Chapter 5 The Network Layer: Control Plane

In this chapter, we'll complete our journey through the network layer by covering the **control-plane** component of the network layer—the *network-wide* logic that controls not only how a datagram is forwarded among routers along an end-to-end path from the source host to the destination host, but also how network-layer components and services are configured and managed. In **Section 5.2**, we'll cover traditional routing algorithms for computing least cost paths in a graph; these algorithms are the basis for two widely deployed Internet routing protocols: OSPF and BGP, that we'll cover in **Sections 5.3** and **5.4**, respectively. As we'll see, OSPF is a routing protocol that operates within a single ISP's network. BGP is a routing protocol that serves to interconnect all of the networks in the Internet; BGP is thus often referred to as the "glue" that holds the Internet together. Traditionally, control-plane routing protocols have been implemented together with data-plane forwarding functions, monolithically, within a router. As we learned in the introduction to **Chapter 4**, software-defined networking (SDN) makes a clear separation between the data and control planes, implementing control-plane functions in a separate "controller" service that is distinct, and remote, from the forwarding components of the routers it controls. We'll cover SDN controllers in **Section 5.5**.

In **Sections 5.6** and **5.7** we'll cover some of the nuts and bolts of managing an IP network: ICMP (the Internet Control Message Protocol) and SNMP (the Simple Network Management Protocol).

5.1 Introduction

Let's quickly set the context for our study of the network control plane by recalling **Figures 4.2** and **4.3**. There, we saw that the forwarding table (in the case of destination-based forwarding) and the flow table (in the case of generalized forwarding) were the principal elements that linked the network layer's data and control planes. We learned that these tables specify the local data-plane forwarding behavior of a router. We saw that in the case of generalized forwarding, the actions taken (**Section 4.4.2**) could include not only forwarding a packet to a router's output port, but also dropping a packet, replicating a packet, and/or rewriting layer 2, 3 or 4 packet-header fields.

In this chapter, we'll study how those forwarding and flow tables are computed, maintained and installed. In our introduction to the network layer in **Section 4.1**, we learned that there are two possible approaches for doing so.

- **Per-router control. Figure 5.1** illustrates the case where a routing algorithm runs in each and every router; both a forwarding and a routing function are contained

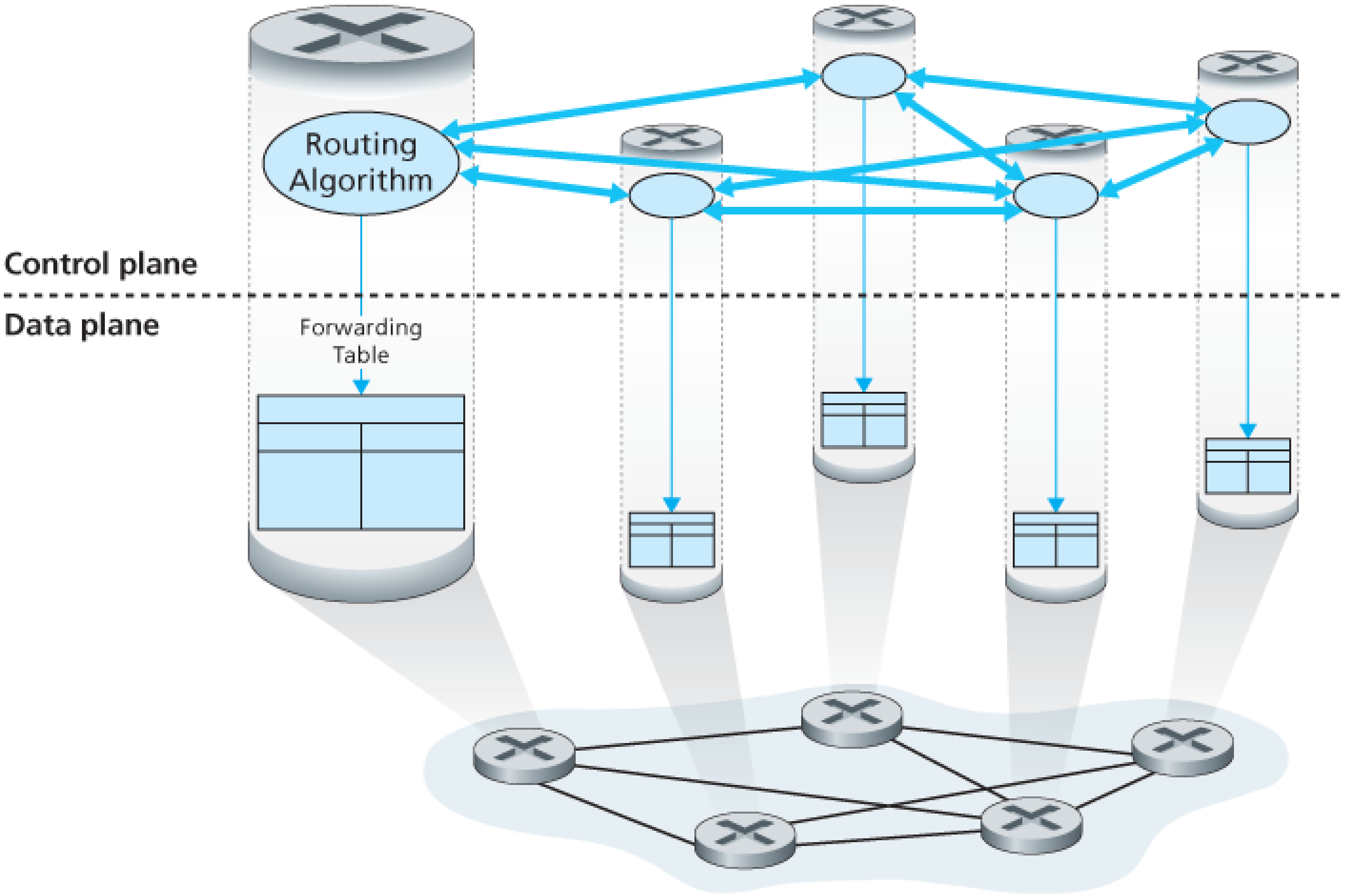


Figure 5.1 Per-router control: Individual routing algorithm components interact in the control plane

within each router. Each router has a routing component that communicates with the routing components in other routers to compute the values for its forwarding table. This per-router control approach has been used in the Internet for decades. The OSPF and BGP protocols that we'll study in **Sections 5.3** and **5.4** are based on this per-router approach to control.

- **Logically centralized control. Figure 5.2** illustrates the case in which a logically centralized controller computes and distributes the forwarding tables to be used by each and every router. As we saw in **Section 4.4**, the generalized match-plus-action abstraction allows the router to perform traditional IP forwarding as well as a rich set of other functions (load sharing, firewalling, and NAT) that had been previously implemented in separate middleboxes.

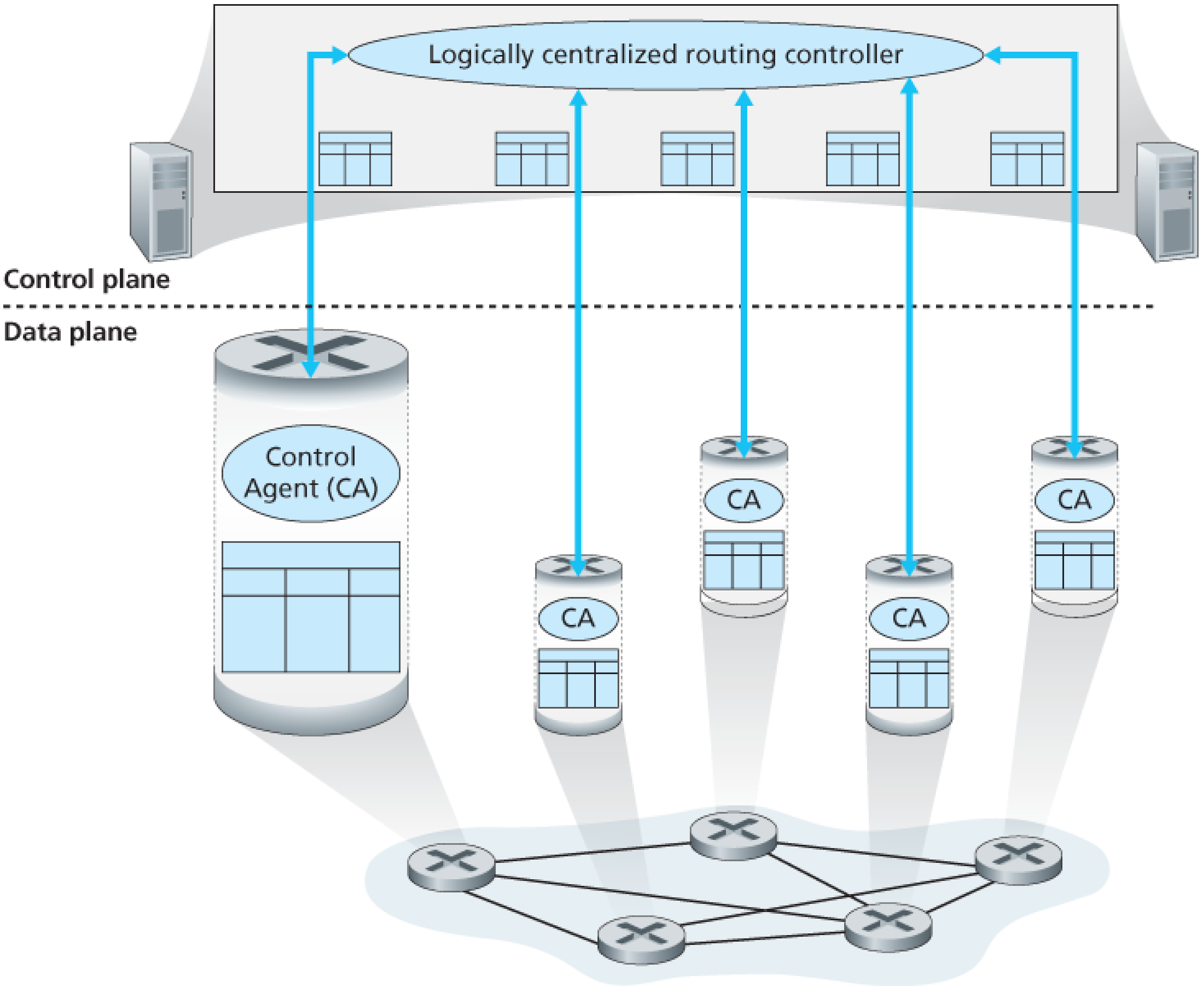


Figure 5.2 Logically centralized control: A distinct, typically remote, controller interacts with local control agents (CAs)

The controller interacts with a control agent (CA) in each of the routers via a well-defined protocol to configure and manage that router's flow table. Typically, the CA has minimum functionality; its job is to communicate with the controller, and to do as the controller commands. Unlike the routing algorithms in **Figure 5.1**, the CAs do not directly interact with each other nor do they actively take part in computing the forwarding table. This is a key distinction between per-router control and logically centralized control.

