to use for that is called the hello protocol. It's a very, very simple protocol. I'll describe it in a moment. And we're going to use this idea that every node is responsible for checking whether each of its neighbors is alive. And if it determines that a neighbor is not alive, it assumes the neighbor is dead and removes it from various tables and data structures and so on.

And in fact, this fail stop assumption is pretty crucial for us, because the assumption is that when a failure occurs of a node, that in fact, the node doesn't respond. If a failure occurs of a link, the assumption here, as well, is that the link stops responding. You don't get to send packets or receive packets over that link.

And for now, we're going to assume that every link is bidirectional. So you send packets in both directions. In reality, there are unidirectional network links. And you have to deal with the problem differently. Not going to worry about that.

So there is a protocol called the hello protocol that runs to detect if your neighbor is alive or not. The second step in our answer is to make the advertisements periodic. And the third step is, what do you do when you receive an advertisement? When you receive an advertisement, you collect a bunch of these advertisements that you receive from various neighbors. In the link state protocol, it's these link state advertisements. In the distance vector protocol, it's these distance vector tuple advertisements. And then you run a periodic integration process.

So if you look at it with a timeline, every node asynchronously-- in other words, independent of the other nodes. You don't have to synchronize the clocks. Every node has its own clock. And every node does these two steps periodically. So from time to time, it sends an advertisement.

It just says, in distance vector, just sends these two columns to its neighbors. In the link state advertisement, it just sends out its link state information, and the flooding process works. And then from time to time, there's this integration of these advertisements that happen. Et cetera.

Now the beautiful part of these protocols is that I've shown this picture here with these integrations happening interspersed with the advertisements. That doesn't actually have to be the case. You could do them pretty much arbitrarily as long as you do them periodically. The beautiful part of these protocols is that every node asynchronously running these advertisement steps and these integration steps, as long as they do this periodically, in the end what you get as a property called eventual convergence.

What that means is, assuming you have all sorts of failures and any pattern of packet losses, link failures, and switches, and then you freeze the system and assume that no more failures happen, then what eventual convergence means is that in some finite time, all of the nodes in the network will converge to correct routing state. That is, in these routing protocols, all of the nodes will end up with an answer that's consistent with what you are trying to optimize. For example, minimum cost paths to all the destinations.

Now, proving that under an arbitrary model of when these advertisements and integration steps are all asynchronous and being done at random times is a little involved. And we're not going to attempt that in this course. The notes talk a little bit about how you get eventual convergence when you assume that all the nodes are running very periodic advertisement steps interspersed with integration steps. The proof is really not that important. What's more important is for you to understand the intention behind why it works. So I'll do that by some examples here.

I have to also tell you what the Hello protocol is. I'll get to that. But for now, just assume it's a module that tells you if the neighbor is alive or not. So is this plan clear to everybody? It's just this same idea, except every node's doing this periodically. So in practice, you might do this every 30 seconds or every three minutes or something like that.

Of course, the longer the time between advertisements, the longer it's going to take for the protocol to converge after a failure or a set of failures. And the shorter the time, it takes a quicker amount of time to converge. But you end up doing a lot more work. And moreover, in practice, many failures are transient. So a link may fail for a few seconds and then come back up.

And so it's in practice not that useful to converge very, very quickly or react very, very quickly. It's important to converge quickly once you start the convergence process. But once you-- detecting that a neighbor is alive or dead on the timescale of a few packets is sometimes too fast, because sometimes failures last for very little time and then they go away. And in the meantime, you have done all this work to converge to a new routing state.

And then when the link comes back up, you're going to do more work to come back to the old answer. You may as well have just been a little lazy. And so deciding these times is tricky. And there's no real systematic way of doing it in practice. But the trade-off is usually between how quickly you wish to converge and how much work you're willing to expand in making that convergence happen.

