## CS 123 Computer Networks: Route Computation

In earlier lectures, we broke routing up into four pieces. The first was addressing by which we defined the structure of IP addresses and prefixes and NATs. The second piece of the puzzle is neighbor routing by which endnodes talk to routers (e.g, things like ARP) and routers discover router neighbors (using say Hello messages). The third and most complex piece of the puzzle is *route computation*. In this piece of the puzzle, assume addresses have been assigned to every subnet and every router knows its neighbors. The problem is to route between subnets that are connected by some path of routers. Which path should be chosen? When a failure occurs, which new path should be chosen? These are the concerns of route computation. The fourth piece, which we study in the next lecture, concerns *forwarding*: in this piece, routes are already computed, and packets go from a source computer to a destination computer by going from router to router following the *forwarding tables* that have been set up in each router by route computation.

Route computation is different within an enterprise or campus and between campuses. For example, Figure 1 shows a plausible fragment of a network connecting 3 UCSD departments. Each department has an Ethernet on which the department machines reside. The "campus" has a core section of four routers, $R2, R3$ and $R4$ and their connecting links. There are also distribution routers $R1$, $R5$, and $R6$ that connect to the ECE, Bio-Eng, and ME (Mechanical Engineering) departments respectively. The topology for a small company could be very similar, and bigger enterprises may only have a larger core networks and lots more distribution routers. Networks such as Figure 1 are called Enterprise networks and are typically managed by a single entity (in this case UCSD) and do not provide transit service for other enterprises. Thus UCSD is called a domain (a single administrative domain) and is given a domain name like ucsd.edu and a Autonomous System (AS) Number.

On the other, UCSD is connected to an ISP called CENIC which in turn is connected to other ISPs. Thus in going from UCSD to Berkeley, it suffices to go to CENIC and then to Berkeley. However, in going from UCSD to MIT, the packet may have to traverse several ISPs such as MCI, UUNET, and Sprint. Such ISPs provide transit service, and the collection of such ISPs form what we call the Wide Area Network or WAN. Notice that the WAN is comprised of individual ISPs, each of which has their own AS number, and each is managed separately. Thus routing across ISPs is a more messy kettle of fish and consists of routing across separately managed domains each of have different agendas and priorities.

Thus it makes sense to separate out route computation into two flavors: one that is used for routing within a domain (intra-domain routing), which is the comparitively easy task of finding shortest routes between co-operating entities within an enterprise. The classic schemes for intra-domain routing that we will study are distance vector and link state routing. Distance vector is used within small enterprises (especially using a Cisco variant called IGRP) and Link State is used within most ISPs (under the name OSPF or IS-IS).

On the other hand, inter-domain routing is left to policy routing which often does not compute shortest paths (or even symmetrical routes) and settles for routes that comply with each individual

ISPs local policy. Such inter-domain routing is thus called policy routing and the only serious protocol in use today is called BGP-4 (for the Border Gateway Protocol). Every leaf domain such as UCSD will have some routers that speak BGP (so-called BGP speakers) that then communicate with a BGP router(s) at their upstream ISP in order to relay packets outside UCSD.
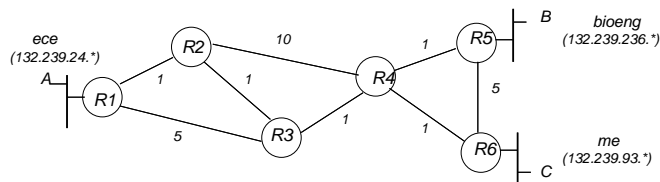


Figure 1: A hypothetical topology connecting 3 departments in UCSD: ECE, ME, and Bio-Engineering

# 1 Intra-domain routing

As we said earlier, there are two principal methods for Route Computation: distance vector and link state. Distance vector is simple and easy to understand and similar to the protocol used for bridges to compute a spanning tree. However, it has a flaw called the count-to-infinity problem that results in slow convergence after failures. Thus most ISPs prefer to use link state routing which fixes this flaw. We will study both protocols.

## 1.1 Distance Vector Routing

Intuitively, distance vector uses ideas very similar to that used by bridges in the spanning tree protocol. In spanning trees, bridges find distances to min ID node (the root) by "gossiping" with their neighbors. Distance vector uses the same idea for updating the distance to all nodes. Bridges keep a single triple *(Root, distance, parent)* that each bridge reports to its neighbors. In distance vector, each router in an enterprise keeps a vector of (ID, distance), where ID is the ID of a subnetwork within the enterprise (e.g., ECE in Figure 1) where a subnetwork is identified by say a prefix like `132.239.24.*`. Note that UCSD itself can be represented by the /16 prefix `132.239.*` but the /16 has been broken up into a 3 /8 prefixes in this picture. Because each enterprise router maintains a *vector* of distances to each subnetwork in the enterprise, this protocol is called distance vector/

Intuitively, each router maintains an estimate of its destination to all the subnetworks in the enterprise. Consider again the topology shown in Figure 1. Assume the managers have set costs for each link as shown. Notice that the link from $R2$ to $R4$ is assigned cost 10 perhaps because it is a slower 100 Mbps "backup" link compared to the other 1 Gbps links between the other routers.

Figure 2 shows a quick snapshot of distance vector operating in the same network from the point of view of one router ($R4$ which is blown up) and from one component of the vector (we choose to focus on the prefix for ECE; the real vector will have 3 components, one for ECE, ME, and ECE).

Intuitively, each neighbor of router $R4$ reports its best estimate for ECE to router $R4$ and $R4$ simply adds its cost to those neighbors and then finds its best estimate which it then reports to all neighbors. Thus, $R4$ gets an estimate of distance 1 from $R2$ and an estimate of distance 2 from $R3$ ($R3$'s shortest path to ECE is via $R2$). It chooses its shortest path as being via $R3$ because the cost via $R2$ is $12 = 2$ (reported cost) $+ 10$ (link cost to $R3$) while the cost via $R3$ is $4 = 3$ (reported cost) $+ 1$ (link cost to $R2$). It then reports this cost to ECE to routers $R5$ and $R6$ and the "gossip" process continues.

Although, we have shown that $R2$ gets information about ECE from $R2$ and $R3$ and reports its distance to $R5$ and $R6$, in reality a router has no way of knowing which routers are "downstream" and which are "upstream". Thus, in reality, $R4$ gets estimates from $R5$ and $R6$ as well (which it will discard because their estimates are for paths through $R4$ itself which will be higher than the ones through $R2$ and $R3$. Similarly, $R2$ will report its estimate of 3 to ECE to $R2$ and $R3$ as well though in the present picture it "seems useless".
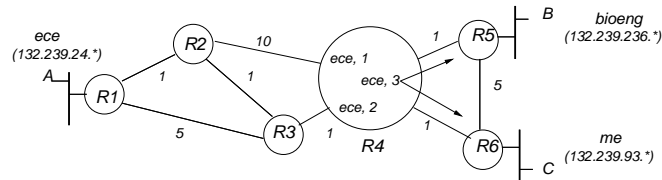


Figure 2: Distance vector operating in the UCSD network of Figure 1. We have blown up router R4 and shown the portion of its distance vector for the ECE prefix only. (In reality, it will have components of its vector for ME, and Bio-Eng as well).