By “logically centralized” control **[Levin 2012]** we mean that the routing control service is accessed as if it were a single central service point, even though the service is likely to be implemented via multiple servers for fault-tolerance, and performance scalability reasons. As we will see in **Section 5.5**, SDN adopts this notion of a logically centralized controller—an approach that is finding increased use in production deployments. Google uses SDN to control the routers in its internal B4 global wide-area network that interconnects its data centers **[Jain 2013]**. SWAN **[Hong 2013]**, from Microsoft Research, uses a logically centralized controller to manage routing and forwarding between a wide area network and a data center network. China Telecom and China Unicom are using SDN both within data centers and between data centers **[Li 2015]**. AT&T has noted **[AT&T 2013]** that it “supports many SDN capabilities and independently defined, proprietary mechanisms that fall under the SDN architectural framework.”

5.2 Routing Algorithms

In this section we'll study **routing algorithms**, whose goal is to determine good paths (equivalently, routes), from senders to receivers, through the network of routers. Typically, a "good" path is one that has the least cost. We'll see that in practice, however, real-world concerns such as policy issues (for example, a rule such as "router *x*, belonging to organization *Y*, should not forward any packets originating from the network owned by organization *Z* ") also come into play. We note that whether the network control plane adopts a per-router control approach or a logically centralized approach, there must always be a well-defined sequence of routers that a packet will cross in traveling from sending to receiving host. Thus, the routing algorithms that compute these paths are of fundamental importance, and another candidate for our top-10 list of fundamentally important networking concepts.

A graph is used to formulate routing problems. Recall that a **graph** $G=(N, E)$ is a set *N* of nodes and a collection *E* of edges, where each edge is a pair of nodes from *N*. In the context of network-layer routing, the nodes in the graph represent routers—the points at which packet-forwarding decisions are made—and the edges connecting these nodes represent the physical links between these routers. Such a graph abstraction of a computer network is shown in **Figure 5.3**. To view some graphs representing real network maps, see **[Dodge 2016**, **Cheswick 2000]**; for a discussion of how well different graph-based models model the Internet, see **[Zegura 1997**, **Faloutsos 1999**, **Li 2004]**.

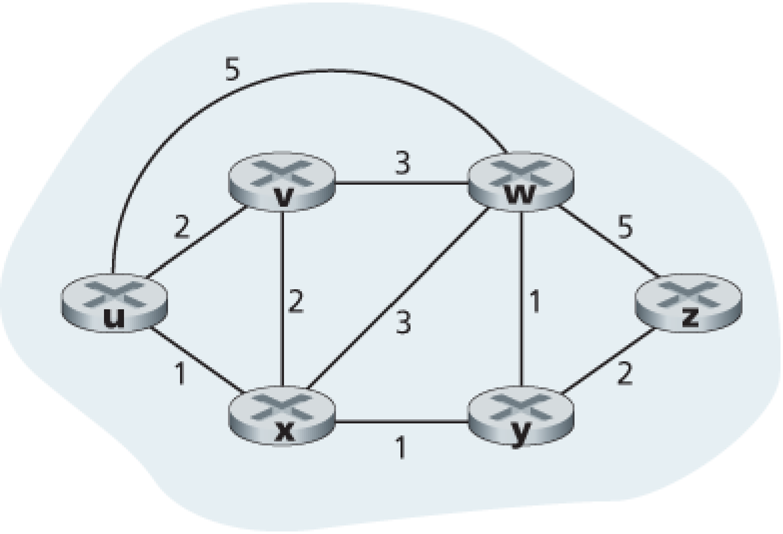


Figure 5.3 Abstract graph model of a computer network

As shown in **Figure 5.3**, an edge also has a value representing its cost. Typically, an edge's cost may reflect the physical length of the corresponding link (for example, a transoceanic link might have a higher cost than a short-haul terrestrial link), the link speed, or the monetary cost associated with a link. For our purposes, we'll simply take the edge costs as a given and won't worry about how they are determined. For any edge (*x*, *y*) in *E*, we denote *c*(*x*, *y*) as the cost of the edge between nodes *x* and *y.* If the pair (*x*, *y*) does not belong to *E*, we set $c(x, y)=\infty$. Also, we'll only consider undirected graphs (i.e., graphs whose edges do not have a direction) in our discussion here, so that edge (*x*, *y*) is the same as edge (*y*, *x*) and that $c(x, y)=c(y, x)$; however, the algorithms we'll study can be easily extended to the case of directed links with a different cost in each direction. Also, a node *y* is said to be a **neighbor** of node *x* if (*x*, *y*) belongs to *E*.

Given that costs are assigned to the various edges in the graph abstraction, a natural goal of a routing algorithm is to identify the least costly paths between sources and destinations. To make this problem more precise, recall that a **path** in a graph $G=(N, E)$ is a sequence of nodes $(x1,x2,\cdots,xp)$ such that each of the pairs $(x1,x2),(x2,x3),\cdots,(xp-1,xp)$ are edges in *E*. The cost of a path $(x1,x2,\cdots, xp)$ is simply the sum of all the edge costs along the path, that is, $c(x1,x2)+c(x2,x3)+\cdots+c(xp-1,xp)$. Given any two nodes *x* and *y*, there are typically many paths between the two nodes, with each path having a cost. One or more of these paths is a **least-cost path**. The least-cost problem is therefore clear: Find a path between the source and destination that has least cost. In **Figure 5.3**, for example, the least-cost path between source node *u* and destination node *w* is (*u*, *x*, *y*, *w*) with a path cost of 3. Note that if all edges in the graph have the same cost, the least-cost path is also the **shortest path** (that is, the path with the smallest number of links between the source and the destination).

As a simple exercise, try finding the least-cost path from node *u* to *z* in **Figure 5.3** and reflect for a moment on how you calculated that path. If you are like most people, you found the path from *u* to *z* by examining **Figure 5.3**, tracing a few routes from *u* to *z*, and somehow convincing yourself that the path you had chosen had the least cost among all possible paths. (Did you check all of the 17 possible paths between *u* and *z*? Probably not!) Such a calculation is an example of a centralized routing algorithm—the routing algorithm was run in one location, your brain, with complete information about the network. Broadly, one way in which we can classify routing algorithms is according to whether they are centralized or decentralized.

- A **centralized routing algorithm** computes the least-cost path between a source and destination using complete, global knowledge about the network. That is, the algorithm takes the connectivity between all nodes and all link costs as inputs. This then requires that the algorithm somehow obtain this information before actually performing the calculation. The calculation itself can be run at one site (e.g., a logically centralized controller as in **Figure 5.2**) or could be replicated in the routing component of each and every router (e.g., as in **Figure 5.1**). The key distinguishing feature here, however, is that the algorithm has complete information about connectivity and link costs. Algorithms with global state information are often referred to as **link-state (LS) algorithms**, since the algorithm must be aware of the cost of each link in the network. We'll study LS algorithms in **Section 5.2.1**.
- In a **decentralized routing algorithm**, the calculation of the least-cost path is carried out in an

iterative, distributed manner by the routers. No node has complete information about the costs of all network links. Instead, each node begins with only the knowledge of the costs of its own directly attached links. Then, through an iterative process of calculation and exchange of information with its neighboring nodes, a node gradually calculates the least-cost path to a destination or set of destinations. The decentralized routing algorithm we'll study below in **Section 5.2.2** is called a distance-vector (DV) algorithm, because each node maintains a vector of estimates of the costs (distances) to all other nodes in the network. Such decentralized algorithms, with interactive message exchange between neighboring routers is perhaps more naturally suited to control planes where the routers interact directly with each other, as in **Figure 5.1**.

A second broad way to classify routing algorithms is according to whether they are static or dynamic. In **static routing algorithms**, routes change very slowly over time, often as a result of human intervention (for example, a human manually editing a link costs). **Dynamic routing algorithms** change the routing paths as the network traffic loads or topology change. A dynamic algorithm can be run either periodically or in direct response to topology or link cost changes. While dynamic algorithms are more responsive to network changes, they are also more susceptible to problems such as routing loops and route oscillation.

A third way to classify routing algorithms is according to whether they are load-sensitive or load-insensitive. In a **load-sensitive algorithm**, link costs vary dynamically to reflect the current level of congestion in the underlying link. If a high cost is associated with a link that is currently congested, a routing algorithm will tend to choose routes around such a congested link. While early ARPAnet routing algorithms were load-sensitive **[McQuillan 1980]**, a number of difficulties were encountered **[Huitema 1998]**. Today's Internet routing algorithms (such as RIP, OSPF, and BGP) are **load-insensitive**, as a link's cost does not explicitly reflect its current (or recent past) level of congestion.

5.2.1 The Link-State (LS) Routing Algorithm

Recall that in a link-state algorithm, the network topology and all link costs are known, that is, available as input to the LS algorithm. In practice this is accomplished by having each node broadcast link-state packets to *all* other nodes in the network, with each link-state packet containing the identities and costs of its attached links. In practice (for example, with the Internet's OSPF routing protocol, discussed in **Section 5.3**) this is often accomplished by a **link-state broadcast** algorithm **[Perlman 1999]**. The result of the nodes' broadcast is that all nodes have an identical and complete view of the network. Each node can then run the LS algorithm and compute the same set of least-cost paths as every other node.

The link-state routing algorithm we present below is known as *Dijkstra's algorithm*, named after its inventor. A closely related algorithm is Prim's algorithm; see **[Cormen 2001]** for a general discussion of graph algorithms. Dijkstra's algorithm computes the least-cost path from one node (the source, which we will refer to as *u*) to all other nodes in the network. Dijkstra's algorithm is iterative and has the property that after the *k*th iteration of the algorithm, the least-cost paths are known to *k* destination nodes, and among the least-cost paths to all destination nodes, these *k* paths will have the *k* smallest costs. Let us define the following notation:

- *D*(*v*): cost of the least-cost path from the source node to destination *v* as of this iteration of the algorithm.
- *p*(*v*): previous node (neighbor of *v*) along the current least-cost path from the source to *v.*
- *N*′: subset of nodes; *v* is in *N*′ if the least-cost path from the source to *v* is definitively known.

The centralized routing algorithm consists of an initialization step followed by a loop. The number of times the loop is executed is equal to the number of nodes in the network. Upon termination, the algorithm will have calculated the shortest paths from the source node *u* to every other node in the network.

Link-State (LS) Algorithm for Source Node u

```
Initialization:
 N' = {u}
 for all nodes v
   if v is a neighbor of u
     then D(v) = c(u, v)
   else D(v) = ∞

Loop
 find w not in N' such that D(w) is a minimum
 add w to N'
 update D(v) for each neighbor v of w and not in N':
       D(v) = min(D(v), D(w)+ c(w, v) )
  /* new cost to v is either old cost to v or known
   least path cost to w plus cost from w to v */
until N'= N
```

As an example, let’s consider the network in **Figure 5.3** and compute the least-cost paths from *u* to all possible destinations. A tabular summary of the algorithm’s computation is shown in **Table 5.1**, where each line in the table gives the values of the algorithm’s variables at the end of the iteration. Let’s consider the few first steps in detail.