So is the plan clear? It's just this same protocol, except we're going to do this periodically. OK. So the first step is this enable liveness, or the hello protocol. So that protocol is actually very easy. Every node-- you have a set of links coming out of the node and then neighbors at the other end of it. So the problem is that the node needs to decide which of these links is working or not working and which of the neighbors are still there versus not there.

And the way the protocol works is that every node in the system on each of its links-- let's call these nodes ABC. On each of these links sends out-- periodically sends out only to its neighbors a packet called a Hello packet. And the Hello packet usually has a sequence number on it, an incrementing sequence number. The idea is now very, very simple. This may be sent periodically, say, every 10 seconds.

If n finds that a certain number of hello packets-- it hasn't heard from its neighbor, one of its neighbors, in some time-- that is, perhaps three hello packets are missing, or four hello packets are missing in a row-- it just decides that that neighbor is dead. It's a very simple idea. So you send out hello packets periodically and if k successive missing hello packets implies the neighbor from which those packets are missing is dead.

Now, in response, what happens is that all of the routes that this node had that went via that neighbor, via that link, are eliminated from the routing table. And you could do that by either simply removing the entry from the routing table or by keeping the entry in the routing table but making-- for those destinations-- and replacing the cost from whatever the value was to infinity.

And it's probably a better idea to replace it with infinity. I'm not exactly sure why. That's kind of what most people do. I think the reason is that you'd like to know that that destination exists in the network. And then when the later route arrives, you can fix it. But you could just remove it, as well.

The other thing that happens is, if you were in a link state protocol, what you would then do is on the next advertisement of the link state, you would simply eliminate that link and that neighbor altogether. So you would not advertise this link and this neighbor as existing anymore. And when that link state advertisement floods through the network, all of the other nodes through the flooding process would determine that that node has gone away and that link has gone away.

And then when they run Dijkstra's algorithm again and recompute the routes, they will no longer assume that that link exists. And they may find new routes to the destination. So that's what the hello protocol does. And so how you pick k, again, it's the same trade-off. It depends on how quickly you want to converge to a real failure.

And picking this is difficult. For example, if you were on a wireless network where the normal packet loss probability might be 10%, something high, then waiting for a larger number of successive failed packets is a good idea, because just because a packet's lost or two packets are lost doesn't mean the link has failed.

On the other hand, if you were running on a highly, highly reliable link in terms of packet loss-- like, you were running on some dedicated optical link where the packet loss rate is one part in a million-- then a single packet missing or two packets missing would be a good indication that that link has actually failed or that node at the other end of the link has failed, and therefore, k could be small.

So again, it totally depends on the actual system context and the normal packet loss rates, because what you're trying to do is to make sure you react to real failure, not to simply packet loss. There's really no way to tell the difference. There's no way to tell the difference between a link that really has failed versus a link with a high packet loss rate. It's a heuristic.

And in fact, there's really no way to tell between a node that has actually failed and gone away and a node that's just heavily overloaded and is extremely slow in responding. There's no way to tell. So these are all heuristics that you have to work with and try to solve the problem. So sometimes you may get it wrong. Sometimes you may find that a link has-- you may declare a link to have failed when, in fact, it's still fine. But that's life. And you just have to deal with it.

So is the story clear so far as to how we deal with routing and a failure? So we're going to apply that to this picture. And you'll find that the answer will work. Yes?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Right. Like I said, what it-- let me repeat this. What it does is, first of all, it really now assumes that both the node and the link have failed. It doesn't really know. Now, it can definitively assume that the link has failed. The node may still be alive, because it may well be that there is a path like that. And n wants to find that route via A to that destination.

So what it does is really two things. The first thing it does is it may have routes in its routing table going through that link. This link is now considered dead. And therefore, it should remove those routes. And then in subsequent advertisements, it should make sure that the cost to that destination is infinity, which is why you would remove it and replace the cost to be infinity so that you tell the other guys that previously I told you I could get to B with the cost of five, but really now it's infinity.