**Distance Vector Databases:** Thus to implement distance vector, a router will have a port database for each neighbor (containing the vector reported by that neighbor) and a central database (containing the vector that is this router's current best estimate to all subnets in the enterprise). This is shown in Figure 3 for a two port router. The enterprise has 3 subnets A, B, C and so the vector has 3 (ID, Cost) components. On Interface P2, the neighboring router has reported a cost of infinity to A, B, and a Cost of 1 to C. On Interface 1, the neighboring router has reported a cost of 3 to A, 3 to B, and 1 to C.
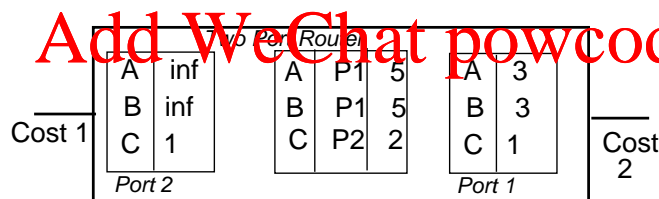


Figure 3: Distance vector databases

To calculate the central table, the router simply calculates for each subnet the minimum (across all neighbors) of the cost reported for that subnet plus the cost (assigned by a manager earlier) to that neighbor. For example, to Subnet C, the best cost is via Interface P2 which gives a cost of $1 + 1 = 2$, as opposed to $1 + 2 = 3$. Notice that the central table has two piece of information for each subnet: the cost and the port number to reach to get to that subnet. For example, the best

3

cost to get to C is 2 and the best way to get to C is via Interface P2.

The simple algorithm (to be modified to deal with failure shortly) is as follows. When a new distance vector is received from a neighbor on an interface, the distance vector is stored in the corresponding port database. The distance vector algorithm is then run to check if the central database needs to be updated (subnet by subnet). If there are changes, the central database is updated and the resulting changed database is sent as a new distance vector (this time with only the first and third columns) on all interfaces of the router (i.e., to all neighbors). This process continues till all the central databases settle down after which a few periodic updates are sent for sanity checking. Of course, the process resumes when a failure occurs.

Finally, the router's *forwarding table* is simply the first two columns of the central table. This is because to forward packets, one only need to know which interface to send a packet; the cost is only useful to compute routes and to send a distance vector to neighbors.

**Distance Vector Dynamics:** Many students and textbooks tend to describe the final form of the distance vector tables which gives the impression of an orderly, and easily understood protocol. In reality, life in the world of protocols is much more messy because of asynchrony (different messages take different times), parallelism (many routers are doing things at the same time) amd failure (messages may get lost, links and routers can crash). One way to appreciate some of the possible scenarios is to watch some possible snapshots of the protocol's state evolving after say a power failure, in which all routers have completely empty tables.

We do so in a series of 7 snapshots, Figure 4 to Figure 10 which show the topology of Figure 1 converging to a stable set of routes for the ECE prefix, starting from a power failure. Initially, all routers except $R1$ have no information about the ECE prefix. $R1$ is *configured* (by some manager) with the ECE prefix being reachable on its leftmost interface at distance 1. Thus in Figure 4 the sequence begins immediately after the power failure by $R1$ sending a distance vector update of its central database information (i.e., ECE is distance 1) to routers $R2$ and $R3$.
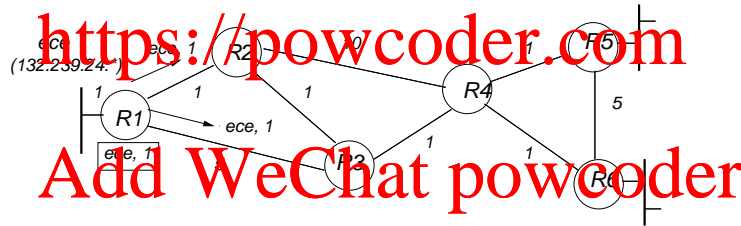


Figure 4: Distance vector dynamics: Snapshot 1 of how distances to the ECE prefix may evolve after a power failure

The sequence continues in Figure 5 with the update to $R3$ arriving while the update from $R1$ to $R2$ is still on the link. Thus, $R2$ updates to a distance of 6 to ECE (because its link to $R1$ has been configured to be of cost 5), and sends its updated value of 6 to $R4$. (It also sends it out to $R2$ and, indeed, to $R1$ but we have ignored that in the picture.) The updated central values are shown boxed.

In Figure 6, the third snapshot, we assume that the update from $R1$ to $R2$ is still in transit but the update to $R4$ arrives, causing $R4$ to update its distance to 7 $(6 + 1)$ and send out its updated value to all neighbors, and we have shown only the updates to $R5$ and $R6$. This is an example
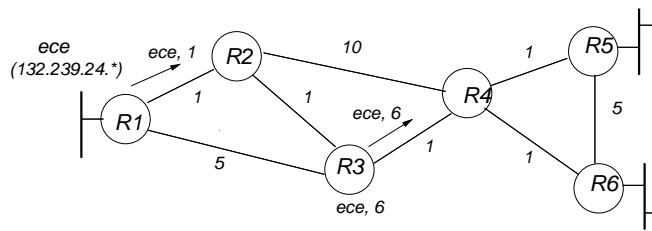
4

Figure 5: Distance vector dynamics: Snapshot 2 of how distances to ece may evolve after a power failure

of *asynchrony* causing unusual behavior. Notice that the best route for $R3$ is via $R1$ and of cost 3. Unfortunately, until the update to $R2$ arrives, $R2$ cannot update $R3$ and the incorrect route is chosen for a while. Such asynchronous behavior is perfectly plausible because of link congestion or a temporary slow down in the route processor at $R1$.
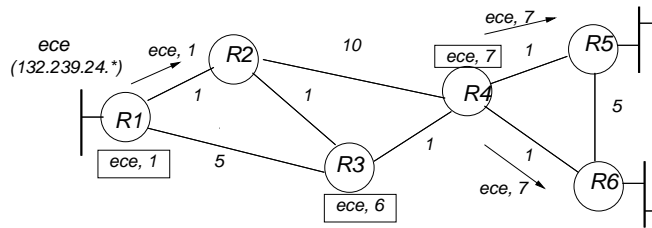


Figure 6: Distance vector dynamics: Snapshot 3 of how distances to ece may evolve after a power failure

In Snapshot 4 shown in Figure 7, the anomalous behavior continues with $R5$ and $R6$ receiving the update from $R4$ and both updating their costs (shown boxed) to 8. However, the behavior begins to correct itself because the laggard update from $R1$ finally arrives at $R2$ causing $R2$ to update to 2 and to send an update of 2 to $R3$.
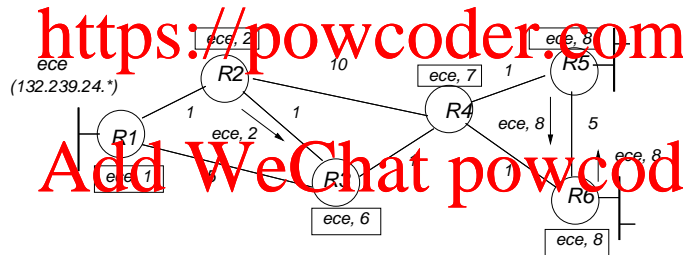


Figure 7: Distance vector dynamics: Snapshot 4 of how distances to ece may evolve after a power failure

In Snapshot 5 shown in Figure 8, the receipt of an update from $R2$ leads $R3$ to update to 3 (the correct value) and send an update to its neighbors.