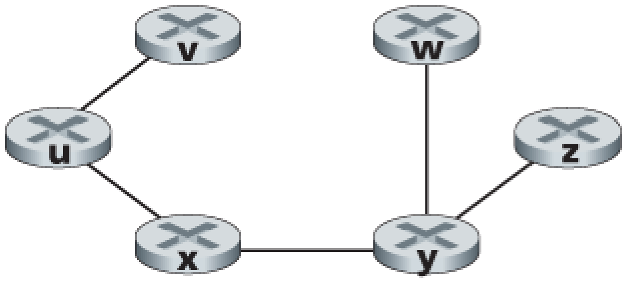
- In the initialization step, the currently known least-cost paths from *u* to its directly attached neighbors, *v, x*, and *w*, are initialized to 2, 1, and 5, respectively. Note in particular that the cost to *w* is set to 5 (even though we will soon see that a lesser-cost path does indeed exist) since this is the cost of the direct (one hop) link from *u* to *w.* The costs to *y* and *z* are set to infinity because they are not directly connected to *u.*

Table 5.1 Running the link-state algorithm on the network in Figure 5.3

step	*N'*	*D (v), p (v)*	*D (w), p (w)*	*D (x), p (x)*	*D (y), p (y)*	*D (z), p (z)*
0	u	2, u	5, u	1,u	∞	∞
1	ux	2, u	4, x		2, x	∞
2	uxy	2, u	3, y			4, y
3	uxyv		3, y			4, y
4	uxyvw					4, y
5	uxyvwz					

- In the first iteration, we look among those nodes not yet added to the set *N′* and find that node with the least cost as of the end of the previous iteration. That node is *x*, with a cost of 1, and thus *x* is added to the set *N′*. Line 12 of the LS algorithm is then performed to update *D*(*v*) for all nodes *v*, yielding the results shown in the second line (Step 1) in **Table 5.1**. The cost of the path to *v* is unchanged. The cost of the path to *w* (which was 5 at the end of the initialization) through node *x* is found to have a cost of 4. Hence this lower-cost path is selected and *w*’s predecessor along the shortest path from *u* is set to *x*. Similarly, the cost to *y* (through *x*) is computed to be 2, and the table is updated accordingly.
- In the second iteration, nodes *v* and *y* are found to have the least-cost paths (2), and we break the tie arbitrarily and add *y* to the set *N′* so that *N′* now contains *u, x*, and *y.* The cost to the remaining nodes not yet in *N′*, that is, nodes *v, w*, and *z*, are updated via line 12 of the LS algorithm, yielding the results shown in the third row in **Table 5.1**.
- And so on . . .

When the LS algorithm terminates, we have, for each node, its predecessor along the least-cost path from the source node. For each predecessor, we also have *its* predecessor, and so in this manner we can construct the entire path from the source to all destinations. The forwarding table in a node, say node *u*, can then be constructed from this information by storing, for each destination, the next-hop node on the least-cost path from *u* to the destination. **Figure 5.4** shows the resulting least-cost paths and forwarding table in *u* for the network in **Figure 5.3**.



Destination	Link
v	(u, v)
w	(u, x)
x	(u, x)
y	(u, x)
z	(u, x)

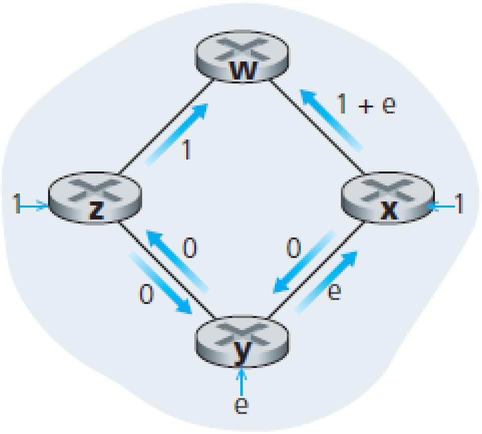
Figure 5.4 Least cost path and forwarding table for node u

What is the computational complexity of this algorithm? That is, given *n* nodes (not counting the source), how much computation must be done in the worst case to find the least-cost paths from the source to all destinations? In the first iteration, we need to search through all *n* nodes to determine the node, *w*, not in *N′* that has the minimum cost. In the second iteration, we need to check $n-1$ nodes to determine the minimum cost; in the third iteration n−2 nodes, and so on. Overall, the total number of nodes we need to search through over all the iterations is n(n+1)/2, and thus we say that the preceding implementation of the LS algorithm has worst-case complexity of order *n* squared: $O(n^2)$. (A more sophisticated implementation of this algorithm, using a data structure known as a heap, can find the minimum in line 9 in logarithmic rather than linear time, thus reducing the complexity.)

Before completing our discussion of the LS algorithm, let us consider a pathology that can arise. **Figure 5.5** shows a simple network topology where link costs are equal to the load carried on the link, for example, reflecting the delay that would be experienced. In this example, link costs are not symmetric; that is, *c*(*u, v*) equals *c*(*v, u*) only if the load carried on both directions on the link (*u, v*) is the same. In this example, node *z* originates a unit of traffic destined for *w*, node *x* also originates a unit of traffic destined for *w*, and node *y* injects an amount of traffic equal to *e*, also destined for *w*. The initial routing is shown in **Figure 5.5(a)** with the link costs corresponding to the amount of traffic carried.

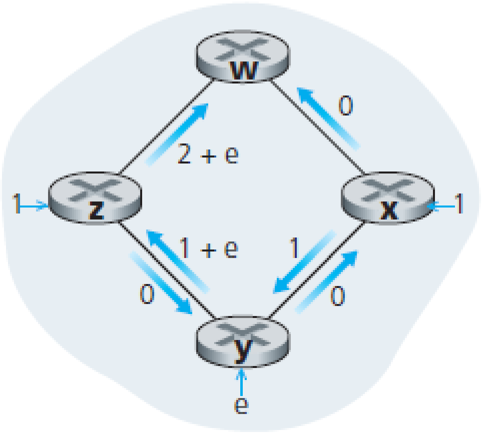
When the LS algorithm is next run, node *y* determines (based on the link costs shown in **Figure 5.5(a)**) that the clockwise path to *w* has a cost of 1, while the counterclockwise path to *w* (which it had been using) has a cost of $1+e$. Hence *y*'s least-cost path to *w* is now clockwise. Similarly, *x* determines that its new least-cost path to *w* is also clockwise, resulting in costs shown in **Figure 5.5(b)**. When the LS algorithm is run next, nodes *x, y*, and *z* all detect a zero-cost path to *w* in the counterclockwise direction, and all route their traffic to the counterclockwise routes. The next time the LS algorithm is run, *x, y*, and *z* all then route their traffic to the clockwise routes.

What can be done to prevent such oscillations (which can occur in any algorithm, not just an LS algorithm, that uses a congestion or delay-based link metric)? One solution would be to mandate that link costs not depend on the amount of traffic

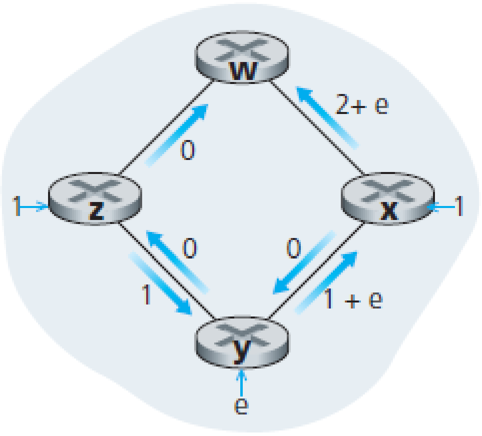


a. Initial routing

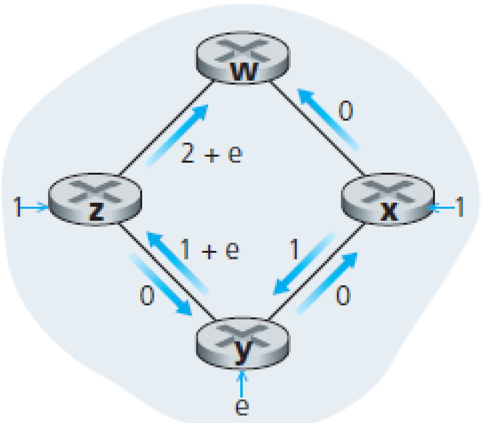
Figure 5.5 Oscillations with congestion-sensitive routing



b. *x*, *y* detect better path to *w*, clockwise



c. *x*, *y*, *z* detect better path to *w*, counterclockwise



d. *x, y, z*, **detect better path to** *w*, **clockwise**

carried—an unacceptable solution since one goal of routing is to avoid highly congested (for example, high-delay) links. Another solution is to ensure that not all routers run the LS algorithm at the same time. This seems a more reasonable solution, since we would hope that even if routers ran the LS algorithm with the same periodicity, the execution instance of the algorithm would not be the same at each node. Interestingly, researchers have found that routers in the Internet can self-synchronize among themselves **[Floyd Synchronization 1994]**. That is, even though they initially execute the algorithm with the same period but at different instants of time, the algorithm execution instance can eventually become, and remain, synchronized at the routers. One way to avoid such self-synchronization is for each router to randomize the time it sends out a link advertisement.

Having studied the LS algorithm, let's consider the other major routing algorithm that is used in practice today—the distance-vector routing algorithm.

5.2.2 The Distance-Vector (DV) Routing Algorithm

Whereas the LS algorithm is an algorithm using global information, the **distance-vector (DV)** algorithm is iterative, asynchronous, and distributed. It is *distributed* in that each node receives some information from one or more of its *directly attached* neighbors, performs a calculation, and then distributes the results of its calculation back to its neighbors. It is *iterative* in that this process continues on until no more information is exchanged between neighbors. (Interestingly, the algorithm is also self-terminating—there is no signal that the computation should stop; it just stops.) The algorithm is *asynchronous* in that it does not require all of the nodes to operate in lockstep with each other. We'll see that an asynchronous, iterative, self-terminating, distributed algorithm is much more interesting and fun than a centralized algorithm!

Before we present the DV algorithm, it will prove beneficial to discuss an important relationship that exists among the costs of the least-cost paths. Let $d_x(y)$ be the cost of the least-cost path from node *x* to node *y*. Then the least costs are related by the celebrated Bellman-Ford equation, namely,

$$d_x(y)=\min_v\{c(x,v)+d_v(y)\}, \tag{5.1}$$

where the *min*$_v$ in the equation is taken over all of *x*'s neighbors. The Bellman-Ford equation is rather intuitive. Indeed, after traveling from *x* to *v*, if we then take the least-cost path from *v* to *y*, the path cost will be $c(x,v)+d_v(y)$. Since we must begin by traveling to some neighbor *v*, the least cost from *x* to *y* is the minimum of $c(x,v)+d_v(y)$ taken over all neighbors *v*.