The second thing that's done in the link state protocol is when you advertise-- you no longer advertise that link. So really, the answer to your question is it assumes that the link has failed. It makes no determination about the node. Yes?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yes. A good protocol-- and this will be tested in your lab eight, in p-set eight, is if a link fence and eventually comes back up, we would like for you to actually find that answer. And this is an important requirement. So if the broadcast-- that's why all the stuff that's done in the background is done periodically. So if a link comes back up, you want to find the correct answer.

Any other questions? OK. So let me apply this idea to this picture here. So what happened here was this thing failed. So C is, at this point in time-- let's assume we're doing distance vector, this protocol here-- so C is going to assume that this link has failed. And therefore, it tells all of the other guys in its next advertisement-- it tells these other guys that it no longer has a route to destination D. And it does that by sending in its next link state advertisement.

Previously, it would have sent-- you know what? Did I tell everybody that you send out the destination and the route in the links-- in the advertisement? I may have done that. I meant the destination and the cost. So maybe I should call this the cost and change that to route. So this is the stuff that's sent in the advertisements. And these two columns are in the routing table.

But anyway, right now here what would be advertised as D at a cost of three, we replace that now with D with the cost of infinity in our advertisements to our neighbors. So that's what C would advertise. And B, when it receives that, would now find that the route that it gets along here had previously a cost of four. But now it says that it's replaced with the cost of infinity. So it will replace-- this routing table entry would go away. And it would replace it with no route and a cost of infinity. And that's what would propagate.

Now, these advertisements are done periodically. So what D is doing, of course, is to send out two advertisements, one this way and one that way. Now, this thing is not going to reach, because this link no longer is alive. But this advertisement works. So when A receives the next distance vector advertisement from D, it now knows that it has a cost-- you know, that link is actually alive and it has a route going there.

Now, this particular example is a little tricky, because what's happening here, of course, is that A previously had two ways of getting to D-- 4 plus 3 this way, or 7 that way. If it were previously using that, then A would also have no route to the destination. And then it would have to wait for this guy to send that route. So when it sent that route, A would now have a valid route to the destination. And in A's next advertisement, it would send that route over to these two guys. So it would send out saying that D is at cost seven.

And it would do the same thing here. The D is at cost 7 to C. Now, C, when it receives the next advertisement that D is at cost 7, compares that route against its current route, which is now infinity, and replaces in its routing table entry D infinity with D, this link here, and a cost of 4 plus 7, which is 11. So it would replace it with D. Let me call this L1, L1, and a cost of 11.

And then on its next advertisement, C would send that out to B according to that advertisement schedule. And similarly here, S, when it receives this from A would, on its next advertisement, after integrating the route to destination D, which would have a cost of 1 plus 7, 8, it would send out an advertisement this way, which would have a cost of 8.

And B, when it receives both of these things, would compare a cost of 8 on this link against a cost of 1 plus 4 plus 7, which is 12. And it would find that 8 is smaller than 12. And therefore, B would use this way of getting to the destination. Does that make sense? Yes?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Well, it's receiving hello packets from all its neighbors. And it's just, if a link is alive and a hello shows up, it processes it. And the moment the first hello shows up, it declares the link to be alive again. Finding that someone is alive is a lot easier than finding that they're dead, at least if they're in networks. It's probably true in life, too.

But it's certainly true of networks because all you have to know-- I mean, assuming there's no malicious nodes, detecting that a node is alive, it takes one packet. Detecting that a node is dead, you're not sure. Maybe the link was down. Maybe it was just a transit failure. Maybe a packet was lost. So it's a lot harder to find that something has crashed than it is to find that something's working.

But yeah, so you keep listening for hello packets. OK, so this is how it converges. Now eventually, of course, because there's some correct working path, eventually it will all converge to the correct answer. If later at some point and then this link comes back up, the same thing occurs, because all of this stuff is being done periodically. And so periodically, these advertisements are going to be sent. C is going to find that there's a better route to go to D via this link as 0.