The sequence finishes in Figure 9 with the updates of $R4$ (to its correct value of 4) and, in Figure 10 with $R5$ and $R6$ reaching the correct values of 5. At this time the route from Mechanical Engineering to ECE is also shown highlighted in bold. Note that this is indeed the shortest path.
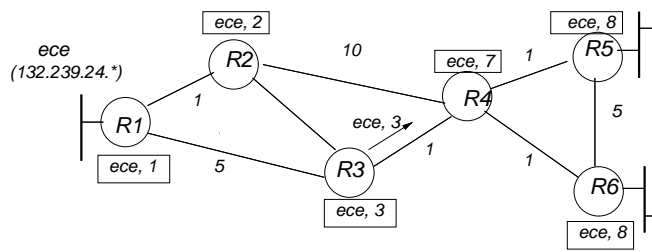
**Link Failures:**

5

ece
(132.239.24.*)

ece, 2
R2   10   ece, 7   ece, 8
R5
R1   1   R4   1
ece, 1   1   ece, 3   1   5
5   R3   1   1   R6
ece, 3   ece, 8

Figure 8: Distance vector dynamics: Snapshot 5 of how distances to ece may evolve after a power failure

ece
(132.239.24.*)

ece, 2
R2   10   ece, 4   ece, 4   ece, 8
R5
R1   1   R4   1
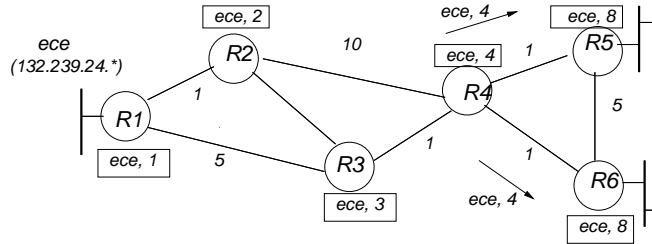ece, 1   5   1   1   5
R3   R6
ece, 3   ece, 4   ece, 8

Figure 9: Distance vector dynamics: Snapshot 6 of how distances to ece may evolve after a power failure

We still have to describe how Distance Vector deals with failure. Let us start with a simple link failure. Suppose the link from $R6$ to $R4$ fails. Neighbor discovery will detect this because $R4$ is supposed to send routing layer hellos to $R6$ periodically, say every second. After say 10 hellos have not been received (in a ten second period), $R6$ decides that the link to $R4$ is dead. When that happens, $R6$ deletes the stored distance vector on that interface (to $R4$ in this example) and recomputes its central database, and the algorithm keeps churning again till it stabilizes.

The effect of a failure of the link from $R4$ to $R6$ is shown in Figure 11. As described earlier, $R6$ fails to receive hellos from $R6$ and so wipes out its port database entry for $R4$ (which was 4). Fortunately, $R6$ has stored the port database entry from $R5$ which says 5. Thus, in the twinkling of an eye $R6$ recalculates its best cost to be 10 (reported by $R5$) plus the configured cost, 5, of the link to $R5$). If $R6$ receives any traffic from a source on the mechanical engineering LAN, it will start routing it to $R5$. Clearly, this was the point of the network manager configuring the slower cost 5 link between $R5$ and $R6$.

**The Count-to-Infinity Problem:**

Let us continue where we left off in Figure 11. We now introduce a second link failure (a double

ece
(132.239.24.*)

ece, 2
R2   10   ece, 4   ece, 5
R5
R1   1   R4   1
ece, 1   1   1   5
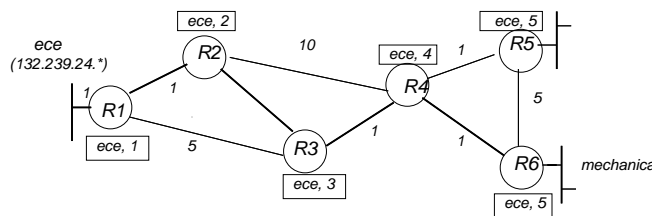5   R3   1   1   R6   mechanical
ece, 3   ece, 5

Figure 10: Distance vector dynamics: Snapshot 7 of how distances to ece may evolve after a power failure. At this point, the distances to ECE have stabilized. The stabilized route from Mechanical Engineering to ECE is now shown in bold.
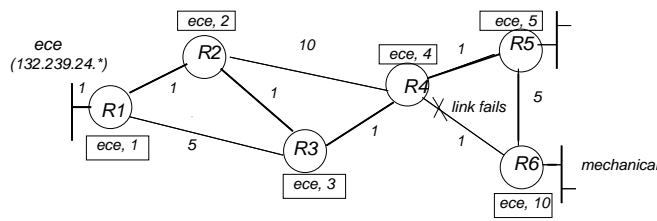
6

Figure 11: Distance vector dynamics on link failure: when the link from $R4$ to $R6$ fails, $R6$ recomputes instantly to go through $R5$ with a new distance of 10. The new stabilized route from Mechanical Engineering to ECE is now shown in bold.

failure like this should be rare, a simpler way to cause it would be to have router $R4$ crash which has the same effect). At this point, the the BioEngineering and Mechanical Engineering Subnets are cut off from ECE, but it will take a while for the routers to figure this out. In particular, when $R5$ finally times out the link to $R4$, it will notice that it has a stored update from $R6$ of distance 10 to ECE. It will thus recompute its distance to ECE to be 15.
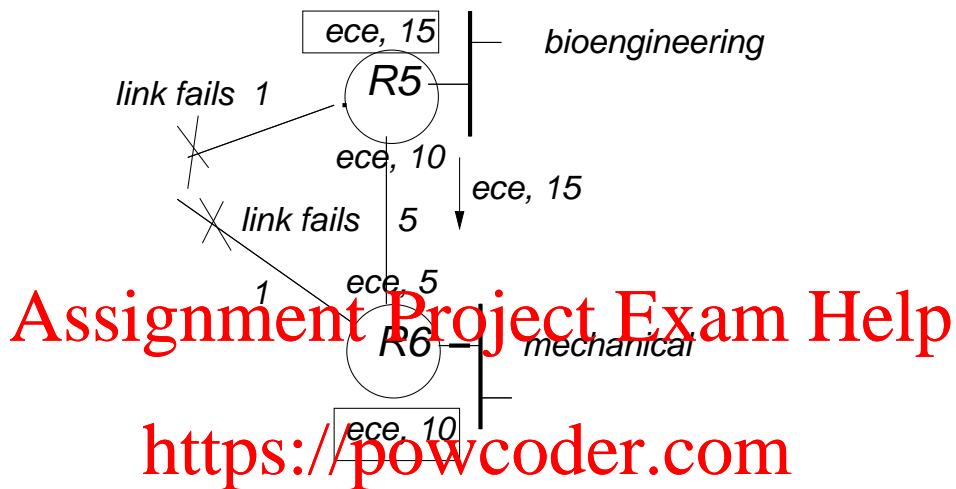


Figure 12: Distance vector dynamics on loss of connectivity: when the link from $R4$ to $R5$ also fails, BioEngineering and Mechanical Engineering are cutoff from the rest of the network. Unfortunately, it takes a while for the routers to realize this using distance vector.