But for those who might be skeptical about the validity of the equation, let's check it for source node *u* and destination node *z* in **Figure 5.3**. The source node *u* has three neighbors: nodes *v*, *x*, and *w*. By walking along various paths in the graph, it is easy to see that $d_v(z)=5$, $d_x(z)=3$, and $d_w(z)=3$. Plugging these values into **Equation 5.1**, along with the costs $c(u,v)=2$, $c(u,x)=1$, and $c(u,w)=5$, gives $d_u(z)=\min\{2+5,5+3,1+3\}=4$, which is obviously true and which is exactly what the Dijskstra algorithm gave us for the same network. This quick verification should help relieve any skepticism you may have.

The Bellman-Ford equation is not just an intellectual curiosity. It actually has significant practical importance: the solution to the Bellman-Ford equation provides the entries in node *x*'s forwarding table. To see this, let *v** be any neighboring node that achieves the minimum in **Equation 5.1**. Then, if node *x* wants to send a packet to node *y* along a least-cost path, it should first forward the packet to node *v**. Thus, node *x*'s forwarding table would specify node *v** as the next-hop router for the ultimate destination *y*. Another important practical contribution of the Bellman-Ford equation is that it suggests the form of the neighbor-to-neighbor communication that will take place in the DV algorithm.

The basic idea is as follows. Each node *x* begins with *D*$_x$(*y*), an estimate of the cost of the least-cost path from itself to node *y*, for all nodes, *y*, in *N*. Let $\mathbf{D}_x=[D_x(y)\text{: } y \text{ in } N]$ be node *x*'s distance vector, which is the vector of cost estimates from *x* to all other nodes, *y*, in *N.* With the DV algorithm, each node *x* maintains the following routing information:

- For each neighbor *v*, the cost *c*(*x, v*) from *x* to directly attached neighbor, *v*
- Node *x*'s distance vector, that is, $\mathbf{D}_x=[D_x(y)\text{: } y \text{ in } N]$, containing *x*'s estimate of its cost to all destinations, *y*, in *N*
- The distance vectors of each of its neighbors, that is, $\mathbf{D}_v=[D_v(y)\text{: } y \text{ in } N]$ for each neighbor *v* of *x*

In the distributed, asynchronous algorithm, from time to time, each node sends a copy of its distance vector to each of its neighbors. When a node *x* receives a new distance vector from any of its neighbors *w*, it saves *w*'s distance vector, and then uses the Bellman-Ford equation to update its own distance vector as follows:

$$D_x(y)=\min_v\{c(x,v)+D_v(y)\} \quad \text{for each node } y \text{ in } N$$

If node *x*'s distance vector has changed as a result of this update step, node *x* will then send its updated distance vector to each of its neighbors, which can in turn update their own distance vectors. Miraculously enough, as long as all the nodes continue to exchange their distance vectors in an asynchronous fashion, each cost estimate $D_x(y)$ converges to $d_x(y)$, the actual cost of the least-cost path from node *x* to node *y* **[Bertsekas 1991]**!

Distance-Vector (DV) Algorithm

At each node, *x*:

```
1  Initialization:
2    for all destinations y in N:
3       $D_x(y) = c(x, y)$ /* if y is not a neighbor then $c(x, y) = \infty$ */
4    for each neighbor w
5       $D_w(y) = ?$ for all destinations y in N
6    for each neighbor w
7       send distance vector $\mathbf{D}_x = [D_x(y): y \text{ in } N]$ to w
8
9  loop
10    wait  (until I see a link cost change to some neighbor w or
11            until I receive a distance vector from some neighbor w)
12
13    for each y in N:
14        $D_x(y) = \min_v\{c(x, v) + D_v(y)\}$
15
16 if $Dx(y)$ changed for any destination y
17        send distance vector $\mathbf{D}_x = [D_x(y): y \text{ in } N]$ to all neighbors
18
19 forever
```

In the DV algorithm, a node *x* updates its distance-vector estimate when it either sees a cost change in one of its directly attached links or receives a distance-vector update from some neighbor. But to update its own forwarding table for a given destination *y*, what node *x* really needs to know is not the shortest-path distance to *y* but instead the neighboring node $v^*(y)$ that is the next-hop router along the shortest path to *y*. As you might expect, the next-hop router $v^*(y)$ is the neighbor *v* that achieves the minimum in Line 14 of the DV algorithm. (If there are multiple neighbors *v* that achieve the minimum, then $v^*(y)$ can be any of the minimizing neighbors.) Thus, in Lines 13–14, for each destination *y*, node *x* also determines $v^*(y)$ and updates its forwarding table for destination *y*.

Recall that the LS algorithm is a centralized algorithm in the sense that it requires each node to first obtain a complete map of the network before running the Dijkstra algorithm. The DV algorithm is *decentralized* and does not use such global information. Indeed, the only information a node will have is the costs of the links to its directly attached neighbors and information it receives from these neighbors. Each node waits for an update from any neighbor (Lines 10–11), calculates its new distance vector when receiving an update (Line 14), and distributes its new distance vector to its neighbors (Lines 16–17). DV-like algorithms are used in many routing protocols in practice, including the Internet's RIP and BGP, ISO IDRP, Novell IPX, and the original ARPAnet.

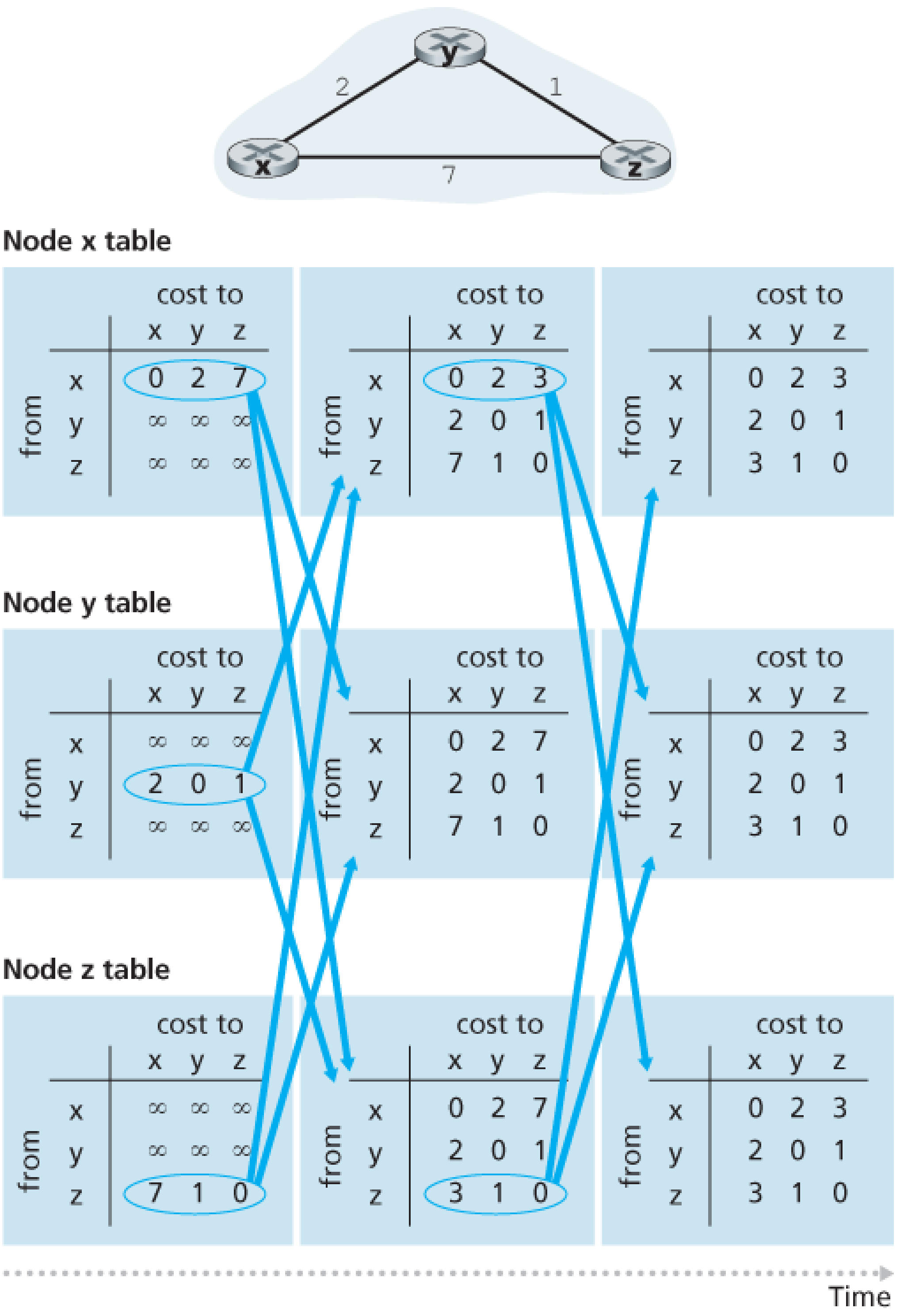
Figure 5.6 illustrates the operation of the DV algorithm for the simple three-node network shown at the top of the figure. The operation of the algorithm is illustrated in a synchronous manner, where all nodes simultaneously receive distance vectors from their neighbors, compute their new distance vectors, and inform their neighbors if their distance vectors have changed. After studying this example, you should convince yourself that the algorithm operates correctly in an asynchronous manner as well, with node computations and update generation/reception occurring at any time.

The leftmost column of the figure displays three initial **routing tables** for each of the three nodes. For example, the table in the upper-left corner is node *x*'s initial routing table. Within a specific routing table, each row is a distance vector— specifically, each node's routing table includes its own distance vector and that of each of its neighbors. Thus, the first row in node *x*'s initial routing table is $D_x=[D_x(x),D_x(y),D_x(z)]=[0,2,7]$. The second and third rows in this table are the most recently received distance vectors from nodes *y* and *z*, respectively. Because at initialization node *x* has not received anything from node *y* or *z*, the entries in the second and third rows are initialized to infinity.