It advertises D now at a cost of 4. And eventually all of the nodes figure it out. And they converge back to the right answer. OK. And you can see that the link state protocol-- the convergence is actually a little bit easier because again, there-- the nodes are periodically advertising these links. So what's going to happen in a link state protocol is if you take the same picture-- and previously the nodes all had routes. And many of those routes went through that link.

You have to wait for the next link state advertisement, which would tell you after this hello protocol discovers that C discovers that this link has failed, it takes that next link state protocol advertised-- link state advertisement by which all of the nodes through the flooding process discover that this link has failed. And they all run Dijkstra's algorithm again. And they will find the correct new answer, which will take them through paths that bypass this failed link.

Now, the same logic applies in both protocols to when a node fails. If this node were to fail, you can sort of think through-- the node failing is actually equivalent to all of the links coming out of the node failing. So it's a somewhat harder problem in terms of just making sure that you're able to find the routes correctly. But this node failing is really the same as all of the links attached to that node failing.

And in a link state protocol, it'll eventually-- you'll discover that. And all of the nodes will compute routes this way. And similarly in a distance vector protocol that's what happens. Now, so far in this picture, I've assumed that once you have these failures and then you pause, nothing else happens. There's no more failures, no packets that are lost, and so on.

But life's actually not so kind. What will happen in practice is that, first of all, before I get to why this stuff is a lot more complicated, does everyone understand how these things work and how they converge correctly to the right answer after failure and after recovery from failure? Any questions?

OK. So now let me tell you all the ways in which this story goes wrong. The first way the story goes wrong is-- let me do it in the context of a link state protocol with a very, very simple picture. Let's say you have-- I think I have a slide. All right. Let's say you have the picture that I've shown up there, so very, very simple picture.

There's A, B, and D. D is the destination. And this is some path. So let's say that what happens is that normally when there's no failures, the way to go from B to D is via A. So B, A, D. And A goes to D directly. Now, let's assume that this link fails. If that link fails and things work great, what's going to happen is that in the next link state advertisement, A tells B that AD no longer exists.

A knows the correct link state from B. And so it computes its path via B, its route via B. And similarly, B realizes that AD doesn't exist anymore. And it computes an alternate route that way. But let's say what happens is that AD fails. And then in the next link state advertisement that A sends out, that packet is lost. Let's say that A's link state advertisement to B is just lost. Packets could get lost.

Now we have a problem because A knows that this has failed. And therefore, when it computes its Dijkstra's algorithm or shortest path algorithm, it knows that what it wants is a route going like that. But B, on the other hand, doesn't know that link AD has failed, because it didn't see that link state advisement which was lost. So what B does is compute its routing table entry, which is the same as it was before, going through that link over here.

Now you have a problem because when A gets a packet that it now-- a data packet that it wants to send to destination D, previously it sent it this way. But now it knows that link has failed. So it sends the packet to B, because its route for D is via B. Well, B gets that packet and looks it up in its routing table. And B believes that the way to get to D is via A. So it sends it back to A.

Well, A gets that packet and says, oh, that's great. This is a packet for destination D. I look it up in my routing table. It goes via B. And this thing [INAUDIBLE] for pretty much as long as you want. This thing here is the simplest example of a general phenomenon called a routing loop.

So the first thing that can happen when-- during the process of route convergence, various kinds of pathologies and problematic conditions could happen. And one of them is a routing loop. The second thing that could happen-- and I showed you that here-- where during the process of convergence, C does not have a route to this link-- to the destination D. But B thought it had a route going via C. But in fact, C just dropped that packet. That's the second condition that happens. It's a dead end.

So both of these things can happen during the process of convergence. Now, these routing loops are particularly problematic because when you have a routing loop, this is an example of a two-hop routing loop. A goes to B. B goes to A goes to B. So it bounces twice.

But you can have more complicated routing loops. You could have a routing loop with four hops that looks like-- or four nodes involved, as opposed to two nodes, where this is destination D. And this A thinks that you have to go that way. B thinks that you have to go this way. C thinks you have to go this way. And let's call this guy E. He thinks you have to go that way. And this could happen.