Now, so far every distance calculated by distance vector corresponded to some path in the network. But 15 corresponds to no path[1] in the network. Where is this bizarre distance coming from? It happens because $R5$ is using an update from $R6$ which described an update that was going through $R5$ itself. When you the reader see this, you may say "Wait a minute. Why doesn't $R5$ just not use an update that uses a path through itself?" Unfortunately, using a single distance does not allow $R5$ to know this. For all it knows, $R6$ may have got a new update from $R4$.

The problem continues in Figure 13 because $R5$ sends its bizarre update of 15 to $R6$ which stores (overwriting the old value of 5 which had caused it to generate 10 as its estimate) and updates to 20 (15 + 5).

---

[1]actually it corresponds to no simple path but does correspond to the original shortest path to $R5$ (of distance 5) plus two traversals of the link from $R5$ to $R66$.

Figure 13: Distance vector dynamics on loss of connectivity: when the link from $R4$ to $R5$ also fails, ECE and ME are cutoff from the rest of the network. Unfortunately, it takes a while for the routers to realize this using distance vector.

The counting up (i.e., the gradual increase in the cost to ECE) continues in Figure 14 with $R6$ sending an update of 20 to $R5$ which overwrites and updates to 25. This behavior will continue until the distances count up to "infinity". If one uses a conventional notion of infinity, this will take a long time. Fortunately, distance vector protocols like RIP define infinity to be a small integer. For example, RIP defines a destination to be unreachable when the distance goes over 16. Thus, by the time we reach Figure 14, both routers have finally realized that ECE is unreachable.
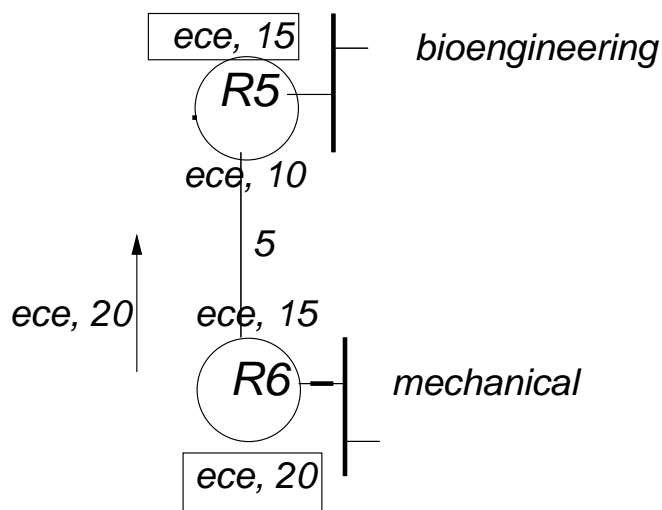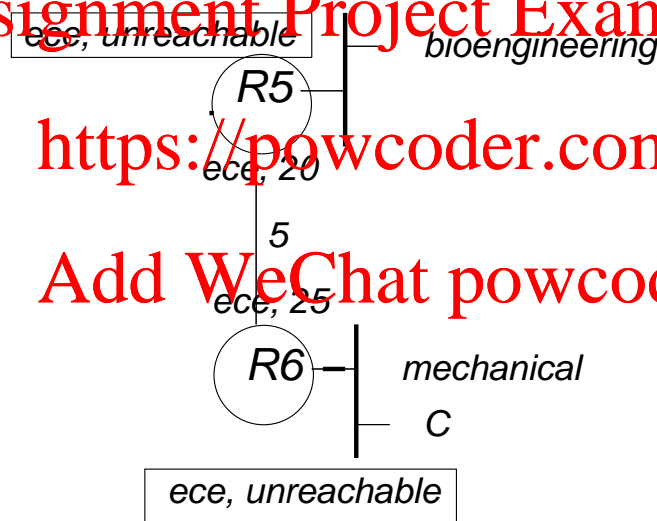
Figure 14: Distance vector dynamics on loss of connectivity: when the link from $R4$ to $R5$ also fails, ECE and ME are cutoff from the rest of the network. When both routers $R5$ and $R6$ finally count up to costs greater than 16, they decide that ece is unreachable. Thus if station $A$ in Mechanical Engineering decides to connect to a web server in ECE, when it sends the TCP SYN Packet to router $R6$, router $R6$ immediately replies with an ICMP "Destination Unreachable" message" which is displayed by your browser.

8

Besides the routing control messages going back and forth (which can be very bad in a LAN with several routers), there is also a major problem with *data packet looping*. Imagine during the snapshots Figure 12 to Figure 14 that some user in Mechanical Engineering tried to connect to a web site in ECE. During that period, $R5$ will get the SYN message (the first TCP message used by a client to connect to a web server) and send it to $R6$. But $R6$ will turn around and send it back to $R5$, who will send it back to $R6$, and so on. This is because each of $R5$ and $R6$ think the bes route to ECE is via each other during the count-up process. This is very serious because the links can be filled with several such data packets looping between routers.

Data packet looping is inevitable in data networks because one router can shift to a route before its neighbors. Till the neighbor also picks up the new routes, the two routers have inconsistent route information and so data packets can loop. However, the data packet looping caused by count-up in distance vector is much serious and prolonged.

One may be tempted to fix this problem by having routers like $R6$ never send an update to $R5$ if its central database was computed based on an update from $R5$. While this "Split Horizon" trick does fix the problem in this example, it does not fix the case of bigger loops. For example, it will not fix the count-up problem for $R1$, $R2$ and $R3$ for the route to say Mechanical Engineering when the two links fail.

This problem can be fixed by having each distance vector update list the Path of routers as well as the distance (a solution used in BGP). However, in the earlier days this was considered unscalable because it could increase the distance vector database by a larger factor (an average path length of 4 would cause the distance vector overhead to grow by a factor of 4). However, ironically enough, BGP uses the same idea. Attempts to fix it with a very scalable solution (distance only, or a few more fields rather than the whole path) have abounded in the literature. The router company Cisco has made an interesting modification to distance vector called IGRP that appears to eliminate data packet looping. UCSD uses IGRP and it seems to work well after failures. IGRP is also Cisco proprietary. Perhaps because of this and the other features of link state, the majority of large ISPs prefer to use a different approach, that we talk about next, called link state routing.