After initialization, each node sends its distance vector to each of its two neighbors. This is illustrated in **Figure 5.6** by the arrows from the first column of tables to the second column of tables. For example, node *x* sends its distance vector $\boldsymbol{D}_x$ = [0, 2, 7] to both nodes *y* and *z*. After receiving the updates, each node recomputes its own distance vector. For example, node *x* computes

$$D_x(x)=0 D_x(y)=\min\{c(x,y)+D_y(y),c(x,z)+D_z(y)\}=\min\{2+0,\ 7+1\}=2 D_x(z)=\min\{c(x,y)+D_y(z),c(x,z)+D_z(z)\}=\min\{2+1,7+0\}=3$$

The second column therefore displays, for each node, the node's new distance vector along with distance vectors just received from its neighbors. Note, for example, that



Node x table

		cost to	
from	x	y	z
x	0	2	7
y	∞	∞	∞
z	∞	∞	∞

		cost to	
from	x	y	z
x	0	2	3
y	2	0	1
z	7	1	0

		cost to	
from	x	y	z
x	0	2	3
y	2	0	1
z	3	1	0

Node y table

		cost to	
from	x	y	z
x	∞	∞	∞
y	2	0	1
z	∞	∞	∞

		cost to	
from	x	y	z
x	0	2	7
y	2	0	1
z	7	1	0

		cost to	
from	x	y	z
x	0	2	3
y	2	0	1
z	3	1	0

Node z table

		cost to	
from	x	y	z
x	∞	∞	∞
y	∞	∞	∞
z	7	1	0

		cost to	
from	x	y	z
x	0	2	7
y	2	0	1
z	3	1	0

		cost to	
from	x	y	z
x	0	2	3
y	2	0	1
z	3	1	0

Figure 5.6 Distance-vector (DV) algorithm in operation

node *x*'s estimate for the least cost to node *z*, $D_x(z)$, has changed from 7 to 3. Also note that for node *x*, neighboring node *y* achieves the minimum in line 14 of the DV algorithm; thus at this stage of the algorithm, we have at node *x* that $v^*(y)=y$ and $v^*(z)=y$.

After the nodes recompute their distance vectors, they again send their updated distance vectors to their neighbors (if there has been a change). This is illustrated in **Figure 5.6** by the arrows from the second column of tables to the third column of tables. Note that only nodes *x* and *z* send updates: node *y*'s distance vector didn't change so node *y* doesn't send an update. After receiving the updates, the nodes then recompute their distance vectors and update their routing tables, which are shown in the third column.

The process of receiving updated distance vectors from neighbors, recomputing routing table entries, and informing neighbors of changed costs of the least-cost path to a destination continues until no update messages are sent. At this point, since no update messages are sent, no further routing table calculations will occur and the algorithm will enter a quiescent state; that is, all nodes will be performing the wait in Lines 10–11 of the DV algorithm. The algorithm remains in the quiescent state until a link cost changes, as discussed next.

Distance-Vector Algorithm: Link-Cost Changes and Link Failure

When a node running the DV algorithm detects a change in the link cost from itself to a neighbor (Lines 10–11), it updates its distance vector (Lines 13–14) and, if there's a change in the cost of the least-cost path, informs its neighbors (Lines 16–17) of its new distance vector. **Figure 5.7(a)** illustrates a scenario where the link cost from *y* to *x* changes from 4 to 1. We focus here only on *y*' and *z*'s distance table entries to destination *x*. The DV algorithm causes the following sequence of events to occur:

- At time t_0, *y* detects the link-cost change (the cost has changed from 4 to 1), updates its distance vector, and informs its neighbors of this change since its distance vector has changed.
- At time t_1, *z* receives the update from *y* and updates its table. It computes a new least cost to *x* (it has decreased from a cost of 5 to a cost of 2) and sends its new distance vector to its neighbors.
- At time t_2, *y* receives *z*'s update and updates its distance table. *y*'s least costs do not change and hence *y* does not send any message to *z*. The algorithm comes to a quiescent state.

Thus, only two iterations are required for the DV algorithm to reach a quiescent state. The good news about the decreased cost between *x* and *y* has propagated quickly through the network.

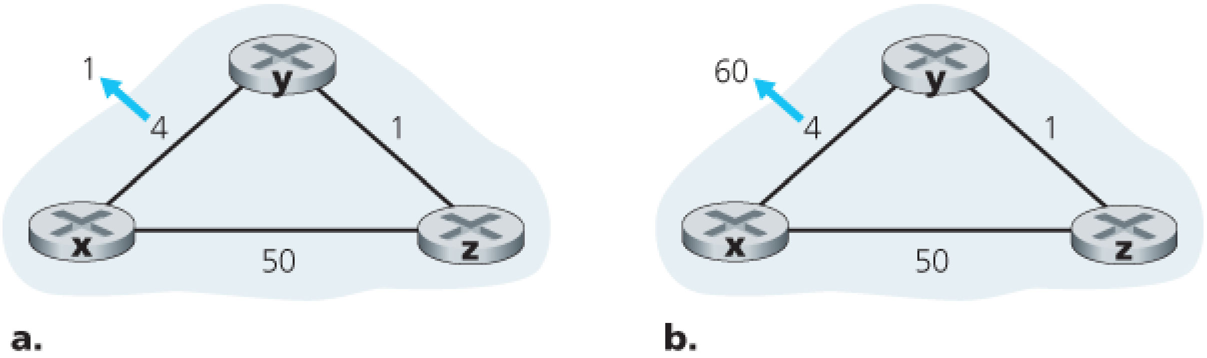


Figure 5.7 Changes in link cost

Let's now consider what can happen when a link cost *increases.* Suppose that the link cost between *x* and *y* increases from 4 to 60, as shown in **Figure 5.7(b)**.

1. Before the link cost changes, $D_y(x)=4$, $D_y(z)=1$, $D_z(y)=1$, and $D_z(x)=5$. At time t_0, *y* detects the link-

cost change (the cost has changed from 4 to 60). *y* computes its new minimum-cost path to *x* to have a cost of

$D_y(x)=\min\{c(y,x)+D_x(x),\ c(y,z)+D_z(x)\}=\min\{60+0,1+5\}=6$

Of course, with our global view of the network, we can see that this new cost via *z* is *wrong.* But the only information node *y* has is that its direct cost to *x* is 60 and that *z* has last told *y* that *z* could get to *x* with a cost of 5. So in order to get to *x, y* would now route through *z*, fully expecting that *z* will be able to get to *x* with a cost of 5. As of t_1 we have a **routing loop**—in order to get to *x, y* routes through *z*, and *z* routes through *y*. A routing loop is like a black hole—a packet destined for *x* arriving at *y* or *z* as of t_1 will bounce back and forth between these two nodes forever (or until the forwarding tables are changed).

2. Since node *y* has computed a new minimum cost to *x*, it informs *z* of its new distance vector at time t_1.
3. Sometime after t_1, *z* receives *y*'s new distance vector, which indicates that *y*'s minimum cost to *x* is 6. *z* knows it can get to *y* with a cost of 1 and hence computes a new least cost to *x* of $D_z(x)=\min\{50+0,1+6\}=7$. Since *z*'s least cost to *x* has increased, it then informs *y* of its new distance vector at t_2.
4. In a similar manner, after receiving *z*'s new distance vector, *y* determines $D_y(x)=8$ and sends *z* its distance vector. *z* then determines $D_z(x)=9$ and sends *y* its distance vector, and so on.

How long will the process continue? You should convince yourself that the loop will persist for 44 iterations (message exchanges between *y* and *z*)—until *z* eventually computes the cost of its path via *y* to be greater than 50. At this point, *z* will (finally!) determine that its least-cost path to *x* is via its direct connection to *x*. *y* will then route to *x* via *z*. The result of the bad news about the increase in link cost has indeed traveled slowly! What would have happened if the link cost *c*(*y, x*) had changed from 4 to 10,000 and the cost *c*(*z, x*) had been 9,999? Because of such scenarios, the problem we have seen is sometimes referred to as the count-to-infinity problem.

Distance-Vector Algorithm: Adding Poisoned Reverse

The specific looping scenario just described can be avoided using a technique known as *poisoned reverse.* The idea is simple—if *z* routes through *y* to get to destination *x*, then *z* will advertise to *y* that its distance to *x* is infinity, that is, *z* will advertise to *y* that $D_z(x)=\infty$ (even though *z* knows $D_z(x)=5$ in truth). *z* will continue telling this little white lie to *y* as long as it routes to *x* via *y*. Since *y* believes that *z* has no path to *x, y* will never attempt to route to *x* via *z*, as long as *z* continues to route to *x* via *y* (and lies about doing so).

Let's now see how poisoned reverse solves the particular looping problem we encountered before in **Figure 5.5(b)**. As a result of the poisoned reverse, *y*'s distance table indicates $D_z(x)=\infty$. When the cost of the (*x*, *y*) link changes from 4 to 60 at time t_0, *y* updates its table and continues to route directly to *x*, albeit at a higher cost of 60, and informs *z* of its new cost to *x*, that is, $D_y(x)=60$. After receiving the update at t_1, *z* immediately shifts its route to *x* to be via the direct (*z, x*) link at a cost of 50. Since this is a new least-cost path to *x*, and since the path no longer passes through *y, z* now informs *y* that $D_z(x)=50$ at t_2. After receiving the update from *z, y* updates its distance table with $D_y(x)=51$. Also, since *z* is now on *y*'s least-cost path to *x*, *y* poisons the reverse path from *z* to *x* by informing *z* at time t_3 that $D_y(x)=\infty$ (even though *y* knows that Dy(x)=51 in truth).

Does poisoned reverse solve the general count-to-infinity problem? It does not. You should convince yourself that loops involving three or more nodes (rather than simply two immediately neighboring nodes) will not be detected by the poisoned reverse technique.

A Comparison of LS and DV Routing Algorithms

The DV and LS algorithms take complementary approaches toward computing routing. In the DV algorithm, each node talks to *only* its directly connected neighbors, but it provides its neighbors with least-cost estimates from itself to *all* the nodes (that it knows about) in the network. The LS algorithm requires global information. Consequently, when implemented in each and every router, e.g., as in **Figure 4.2** and 5.1, each node would need to communicate with *all* other nodes (via broadcast), but it tells them *only* the costs of its directly connected links. Let's conclude our study of LS and DV algorithms with a quick comparison of some of their attributes. Recall that *N* is the set of nodes (routers) and *E* is the set of edges (links).

- **Message complexity.** We have seen that LS requires each node to know the cost of each link in the network. This requires O(|N| |E|) messages to be sent. Also, whenever a link cost changes, the new link cost must be sent to all nodes. The DV algorithm requires message exchanges between directly connected neighbors at each iteration. We have seen that the time needed for the algorithm to converge can depend on many factors. When link costs change, the DV algorithm will propagate the results of the changed link cost only if the new link cost results in a changed least-cost path for one of the nodes attached to that link.
- **Speed of convergence.** We have seen that our implementation of LS is an $O(|N|^2)$ algorithm requiring O(|N| |E|)) messages. The DV algorithm can converge slowly and can have routing loops while the algorithm is converging. DV also suffers from the count-to-infinity problem.
- **Robustness.** What can happen if a router fails, misbehaves, or is sabotaged? Under LS, a router could broadcast an incorrect cost for one of its attached links (but no others). A node could also corrupt or drop any packets it received as part of an LS broadcast. But an LS node is computing only its own forwarding tables; other nodes are performing similar calculations for themselves. This means route calculations are somewhat separated under LS, providing a degree of robustness. Under DV, a node can advertise incorrect least-cost paths to any or all destinations. (Indeed, in 1997, a malfunctioning router in a small ISP provided national backbone routers with erroneous routing information. This caused other routers to flood the malfunctioning router with traffic and caused large portions of the

Internet to become disconnected for up to several hours **[Neumann 1997]**.) More generally, we note that, at each iteration, a node's calculation in DV is passed on to its neighbor and then indirectly to its neighbor's neighbor on the next iteration. In this sense, an incorrect node calculation can be diffused through the entire network under DV.