So you end up with packets cycling around. Now, these packets cycling around-- you know, there's really no way to-- once you have routing table entries that have somehow converged, until it gets fixed, somehow if they've converged to routes that-- where for B, you have to use this link, and C has to use this link, and so forth, and you get a cycle, what ends up happening is these packets cycle forever. There's really no way to avoid the packets cycling forever.

Now, this is, of course-- eventually this will be fixed. If the routing protocol eventually converges, it will eventually discover that this is wrong and find the correct answer. But during the process of convergence, bad things could happen like this. And that's why we have on packets in packet switch networks a field called the hop limit field. And that's on a data packet.

So you have the source of the packet sets a hop limit-- let's say 32. It just says that I need to get to the destination and I know it shouldn't take more than 32 hops, no matter what happens. And then every node that-- every switch that gets this packet reduces the hop limit by 1. And eventually when the hop limit gets to zero, the packet's discarded. So this is a way to flush packets out of the network.

And usually you use this mechanism to handle the case when you get stuck in a routing loop, you don't want these packets to cycle around forever and ever and ever, because these packets move around the network in milliseconds. And the routing protocols take minutes to converge, or many, many seconds to converge. So that's many, many milliseconds. These packets could remain in the network forever and ever using the bandwidth and no one's getting any use out of it. So you have this hop limit field to flush packets out of the system.

But of course, what we'd like to do is to design protocols with guaranteed no routing loops at all. Unfortunately, it's impossible to do that. But what we can try to do is to reduce and mitigate the effects of routing loops.

Now, I want to go through a few more examples of routing loops in-- this is in the link state protocol. I want to actually now talk about what happens with a distance vector protocol and show you why this basic, simple distance vector protocol, which is the first routing protocol that was invented, has some problems and how we go about fixing it. And eventually I'll talk about how this is all used on the internet today.

So here's how a distance protocol-- a distance vector protocol might get stuck in a weird kind of routing loop. So let's take this example here where you have five nodes and we're all interested in finding routes to destination E. And the general lesson I want to get at here is that a distance vector protocol is extremely simple, but it only works on small networks. And for bigger networks, we want something better. So that's where the story's going.

So let me refresh where we are, all the discussion I had so far. So let's assume that link AC fails in this picture. So what you would like to-- and assume that all link costs are 1. So we don't worry about costs at this point. What you want to have happen is for A to discover that this-- A discovers that's failed. And when the routing converges, you would like A to use this link as its route to destination E.

And the cost would be 1, 1, 1, which would be 3. All right. So when A discovers failure, it sends a cost of E is infinity to its neighbor-- in particular, to B. And then B, of course, has a route to destination E at cost 2. B advertises that to A. And then A says, now I have a route to destination E. And thus, this is an example of a good converging routing protocol. Everything's good.

Now, let's assume I complicate the picture. Let's assume that link BD also fails. So now what's the correct answer? Well, these two guys have a route to E. But the network has become disconnected. So the correct answer, the correct convergent answer here is that A and B both discover and instantiate in the routing tables entries that say that E is at a cost of infinity because there's no path, which means that it's an infinite cost.

So when a packet arrives at B for destination DE, you just drop the packet. But this could happen. So here's an example of how that happens. So let's say that this link fails. B discovers that through the hello protocol. And at this point, B changes its routing table entry so that E is at infinity. B had previously sent information to A saying that B was-- E was at distance 2 or cost 2.

And now it says, well, I told you that E was at cost 2 before. But I'm changing my mind. It's at cost infinity. And A says, OK. My entry for E now has cost infinity and both of them have converged correctly to the right answer.

Now, unfortunately, that's not the only thing that could happen. This was in the lucky situation when B discovered it had failed and immediately sent out its cost to A. But what could happen is a little different. What could happen is that B could discover that D has failed and change its routing table entry to infinity. But before it gets a chance to send out its advertisement to A-- or perhaps it sends out its advertisement with a cost of infinity to A, but it got lost-- in either of those cases, what could happen is A could send out its routing table cost to B for destination E, because that's what's happening periodically, right?