# 2   Link State Routing

The ARPANET is a large national network that is part of the global Internet. (ARPANET has a net number). Classic network in a historic sense. Originally, ARPANET used distance vector. However, failure recovery times were very slow after node failures because of count-to-infinity problem. Also data packets kept looping during this period. New ARPANET moved to link state routing which has quicker response to failures and no count-up problem. Similar design used by OSI Routing and by IP Networks in what is today called OSPF.

**Link State basic idea**

- **Step 1, LSP Generation:** Each node knows the default (or manager settable) cost of its outgoing links. Neighbor discovery is used to compile a list of neighbors that are UP. This information, along with link costs, is placed in a Link State Packet (LSP).

9

- **Step 2, LSP Propagation:** Each source broadcasts its LSP to *all* other nodes using a primitive flooding mechanism called intelligent flooding.

- **Step 3, Dijkstra Shortest Path Computation:** After the LSP propagation process stabilizes. each node has a complete and identical picture of the network graph. Then each node $S$ uses any shortest path algorithm (i.e., Dijkstra's) to compute the next node on the shortest path from $S$ to every other node $D$.

There is a famous joke by Sidney Harris about two mathematicians discussing a proof. The proof has 3 steps: Step 1 and Step are obvious but in Step 2 it says "Then a miracle occurs . . .". The second mathematician is heard remarking, "I think you should be more clear in Step 2". Similarly, in our description of Link State routing, Steps 1 and 3 are obvious. The real miracle is in Step 2. How in the world are we going to send link state packets to every other router before we have computed routes to all routers. Even if we had, we have only studied how to send to one destination, not to multiple destinations. The miracle is answered by a clever protocol called "Intelligent Flooding" We will soon make Step 2, Intelligent Flooding, more clear. But let us start by more completely understanding Step 1.

### Step 1, LSP Generation

Each router computes a list of its adjacent neighboring routers (or prefixes, for cases like $R5$, $R1$ and $R6$) that are currently up (as verified by hellos) and places this information in a link state packet. One can think of the link state packet as describing the *local* neighborhood of a router. For example, in Figure 15, Router $R4$ has a distance 10 to $R2$, a distance 1 to $R3$, a distance 1 to $R5$, and a distance 1 to $R6$. It places this list of neighbors in linear fashion in a Link State Packet (LSP, so called because it describes the state of all its neighboring links) as shown in Figure 15. The packet contains a special routing address and is sent with source $R4$. It then contains a type field which says it is an LSP packet, followed by a list of up neighbors. Note that *every* router in the network computes such a LSP and sends a separate LSP with that router ID has the source address.

Figure 15: Link State packet sent by router $R4$

The more interesting LSP generation occurs on failure. Assuming the link from R4toR6 dies as we described earlier, the new Link State packet sent from $R4$ will not contain the link from $R4$ to $R6$. As usual after the hellos are timed out, $R4$ will delete the link from its LSP, and then attempt to propagate the new LSP from itself to the entire network. Intuitively, the propagation of all post-failure LSPs will produce the correct map of the network after failures: thus Link State

10

is much more responsive to failures. In this particular example, only $R4$ and $R6$ will notice the failure and regenerate and rebroadcast their LSPs. Other routers do not generate new LSPs and so do not need to broadcast their current LSPs again.
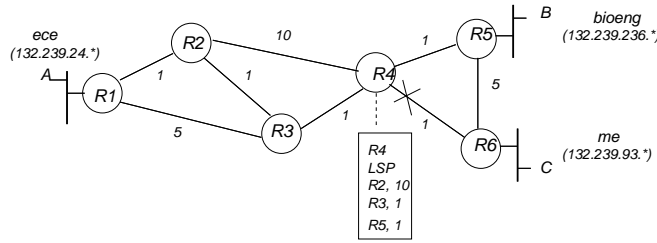


Figure 16: Link state packet generated by $R4$ after failure of link from $R4$ to $R6$

**Step 2, Intelligent Flooding:** How should a router send its most recent LSP to all other routers in the network? The simplest idea is to use *flooding* as bridges do when they do not know where a destination is. In other words, when a router receives an LSP, it should forward it to all other neighbors except for the neighbor who sent the LSP. Flooding certainly ensures that a LSP reaches all routers but it works for bridges only because the bridge topology is a *spanning tree*. In topologies like Figure 1 where there is a loop, flooding can cause LSPs to go around forever in loops. Flooding by itself also does not provide any notion of "recentness": after a failure, we want the newest LSP sent by a router (which clearly spells out all failed links) to go to every other router.

Before we study intelligent flooding (which eliminates loops and also provides a notion of recentness), let us see how ordinary flooding causes loops. Imagine in Figure 1 that $R1$ sends a new LSP $L$. $R1$ sends $L$ to $R2$ and $R3$. $R2$ receives $L$ from $R1$ and sends it to $R3$ amd $R4$. Similarly, $R3$ receives $L$ from $R1$ and floods $L$ to $R2$ and $R4$. $R4$ receives two copies of $L$. When it receives a copy from $R2$ it sends it back to $R3$ as well as to $R5$ and to $R6$. Similarly, when it receives a copy from $R3$ it sends it back to $R2$ as well as to $R5$ and $R6$. Unfortunately, when $R3$ receives the copy of $L$ back from $R4$ (according to the rules of simple flooding) it should send it back to $R2$ and also to $R1$. Similarly, when $R2$ receives the copy of $L$ back from $R4$ it sends a copy to $R3$ and to $R1$. It is easy to see that while all routers get a copy of $L$ the process never terminates with multiple copies of $L$ circulating in every cycle in the topology (for example, $R2$, $R3$ and $R4$ will constantly send copies of $L$ to each other).

Clearly the problem is caused by the fact that when (say) $R3$ receives a copy of $L$ back from $R4$ which it already has, $R3$ should not flood the packet again. Using terminology we used earlier in sliding window protocols, $R3$ should not flood a duplicate of a LSP it has already received. Now duplicate supression can be done by comparing an incoming LSP with the contents of a stored LSP. However, as with reliable data links this is more efficiently done by attaching a *sequence number* to a LSP and comparing the sequence number. More fundamentally (again as with sliding window protocols), a sequence number also gives you a notion of "recentness": when $R1$ wants to send a new LSP, it simply increments its sequence number to persuade the other routers to accept its most recent offering and discard older LSPs stored for $R1$.

11

Thus intelligent flooding modifies ordinary flooding as follows: when a router $R$ receives a LSP $L$ with LSP source $S$ and sequence number $x$, router $R1$ first looks to see if it has a stored LSP $L'$ for source $S$. If $R$ does not have a stored LSP $L'$ from $S$, $R$ does ordinary flooding on $L$. If $R$ does have a stored copy $L'$, it compares the sequence number of $L'$, say $y$ with the sequence number $x$ of the incoming LSP $L$. If $x \leq y$, router $R$ assumes this is a duplicate or an older LSP and discards the LSP (most link state applications also send an ack back to the previous neighbor to stop retransmissions). However, if $x > y$, router $R$ assumes it has "more recent" information from $S$ and does ordinary flooding on $L$ (in other words, sends it on all interfaces except the one it has received it on, and often sends an ack back on the interface on which it received $L$).