In the end, neither algorithm is an obvious winner over the other; indeed, both algorithms are used in the Internet.

5.3 Intra-AS Routing in the Internet: OSPF

In our study of routing algorithms so far, we've viewed the network simply as a collection of interconnected routers. One router was indistinguishable from another in the sense that all routers executed the same routing algorithm to compute routing paths through the entire network. In practice, this model and its view of a homogenous set of routers all executing the same routing algorithm is simplistic for two important reasons:

- **Scale.** As the number of routers becomes large, the overhead involved in communicating, computing, and storing routing information becomes prohibitive. Today's Internet consists of hundreds of millions of routers. Storing routing information for possible destinations at each of these routers would clearly require enormous amounts of memory. The overhead required to broadcast connectivity and link cost updates among all of the routers would be huge! A distance-vector algorithm that iterated among such a large number of routers would surely never converge. Clearly, something must be done to reduce the complexity of route computation in a network as large as the Internet.
- **Administrative autonomy.** As described in **Section 1.3**, the Internet is a network of ISPs, with each ISP consisting of its own network of routers. An ISP generally desires to operate its network as it pleases (for example, to run whatever routing algorithm it chooses within its network) or to hide aspects of its network's internal organization from the outside. Ideally, an organization should be able to operate and administer its network as it wishes, while still being able to connect its network to other outside networks.

Both of these problems can be solved by organizing routers into **autonomous systems (ASs)**, with each AS consisting of a group of routers that are under the same administrative control. Often the routers in an ISP, and the links that interconnect them, constitute a single AS. Some ISPs, however, partition their network into multiple ASs. In particular, some tier-1 ISPs use one gigantic AS for their entire network, whereas others break up their ISP into tens of interconnected ASs. An autonomous system is identified by its globally unique autonomous system number (ASN) **[RFC 1930]**. AS numbers, like IP addresses, are assigned by ICANN regional registries **[ICANN 2016]**.

Routers within the same AS all run the same routing algorithm and have information about each other. The routing algorithm running within an autonomous system is called an **intra-autonomous system routing protocol**.

Open Shortest Path First (OSPF)

OSPF routing and its closely related cousin, IS-IS, are widely used for intra-AS routing in the Internet. The Open in OSPF indicates that the routing protocol specification is publicly available (for example, as opposed to Cisco's EIGRP protocol, which was only recently became open **[Savage 2015]**, after roughly 20 years as a Cisco-proprietary protocol). The most recent version of OSPF, version 2, is defined in **[RFC 2328]**, a public document.

> PRINCIPLES IN PRACTICE
>
> SETTING OSPF LINK WEIGHTS
>
> Our discussion of link-state routing has implicitly assumed that link weights are set, a routing algorithm such as OSPF is run, and traffic flows according to the routing tables computed by the LS algorithm. In terms of cause and effect, the link weights are given (i.e., they come first) and result (via Dijkstra's algorithm) in routing paths that minimize overall cost. In this viewpoint, link weights reflect the cost of using a link (e.g., if link weights are inversely proportional to capacity, then the use of high-capacity links would have smaller weight and thus be more attractive from a routing standpoint) and Dijsktra's algorithm serves to minimize overall cost.
>
> In practice, the cause and effect relationship between link weights and routing paths may be reversed, with network operators configuring link weights in order to obtain routing paths that achieve certain traffic engineering goals **[Fortz 2000**, **Fortz 2002]**. For example, suppose a network operator has an estimate of traffic flow entering the network at each ingress point and destined for each egress point. The operator may then want to put in place a specific routing of ingress-to-egress flows that minimizes the maximum utilization over all of the network's links. But with a routing algorithm such as OSPF, the operator's main "knobs" for tuning the routing of flows through the network are the link weights. Thus, in order to achieve the goal of minimizing the maximum link utilization, the operator must find the set of link weights that achieves this goal. This is a reversal of the cause and effect relationship—the desired routing of flows is known, and the OSPF link weights must be found such that the OSPF routing algorithm results in this desired routing of flows.

OSPF is a link-state protocol that uses flooding of link-state information and a Dijkstra's least-cost path algorithm. With OSPF, each router constructs a complete topological map (that is, a graph) of the entire autonomous system. Each router then locally runs Dijkstra's shortest-path algorithm to determine a shortest-path tree to all *subnets*, with itself as the root node. Individual link costs are configured by the network administrator (see sidebar, **Principles and Practice: Setting OSPF Weights**). The administrator might choose to set all link costs to 1, thus achieving minimum-hop routing, or might choose to set the link weights to be inversely proportional to link capacity in order to discourage traffic from using low-bandwidth links. OSPF does not mandate a policy for how link weights are set (that is the job of the network administrator), but instead provides the mechanisms (protocol) for determining least-cost path routing for the given set of link weights.

With OSPF, a router broadcasts routing information to *all* other routers in the autonomous system, not just to its neighboring routers. A router broadcasts link-state information whenever there is a change in a link's state (for example, a change in cost or a change in up/down status). It also broadcasts a link's state periodically (at least once every 30 minutes), even if the link's state has not changed. RFC 2328 notes that "this periodic updating of link state advertisements adds robustness to the link state algorithm." OSPF advertisements are contained in OSPF messages that are carried directly by IP, with an upper-layer protocol of 89 for OSPF. Thus, the OSPF protocol must itself implement functionality such as reliable message transfer and link-state broadcast. The OSPF protocol also checks that links are operational (via a HELLO message that is sent to an attached neighbor) and allows an OSPF router to obtain a neighboring router's database of network-wide link state.

Some of the advances embodied in OSPF include the following:

- **Security.** Exchanges between OSPF routers (for example, link-state updates) can be authenticated. With authentication, only trusted routers can participate in the OSPF protocol within an AS, thus preventing malicious intruders (or networking students taking their newfound knowledge out for a joyride) from injecting incorrect information into router tables. By default, OSPF packets between routers are not authenticated and could be forged. Two types of authentication can be configured—simple and MD5 (see **Chapter 8** for a discussion on MD5 and authentication in general). With simple authentication, the same password is configured on each router. When a router sends an OSPF packet, it includes the password in plaintext. Clearly, simple authentication is not very secure. MD5 authentication is based on shared secret keys that are configured in all the routers. For each OSPF packet that it sends, the router computes the MD5 hash of the content of the OSPF packet appended with the secret key. (See the discussion of message authentication codes in **Chapter 8**.) Then the router includes the resulting hash value in the OSPF packet. The receiving router, using the preconfigured secret key, will compute an MD5 hash of the packet and compare it with the hash value that the packet carries, thus verifying the packet's authenticity. Sequence numbers are also used with MD5 authentication to protect against replay attacks.
- **Multiple same-cost paths.** When multiple paths to a destination have the same cost, OSPF allows multiple paths to be used (that is, a single path need not be chosen for carrying all traffic when multiple equal-cost paths exist).
- **Integrated support for unicast and multicast routing.** Multicast OSPF (MOSPF) **[RFC 1584]** provides simple extensions to OSPF to provide for multicast routing. MOSPF uses the existing OSPF link database and adds a new type of link-state advertisement to the existing OSPF link-state broadcast mechanism.
- **Support for hierarchy within a single AS.** An OSPF autonomous system can be configured hierarchically into areas. Each area runs its own OSPF link-state routing algorithm, with each router in an area broadcasting its link state to all other routers in that area. Within each area, one or more

area border routers are responsible for routing packets outside the area. Lastly, exactly one OSPF area in the AS is configured to be the backbone area. The primary role of the backbone area is to route traffic between the other areas in the AS. The backbone always contains all area border routers in the AS and may contain non-border routers as well. Inter-area routing within the AS requires that the packet be first routed to an area border router (intra-area routing), then routed through the backbone to the area border router that is in the destination area, and then routed to the final destination.

OSPF is a relatively complex protocol, and our coverage here has been necessarily brief; **[Huitema 1998**; **Moy 1998**; **RFC 2328]** provide additional details.

5.4 Routing Among the ISPs: BGP

We just learned that OSPF is an example of an intra-AS routing protocol. When routing a packet between a source and destination within the same AS, the route the packet follows is entirely determined by the intra-AS routing protocol. However, to route a packet across multiple ASs, say from a smartphone in Timbuktu to a server in a datacenter in Silicon Valley, we need an **inter-autonomous system routing protocol**. Since an inter-AS routing protocol involves coordination among multiple ASs, communicating ASs must run the same inter-AS routing protocol. In fact, in the Internet, all ASs run the same inter-AS routing protocol, called the Border Gateway Protocol, more commonly known as **BGP** **[RFC 4271**; **Stewart 1999]**.

BGP is arguably the most important of all the Internet protocols (the only other contender would be the IP protocol that we studied in **Section 4.3**), as it is the protocol that glues the thousands of ISPs in the Internet together. As we will soon see, BGP is a decentralized and asynchronous protocol in the vein of distance-vector routing described in **Section 5.2.2**. Although BGP is a complex and challenging protocol, to understand the Internet on a deep level, we need to become familiar with its underpinnings and operation. The time we devote to learning BGP will be well worth the effort.

5.4.1 The Role of BGP

To understand the responsibilities of BGP, consider an AS and an arbitrary router in that AS. Recall that every router has a forwarding table, which plays the central role in the process of forwarding arriving packets to outbound router links. As we have learned, for destinations that are within the same AS, the entries in the router's forwarding table are determined by the AS's intra-AS routing protocol. But what about destinations that are outside of the AS? This is precisely where BGP comes to the rescue.

In BGP, packets are not routed to a specific destination address, but instead to CIDRized prefixes, with each prefix representing a subnet or a collection of subnets. In the world of BGP, a destination may take the form 138.16.68/22, which for this example includes 1,024 IP addresses. Thus, a router's forwarding table will have entries of the form (*x*, *I*), where *x* is a prefix (such as 138.16.68/22) and *I* is an interface number for one of the router's interfaces.

As an inter-AS routing protocol, BGP provides each router a means to:

1. **Obtain prefix reachability information from neighboring ASs.** In particular, BGP allows each

subnet to advertise its existence to the rest of the Internet. A subnet screams, “I exist and I am here,” and BGP makes sure that all the routers in the Internet know about this subnet. If it weren’t for BGP, each subnet would be an isolated island—alone, unknown and unreachable by the rest of the Internet.