Every node is periodically running this. And the times are all asynchronous. Every node has its own notion of when it should send out its advertisement. So what happens now is A sends out an advertisement to B saying it has a route to destination E at a cost of 3, which is very valid, right? After all, A does have a route in its routing table to E, whose cost is 3.

It so happens it goes through B. But A doesn't yet know that that link BD has failed. Now we're a little bit in trouble because B now believes that its routing table entry for E is at cost of infinity because BD has failed that link. And now it sees an advertisement from A with a better cost.

The route says, wow, this is cool. I now have a path of cost 3 via A to E, which is better than my cost of infinity. And so I'm going to assume that I have an entry to E at a cost of 4. So now you can see that this is actually not a valid route at all.

Now, B actually, on the face of it, has no way of knowing if A is telling it a different route. So it could conceivably be the case that A has a different route going that way whose cost is 3. And A can legitimately have a cost of 3 to E that it could be telling B about. But in this protocol, there's no way for B to distinguish that case from the case where A is just repeating to B a route that it received via B. But it's just telling it that it has a route via B.

Now, B, therefore, says that it has a cost of 4 to E. And it sends that to A. And A says, whoa. Previously, the cost was 3-- 2 from B. And now B is telling me that the cost is 4, which means I need to make my cost equal to 4 plus 1, which is 5. And I'm going to send that back down.

B says, all right. I've got a cost of 5. Previously, that same thing had a cost of 3. So now I'm going to make my cost 6. And this goes on forever. Now, in the meantime, if there are packets showing up at either A or B for destination E, they're just going to go bouncing between these two guys. This is a routing loop.

Now, when does this stop? When do these guys stop sending these incrementing costs? Sorry?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. You need a value of infinity. You need to say that, at some point, they're going to reach infinity and we're going to stop. So in other words, for this protocol as it's presented to converge in a legitimate amount-- reasonable amount of time, your value of infinity should be small.

So this thing has a colorful name. It's called counting to infinity. Now, in reality, in any network, the cost of infinity cannot be smaller than the minimum-- the maximum minimum cost path. If you have a minimum cost path that has a cost of 75, for whatever reason, infinity had better be bigger than 75. Right?

So what it means is that you have a problem with this protocol. It works great on small networks. But it only works on small networks. And the reason for that is that it needs the value of infinity that's not very big. So this is why distance vector protocols are only used for really simple, small networks. And the moment the network becomes a certain size or when you want costs that are large values, you really can't use this protocol.

So how do you fix this problem? Any ideas on how to solve this problem? Clearly the internet is pretty big. We're not counting to infinity throughout the whole internet. Or at least, we don't think we are. So how would you fix this problem? Any ideas? Yes?

AUDIENCE: [INAUDIBLE].

PROFESSOR: So we do have a hop limit on packets. So all these packets might have a hop limit. So the packets don't remain in the network for a long time. But that doesn't solve the problem of the routing protocol to converge takes a time which is this counting to infinity problem. So you want a better solution in some way. Yes?

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK. So that's one good idea, which is-- in fact, that's how they started trying to solve this problem, which is, if you have a route to a destination coming from a neighbor, don't send back the same route to them. In other words, in this case, A's route to E was via B. So A should not advertise a route to E-- a route for destination E back to B.

If you do that, there's a name for it. It's called split horizon and the notes describe how this protocol works. Or you could do even better. A could advertise to B that its route to destination E has a cost of infinity, forcing B to definitely not use that route, no matter what happens, because A perceived that route via B. So A should tell B that the cost of that route is infinity because under no circumstances does A want B to use the same route that it received. So you could do that.

The trouble with that is it doesn't-- it solves these two-hop loop problems, but it doesn't solve four-hop loop problems. So you could have a situation where this link fails and C discovers that. But before C sends out its update, B sends out its route to C. And so C thinks it can use B.