To see how this works imagine that $R1$ wants to send a new LSP to the network and $R1$ has never sent a LSP before. Thus no router in Figure 1 has a stored LSP from $R1$. $R1$ sends its first LSP $L$ with sequence number say 1 to $R2$ and $R3$. When $R2$ receives it, it stores $L$ and sends to $R3$ and $R4$. When $R3$ receives $L$ it also stores $L$ and sends to $R2$ and $R4$. So far it is very similar to ordinary flooding. However, when $R2$ receives a copy of $L$ from $R3$, $R2$ discards it because the sequence number (1) of its stored copy from $R1$ matches the sequence number of the incoming packet. Similarly, when $R3$ receives a copy of $L$ from $R2$, it discards the packet for similar reasons.

Next, when $R4$ receives $L$ from say $R2$, it stores $L$ and sends to $R5$ and $R6$ and $R3$. However, when $R4$ receives a second copy of $L$ from $R3$, it discards it as a duplicate because it has sequence number 1. Similarly, when $R3$ receives the copy of $L$ back from $R4$ it also discards it as a duplicate. Thus the indefinite looping caused by ordinary flooding has been stopped using intelligent flooding. Finally, when $R5$ gets a copy from $R4$ it stores it and sends to $R6$. Theoretically, it is possible for $R6$ to receive a copy of $L$ from $R5$ before the copy from $R4$ (which may still be in transit on the link from $R4$ to $R5$.) However, when it gets the second copy from $R4$ that copy is discarded.
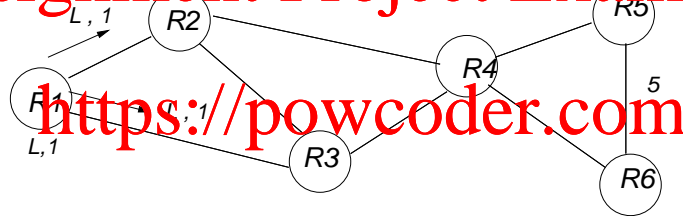
Figure 17: Initial Propagation of link state packet generated by $R1$
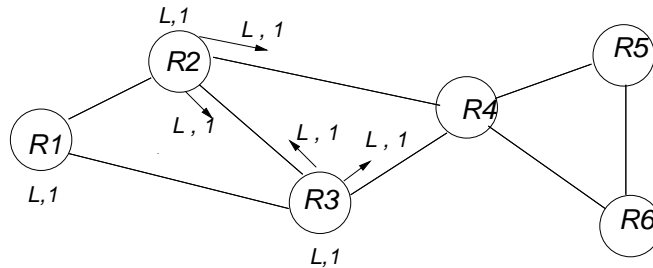


Figure 18: Propagation of link state packet generated by $R1$ reaches $R2$ and $R3$
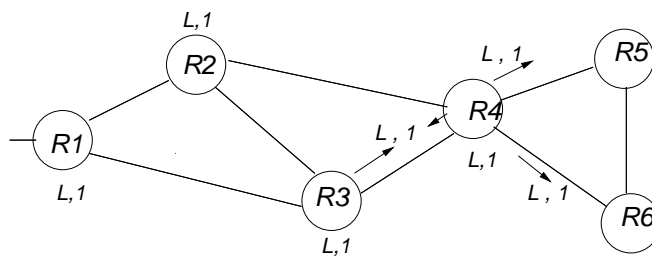
12

Figure 19: Propagation of link state packet generated by $R1$ reaches $R4$ via the copy from $R2$
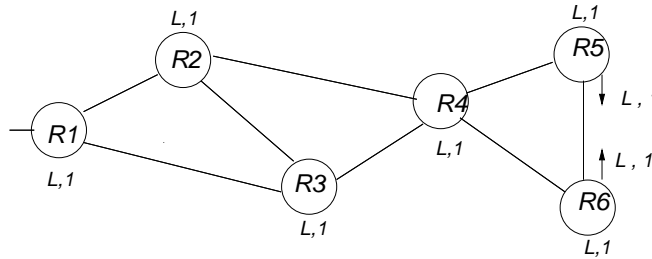


Figure 20: Propagation of link state packet generated by $R1$ reaches $R5$ and $R6$

The process of flooding the first LSP from $R1$ is shown in Figure 17 to Figure 21. Notice that at the end of the process, the LSP transmission has stopped and every router has a copy of the LSP from $R1$. Although we focused on $R1$, every router in Figure 1 uses a similar process to propagate their LSPs to every other router. Thus each router has what is known as a LSP database which contains the latest stored LSP received from every other router in the network. Note that LSP routing is typically done within an enterprise; thus LSPs are not flooded outside an enterprise. (Students sometimes think that each router has to receive an LSP from every router in the Internet which would be very unscalable.) Very large enterprises or ISPs are even broken into smaller areas and LSP flooding only occurs within the area, with another inter-area protocol between areas. So one should expect an LSP database to have no more than a few thousand entries.

There are two more subtleties to intelligent flooding.

- **1, Jumping:** Imagine that router $R1$ has sent 64,000 LSPs and then crashes. Imagine that all other routers have a stored LSP from $R1$ with sequence number 64,000. When $R1$ comes
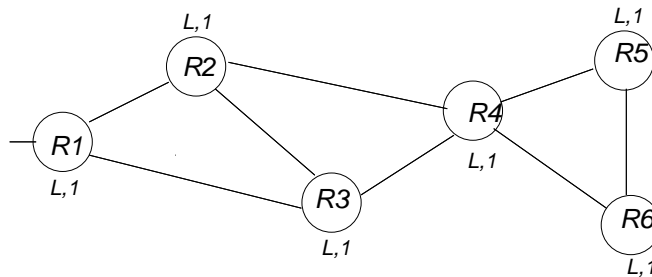


Figure 21: Propagation of link state packet stabilizes

13

up, imagine it has no memory (typically routers have no disk and often no NVRAM). It is logical for $R1$ to come up and try to send a new LSP with sequence number 1. Unfortunately, the rest of the network will simply discard $R1$'s own LSPs as being "too old". If $R1$'s new LSP is different (because it has found some new neighbors after its crash) from its older LSP, this will cause a serious problem.