2. **Determine the “best” routes to the prefixes.** A router may learn about two or more different routes to a specific prefix. To determine the best route, the router will locally run a BGP route-selection procedure (using the prefix reachability information it obtained via neighboring routers). The best route will be determined based on policy as well as the reachability information.

Let us now delve into how BGP carries out these two tasks.

+++

5.4.2 Advertising BGP Route Information

Consider the network shown in **Figure 5.8**. As we can see, this simple network has three autonomous systems: AS1, AS2, and AS3. As shown, AS3 includes a subnet with prefix x. For each AS, each router is either a **gateway router** or an **internal router**. A gateway router is a router on the edge of an AS that directly connects to one or more routers in other ASs. An **internal router** connects only to hosts and routers within its own AS. In AS1, for example, router 1c is a gateway router; routers 1a, 1b, and 1d are internal routers.

Let’s consider the task of advertising reachability information for prefix x to all of the routers shown in **Figure 5.8**. At a high level, this is straightforward. First, AS3 sends a BGP message to AS2, saying that x exists and is in AS3; let’s denote this message as “AS3 x”. Then AS2 sends a BGP message to AS1, saying that x exists and that you can get to x by first passing through AS2 and then going to AS3; let’s denote that message as “AS2 AS3 x”. In this manner, each of the autonomous systems will not only learn about the existence of x, but also learn about a path of autonomous systems that leads to x.

Although the discussion in the above paragraph about advertising BGP reachability information should get the general idea across, it is not precise in the sense that autonomous systems do not actually send messages to each other, but instead routers do. To understand this, let’s now re-examine the example in **Figure 5.8**. In BGP,

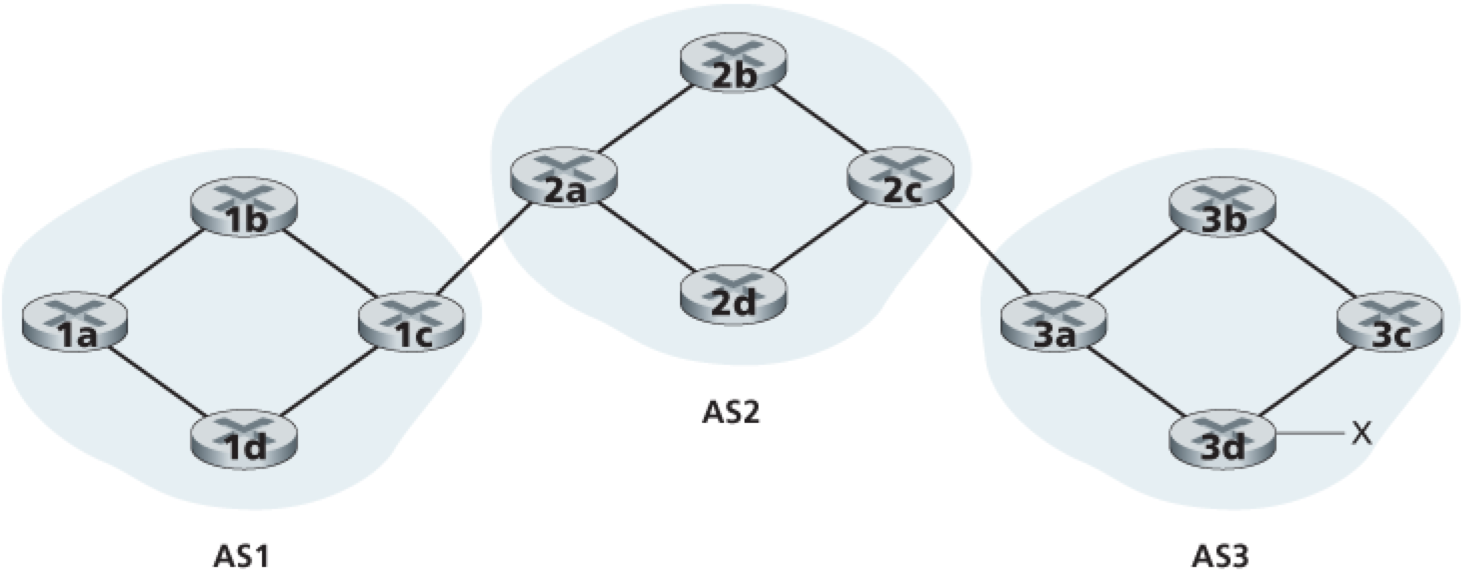


Figure 5.8 Network with three autonomous systems. AS3 includes a subnet with prefix x

pairs of routers exchange routing information over semi-permanent TCP connections using port 179. Each such TCP connection, along with all the BGP messages sent over the connection, is called a **BGP connection**. Furthermore, a BGP connection that spans two ASs is called an **external BGP (eBGP)** connection, and a BGP session between routers in the same AS is called an **internal BGP (iBGP)** connection. Examples of BGP connections for the network in **Figure 5.8** are shown in **Figure 5.9**. There is typically one eBGP connection for each link that directly connects gateway routers in different ASs; thus, in **Figure 5.9**, there is an eBGP connection between gateway routers 1c and 2a and an eBGP connection between gateway routers 2c and 3a.

There are also iBGP connections between routers within each of the ASs. In particular, **Figure 5.9** displays a common configuration of one BGP connection for each pair of routers internal to an AS, creating a mesh of TCP connections within each AS. In **Figure 5.9**, the eBGP connections are shown with the long dashes; the iBGP connections are shown with the short dashes. Note that iBGP connections do not always correspond to physical links.

In order to propagate the reachability information, both iBGP and eBGP sessions are used. Consider again advertising the reachability information for prefix x to all routers in AS1 and AS2. In this process, gateway router 3a first sends an eBGP message “AS3 x” to gateway router 2c. Gateway router 2c then sends the iBGP message “AS3 x” to all of the other routers in AS2, including to gateway router 2a. Gateway router 2a then sends the eBGP message “AS2 AS3 x” to gateway router 1c.

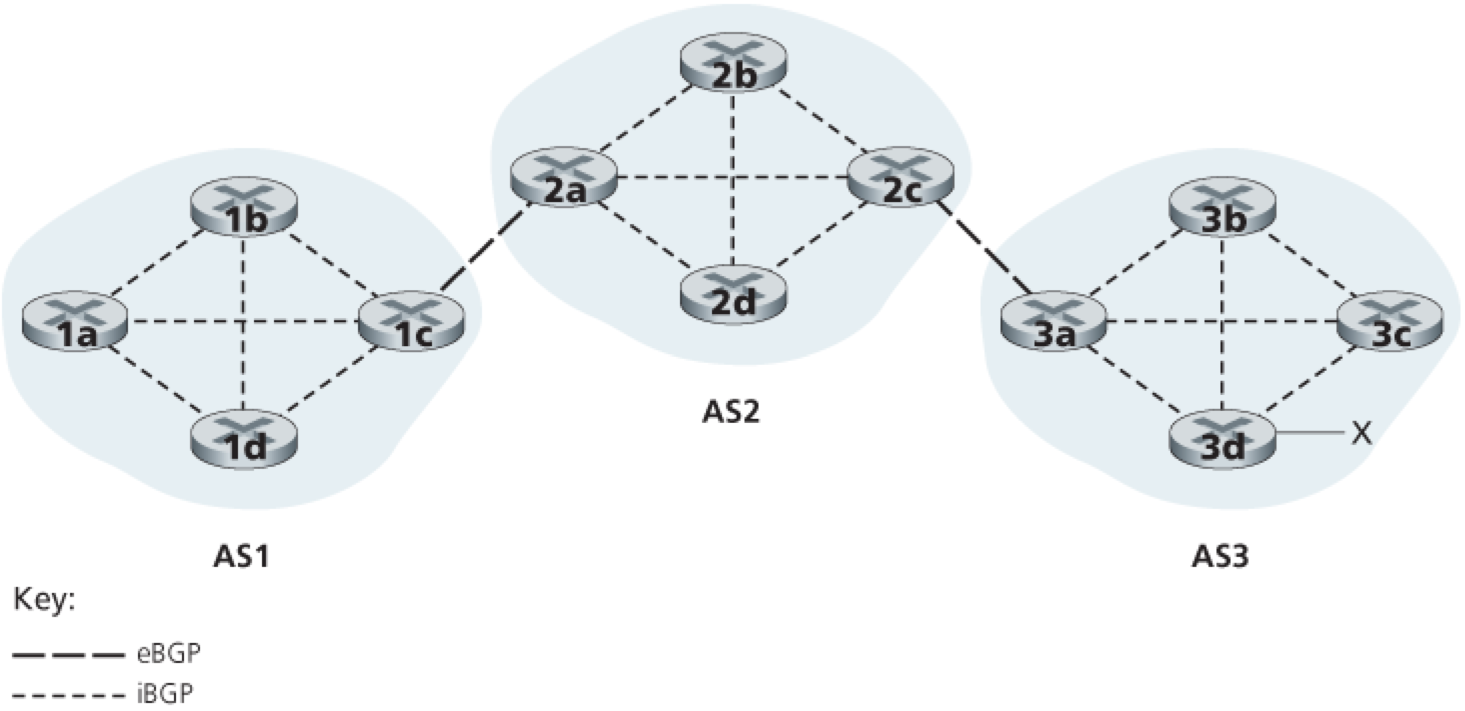


Figure 5.9 eBGP and iBGP connections

Finally, gateway router 1c uses iBGP to send the message “AS2 AS3 x” to all the routers in AS1. After this process is complete, each router in AS1 and AS2 is aware of the existence of x and is also aware of an AS path that leads to x.

Of course, in a real network, from a given router there may be many different paths to a given destination, each through a different sequence of ASs. For example, consider the network in **Figure 5.10**, which is the original network in **Figure 5.8**, with an additional physical link from router 1d to router 3d. In this case, there are two paths from AS1 to x: the path “AS2 AS3 x” via router 1c; and the new path “AS3 x” via the router 1d.

5.4.3 Determining the Best Routes

As we have just learned, there may be many paths from a given router to a destination subnet. In fact, in the Internet, routers often receive reachability information about dozens of different possible paths. How does a router choose among these paths (and then configure its forwarding table accordingly)?