In the meantime, B thinks it has a route via A. So you might end up with packets cycling around in longer hop loops. So that idea that you had doesn't actually solve the more general problem. So any other modification or idea that can solve the problem?

So one thing you could do is something called path vector, which is what you could do is every node, rather than just sending the cost, it could send the entire route, that is-- sorry. It could send the entire path. It could send the list of nodes that corresponds to that particular route in its routing table advertisement. So I'll show that with a picture here.

So E could send out not just its destination and a cost, which previously, it would say that to come to E, E would say the cost is zero. But it could now say the cost is zero and the path is E. And then each of these other guys, C and D, could send out their own advertisements saying the cost is 2 or whatever-- the cost is 1. But they could also say that the path is the DE.

So D says that my path to get to E is DE in its advertisement. And B here, when it receives that, could send out its own path vector, which is the list of nodes or the list of switches that corresponds to an actual path that's advertised.

And now the rule for how you integrate a route into your routing table entries is very simple. If you see an advertisement with your own identity in that advertisement, then you know that that's just a rumor that you started or you were involved with, so you shouldn't integrate it. So in particular, in this example here, if B, for example, were to see an advertisement from A with a path that was A, B, D, E, then B wouldn't integrate that.

So what would happen in the picture I showed you before is if these two links were to fail, what would happen is that B would have-- if that link failed, B would have sent that BDE over here. And when A advertises that back to B, it would have ABDE show up. And B now sees ABDE. And B finds that its own name is in that vector or in that advertisement and says, I should pay no attention to that.

And as long as you find your own name somewhere in that list of nodes that routing advertisement went through, you know that you shouldn't pay any attention to it because you were involved in creating that advertisement. And so you shouldn't pay attention to it.

This protocol is called path vector. It's used on the internet in something called the Border Gateway Protocol, which runs between autonomous systems. And that's actually what makes the internet essentially converge and not have these routing loops that go between different internet service providers. Any questions, comments so far about any of this stuff?

So let me summarize everything about routing protocols. And we pick this up in recitation with some problems tomorrow. So the last two lectures in recitation, we've spoken about the network layer. And the main problem that's solved by the network layer is how to get packet routing to work. How do you find good paths between different nodes in the network, between different switches in the network?

Now, we've separated out the tasks of routing from forwarding. Forwarding is what happens when a packet arrives at a switch. There's a lookup that happens in a routing table. You take the destination. You look it up in the table, find the link in the routing table, and ship the packet. So that's done-- usually you want it to be done very, very fast. Routing is the process by which the nodes create routing table entries. And that's a very distributed process. It runs amongst all of the other-- all of the switches in the network.

We looked at two routing protocols-- link state and distance vector. In distance vector, the computation is distributed with these Bellman-Ford update steps. And the distance vector protocol is very beautiful in that it's very, very simple. It works for small networks. But to make the ideal work for bigger networks, you have to enhance the distance with the actual path. And if you enhance it with the path, you actually avoid a lot of these routing loops that show up. You can't eliminate it. But you can mitigate the effect of it.

In the link state protocol, there's actually more work that's done. There's a lot more information that's flooded between nodes. But the protocol converges quicker than these distance vector and path vector protocols usually. Link state protocol, you flood this neighbor information. You consume more bandwidth. There's a lot more bandwidth that's used in the network in flooding it. And the computation is centralized. You run Dijkstra's shortest path.

So what the internet does in general-- I'll pick up on this two lecture-- or three lectures from now, when I talk about how the internet really works and applies the concepts we've studied. What you'll find is that networks like MIT's network will run a protocol like-- a link state like protocol to achieve connectivity between nodes inside MIT.

And then routers at the edge of MIT connecting to other internet service providers run a path vector protocol, like BGP. And all of these things work together and they work because ultimately, all of the switches create these routing table entries that have a mapping between destinations and routes or links that have to be used. So that's the routing story. We'll pick it up in recitation tomorrow and see you back on Wednesday.