The solution is twofold. First, we modify the rules of LSP propagation such that when a router $R$ receives an older (than its stored LSP) LSP, instead of merely discarding the LSP, the router sends back its *stored* LSP to the neighbor who sent it the older LSP. (This is intuitively helpful as the neighbor is clearly out of date). Second, when a source like $R1$ receives a copy of its own LSP with a larger sequence number $z$ than its current LSP, $R1$ jumps to $z+1$ and sends its current stored LSP with sequence number $z+1$ to all neighbors. This will result in $R1$'s LSP being believed. In the example above, when $R1$ restarts and sends an LSP with sequence number 1 to $R2$ and $R3$, assume that $R2$ receives it and sends back the LSP with sequence number 64,000 to $R1$. $R1$ jumps to 64,001 and sends back to $R2$ and $R3$. This new LSP now propagates through the network. This is shown in Figure 22 to Figure 25.
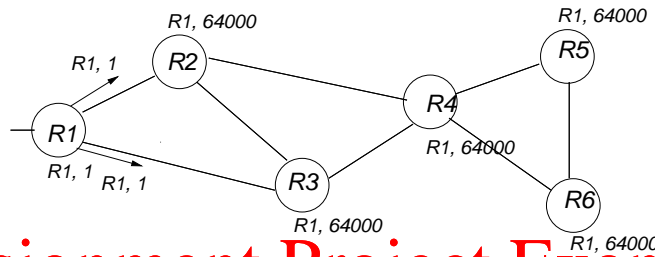
Figure 22: Illustrating jumping: Initially $R1$ comes up after a crash with sequence number 1
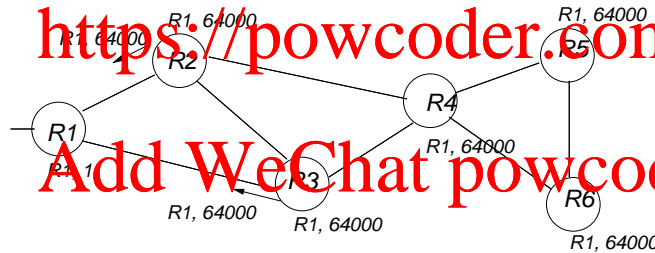
Figure 23: $R2$ and $R3$ reply with their current LSP for $R1$

- **2, Aging:** What happens if a router reaches the maximum of its sequence number space. Should it wrap around like sliding window protocols. There is a sensible way to do this but unfortunately an early bug in one of the first LSP implementations that used wraparound(in the ARPANET) caused a faulty router to place 3 LSPs $L1$, $L2$, and $L3$ in the network, where $L1 < L2 < L3$ but $L3 < L1$. The three updates then kept looping through the network. Thus current generations of LSP protocols use a linear sequence number space. They also use a large space (64 bits) which makes reaching the top of the space very unlikely.
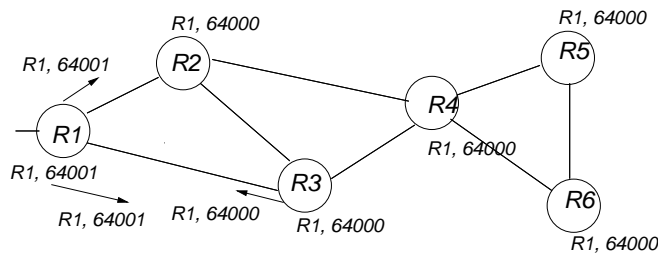
14

Figure 24: Illustrating jumping: $R1$ jumps to $64001$ and floods to $R2$ and $R3$
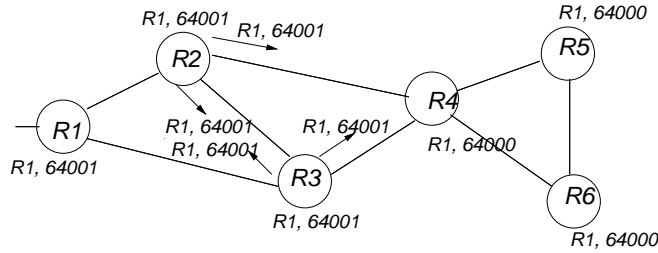


Figure 25: Illustrating jumping: $64001$ takes over $R2$ and $R3$

If at all a bug causes this unlikely event to happen we use two safeguards. First, all LSPs are also aged out. Every LSP carries an age field and is aged out after say half an hour. (This forces all routers to send a new LSP every half hour even if no data has changed). Thus in the worst case, a source router would be disbelieved for half an hour. After all routers time out its older LSPs, it can start again at sequence number 1. Even the half hour delay can be avoided if the router has multiple IP addresses (which all routers do because they need an IP address per interface). Then if the previous LSP used source address $X$, the router can come up and send new LSPs with an alternate IP address $Y$. From the point of view of the network, it is as if a router with source address $X$ crashed and an identical looking router called $Y$ came up in place of $X$. This is pragmatic engineering at its best!

**Step 3: Computing Routes**

Assume for now that LSP propagation has done its job and that a router like $R4$ in Figure 1 has obtained all the LSPs of all the routers $R1$ through $R6$ in the network. This provides $R4$ with a map of the network which it can then use (using any shortest path algorithm) to compute the
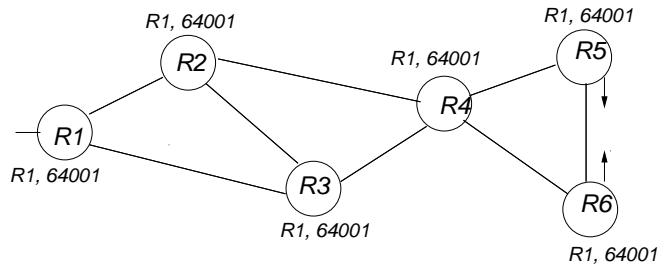


Figure 26: Illustrating jumping: $64001$ takes over the network. $R1$'s new LSP is finally believed by all routers in the enterprise

15

shortest path to all subnets in the network. In practice, the most efficient algorithm is Dijktra's algorithm which you may have studied in an Algorithms class.

Dijsktra's algorithm at any router works schematically as follows. Assume the router $R$ has already computed shortest paths to some set of routers and prefixes and these form a tree, known as the permanent set. All routers and prefixes reachable via some node in the existing tree are marked as the tentative set (shown dashed in the figure) along with their costs. The algorithm starts with only $R$ in the permanent set and all of $R's$ neighbors in the tentative set. At each iteration, the router picks the node (router or prefix) that has the shortest cost in the tentative set and includes it in the permanent set. If the node so picked is a router $S$, $R$ uses $S$'s LSP to see if it can add new nodes to the tentative set that are reachable via $S$. Alternatively, $S$ may allow a node that is already in the tenative set to be reachable at a shorter cost (this is the reason why nodes are marked as tentative; later nodes added to the permanent set may result in shorter costs to such nodes). The iterations continue with a node being added to the permanent set at each iteration; the algorithm terminates when the tentative set is empty.

Let us see how Dijkstra's algorithm would work for the network of Figure 1 when run at router $R4$ (note that all routers run this algorithm to find routes) assuming that $R4$ has all the LSPs in the network.
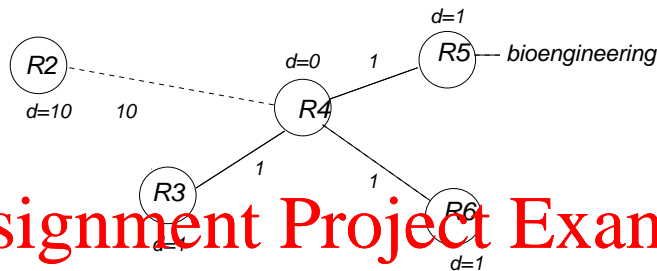


Figure 27: First iteration of Dijktra's algorithm run at router $R4$ in Figure 1 adds $R5$ to the permanent set

In the first iteration, as shown in Figure 27 $R4$ is in the Permanent set at cost 0 (obviously, from itself!) and its neighbors $R5$, $R6$, $R2$ and $R3$ are in the Tentative Set. Notice that this information comes from $R4$'s own Link State packet. In the first iteration, the shortest cost neighbor is one of $R3$, $R5$, and $R6$. In case of a tie, the algorithm can choose arbitrarily, say $R5$. At this point, $R5$'s LSP is examined. This shows another possible route to $R6$, but this is a longer one (of cost 6) and so this is ignored. However, the BioEngineering Prefix (132.239.236.*) is added to the tentative set at distance 2 (we assume the Mechanical Engineering LAN has cost 1) as it is in $R5$'s LSP.. This is shown in Figure 28. Similarly, assuming the second iteration chooses $R6$, the only addition is the Mechanical Engineering prefix (found by consulting $R6$'s LSP) to the tentative set at cost 2 as shown in Figure 29.

By the third iteration, the only possible choice is node $R3$ of cost 1 (it is the shortest cost of those remaining in the tentative set). When $R3$'s LSP is examined, there are two changes made to the Tentative Set. First $R2$ which used to be in the Tentative Set at a cost of 10, is now changed to reflect the cost of 2 via $R3$. Second, $R1$ is added at a cost of 6 (via $R3$). This is shown in Figure 29.
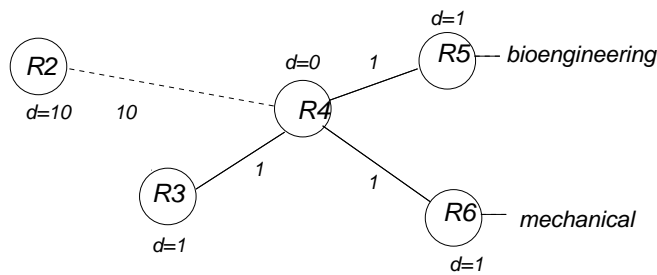
16

Figure 28: Second iteration of Dijktra's algorithm run at router $R4$ in Figure 1 adds $R6$ to the permanent set


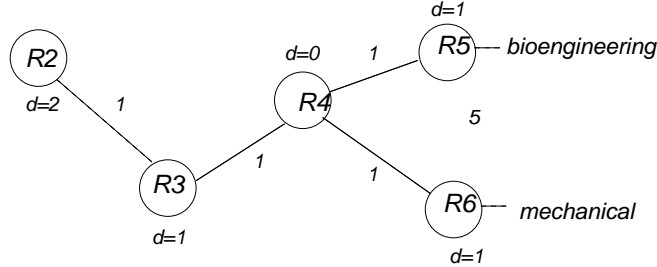
Figure 29: Third iteration of Dijktra's algorithm run at router $R4$ in Figure 1 adds $R3$ to the permanent set

The fourth and fifth iterations pick, say, the BioEngineering and Mechanical prefixes at cost 2. Since these are prefixes and have bno associated LSP, they make no difference to the Tentative Set. In the sixth iteration, $R2$ is picked (it could have been picked before the BioEngineering and Mechanical prefixes). This has the effect of changing the cost of $R2$ to its true cost of 3 in the Tentative Set. A the seventh iteration, $R1$ is picked which brings in the ECE prefix into the Tentative set (Figure 30). Finally, the ECE prefix is added in the last iteration and the complete shortest path tree rooted at $R4$ is shown at the top of Figure 31.
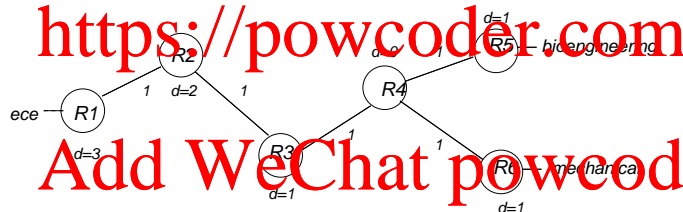


Figure 30: Seventh iteration of Dijktra's algorithm run at router $R4$ in Figure 1 adds $R1$ to the permanent set

While Figure 31 is the final shortest path tree at $R4$ it has too much information to be the forwarding table. For the purpose of the forwarding table, all that $R4$ needs to know is the next interface to forward any packet to, not the complete route. To obtain the forwarding table from Figure 31 assume that the interfaces of router $R4$ are named as in Figure 31. Thus neighbor $R2$ (interface not used, shown dashed) corresponds to interface $P1$, $R3$ to interface $P2$, $R5$ to interface $P3$, and $R6$ to interface $P4$. With this information, it is easy to map the shortest path tree of Figure 31 to the forwarding table shown at the bottom of Figure 31. Notice that we have only placed the prefixes in the forwarding table for simplicity.
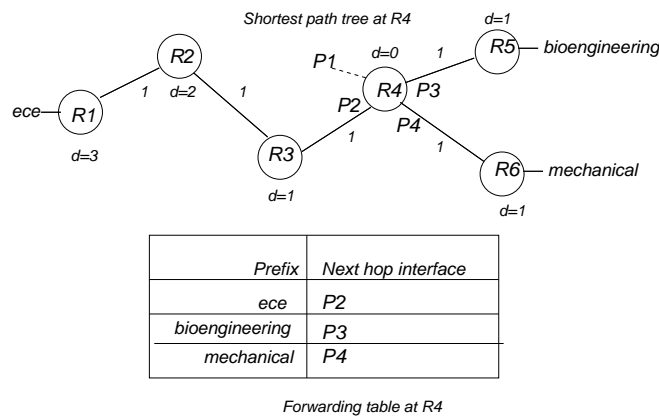
17

Figure 31: Final shortest path tree run at router $R4$ in Figure 1

In most routers, the routing table (top of Figure 31) together with the LSP database are stored in a route processor, which is often a PC like processor with lots of slow memory to store the (large) LSP database. The forwarding table, on the other hand, is ofen stored in fast memory, which is sometimes shared by all router interfaces but is sometimes replicated on every interface.

Theoretically, once LSP propagation stabilizes and every router has the LSP of every other router in the enterprise, a decision process can run at every router to compute shortest path routes from that router to all prefixes in the network. In practice, a router never knows when the set of LSPs have stabilized. Thus whenever a new LSP arrives, a router recomputes its shortest paths. If new LSPs arrive during the route computation, the old route computation is not abandoned; instead the new LSPs are held for the next computation that starts when the previous computation has finished.

Thus, for example in the topology of Figure 1, it is possible (but unlikely) that $R4$ does a Dijsktra computation without getting the LSP from $R3$. It then may decide that the best way to ECE is via $R2$. However, when the LSP from $R3$ arrives a new Dijkstra computation is done at $R4$ to compute the true route via $R3$.

18