Before addressing this critical question, we need to introduce a little more BGP terminology. When a router advertises a prefix across a BGP connection, it includes with the prefix several **BGP attributes**. In BGP jargon, a prefix along with its attributes is called a **route**. Two of the more important attributes are AS-PATH and NEXT-HOP. The AS-PATH attribute contains the list of ASs through which the

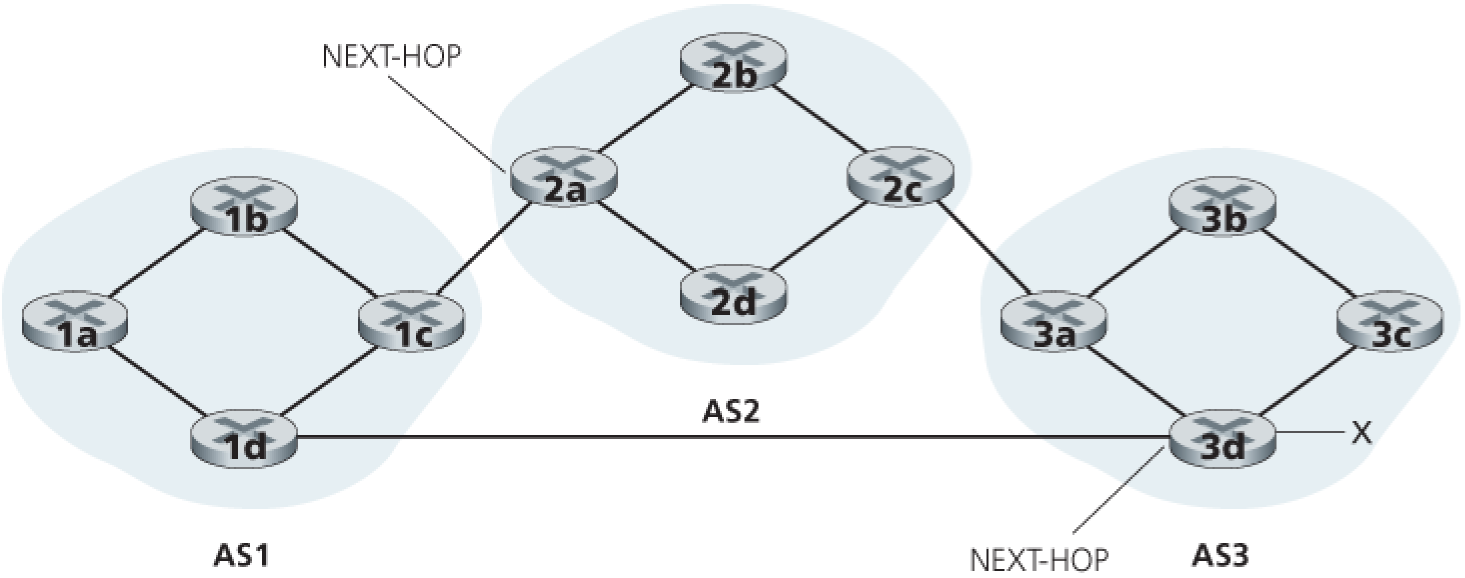


Figure 5.10 Network augmented with peering link between AS1 and AS3

advertisement has passed, as we've seen in our examples above. To generate the AS-PATH value, when a prefix is passed to an AS, the AS adds its ASN to the existing list in the AS-PATH. For example, in **Figure 5.10**, there are two routes from AS1 to subnet x: one which uses the AS-PATH "AS2 AS3"; and another that uses the AS-PATH "A3". BGP routers also use the AS-PATH attribute to detect and prevent looping advertisements; specifically, if a router sees that its own AS is contained in the path list, it will reject the advertisement.

Providing the critical link between the inter-AS and intra-AS routing protocols, the NEXT-HOP attribute has a subtle but important use. The NEXT-HOP is the *IP address of the router interface that begins the AS-PATH*. To gain insight into this attribute, let's again refer to **Figure 5.10**. As indicated in **Figure 5.10**, the NEXT-HOP attribute for the route "AS2 AS3 x" from AS1 to x that passes through AS2 is the IP address of the left interface on router 2a. The NEXT-HOP attribute for the route "AS3 x" from AS1 to x that bypasses AS2 is the IP address of the leftmost interface of router 3d. In summary, in this toy example, each router in AS1 becomes aware of two BGP routes to prefix x:

> IP address of leftmost interface for router 2a; AS2 AS3; x
>
> IP address of leftmost interface of router 3d; AS3; x

Here, each BGP route is written as a list with three components: NEXT-HOP; AS-PATH; destination prefix. In practice, a BGP route includes additional attributes, which we will ignore for the time being. Note that the NEXT-HOP attribute is an IP address of a router that does *not* belong to AS1; however, the subnet that contains this IP address directly attaches to AS1.

Hot Potato Routing

We are now *finally* in position to talk about BGP routing algorithms in a precise manner. We will begin with one of the simplest routing algorithms, namely, **hot potato routing**.

Consider router 1b in the network in **Figure 5.10**. As just described, this router will learn about two possible BGP routes to prefix x. In hot potato routing, the route chosen (from among all possible routes) is that route with the least cost to the NEXT-HOP router beginning that route. In this example, router 1b will consult its intra-AS routing information to find the least-cost intra-AS path to NEXT-HOP router 2a and the least-cost intra-AS path to NEXT-HOP router 3d, and then select the route with the smallest of these least-cost paths. For example, suppose that cost is defined as the number of links traversed. Then the least cost from router 1b to router 2a is 2, the least cost from router 1b to router 2d is 3, and router 2a would therefore be selected. Router 1b would then consult its forwarding table (configured by its intra-AS algorithm) and find the interface *I* that is on the least-cost path to router 2a. It then adds (*x*, *I*) to its forwarding table.

The steps for adding an outside-AS prefix in a router's forwarding table for hot potato routing are summarized in **Figure 5.11**. It is important to note that when adding an outside-AS prefix into a forwarding table, both the inter-AS routing protocol (BGP) and the intra-AS routing protocol (e.g., OSPF) are used.

The idea behind hot-potato routing is for router 1b to get packets out of its AS as quickly as possible (more specifically, with the least cost possible) without worrying about the cost of the remaining portions of the path outside of its AS to the destination. In the name "hot potato routing," a packet is analogous to a hot potato that is burning in your hands. Because it is burning hot, you want to pass it off to another person (another AS) as quickly as possible. Hot potato routing is thus a selfish algorithm—it tries to reduce the cost in its own AS while ignoring the other components of the end-to-end costs outside its AS. Note that with hot potato routing, two routers in the same AS may choose two different AS paths to the same prefix. For example, we just saw that router 1b would send packets through AS2 to reach x. However, router 1d would bypass AS2 and send packets directly to AS3 to reach x.

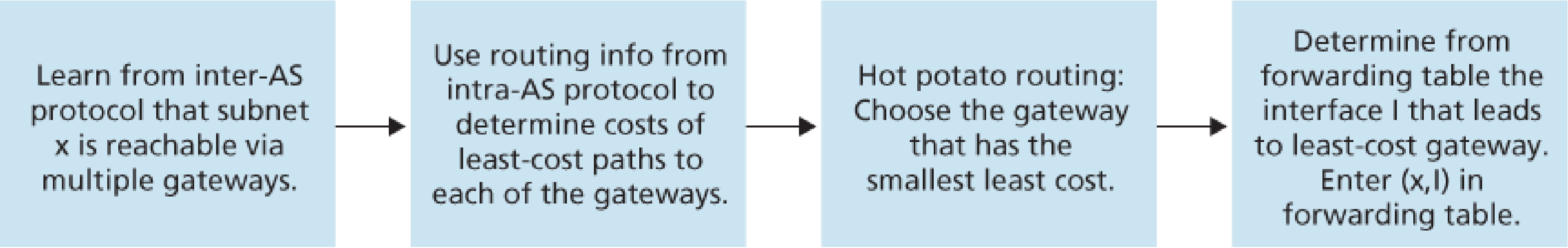

Learn from inter-AS protocol that subnet x is reachable via multiple gateways.
Use routing info from intra-AS protocol to determine costs of least-cost paths to each of the gateways.
Hot potato routing: Choose the gateway that has the smallest least cost.
Determine from forwarding table the interface I that leads to least-cost gateway. Enter (x,I) in forwarding table.

Figure 5.11 Steps in adding outside-AS destination in a router's forwarding table

Route-Selection Algorithm

In practice, BGP uses an algorithm that is more complicated than hot potato routing, but nevertheless incorporates hot potato routing. For any given destination prefix, the input into BGP's route-selection algorithm is the set of all routes to that prefix that have been learned and accepted by the router. If there is only one such route, then BGP obviously selects that route. If there are two or more routes to the same prefix, then BGP sequentially invokes the following elimination rules until one route remains:

1. A route is assigned a **local preference** value as one of its attributes (in addition to the AS-PATH and NEXT-HOP attributes). The local preference of a route could have been set by the router or could have been learned from another router in the same AS. The value of the local preference attribute is a policy decision that is left entirely up to the AS's network administrator. (We will shortly discuss BGP policy issues in some detail.) The routes with the highest local preference values are selected.
2. From the remaining routes (all with the same highest local preference value), the route with the shortest AS-PATH is selected. If this rule were the only rule for route selection, then BGP would be using a DV algorithm for path determination, where the distance metric uses the number of AS hops rather than the number of router hops.
3. From the remaining routes (all with the same highest local preference value and the same AS-PATH length), hot potato routing is used, that is, the route with the closest NEXT-HOP router is selected.
4. If more than one route still remains, the router uses BGP identifiers to select the route; see **[Stewart 1999]**.

As an example, let's again consider router 1b in **Figure 5.10**. Recall that there are exactly two BGP routes to prefix x, one that passes through AS2 and one that bypasses AS2. Also recall that if hot potato routing on its own were used, then BGP would route packets through AS2 to prefix x. But in the above route-selection algorithm, rule 2 is applied before rule 3, causing BGP to select the route that bypasses AS2, since that route has a shorter AS PATH. So we see that with the above route-selection algorithm, BGP is no longer a selfish algorithm—it first looks for routes with short AS paths (thereby likely reducing end-to-end delay).

As noted above, BGP is the *de facto* standard for inter-AS routing for the Internet. To see the contents of various BGP routing tables (large!) extracted from routers in tier-1 ISPs, see **http://www.routeviews.org**. BGP routing tables often contain over half a million routes (that is, prefixes and corresponding attributes). Statistics about the size and characteristics of BGP routing tables are presented in **[Potaroo 2016]**.

5.4.4 IP-Anycast

In addition to being the Internet's inter-AS routing protocol, BGP is often used to implement the IP-anycast service **[RFC 1546**, **RFC 7094]**, which is commonly used in DNS. To motivate IP-anycast, consider that in many applications, we are interested in (1) replicating the same content on different servers in many different dispersed geographical locations, and (2) having each user access the content from the server that is closest. For example, a CDN may replicate videos and other objects on servers in different countries. Similarly, the DNS system can replicate DNS records on DNS servers throughout the world. When a user wants to access this replicated content, it is desirable to point the user to the "nearest" server with the replicated content. BGP's route-selection algorithm provides an easy and natural mechanism for doing so.

To make our discussion concrete, let's describe how a CDN might use IP-anycast. As shown in **Figure 5.12**, during the IP-anycast configuration stage, the CDN company assigns the *same* IP address to each of its servers, and uses standard BGP to advertise this IP address from each of the servers. When a BGP router receives multiple route advertisements for this IP address, it treats these advertisements as providing different paths to the same physical location (when, in fact, the advertisements are for different paths to different physical locations). When configuring its routing table, each router will locally use the BGP route-selection algorithm to pick the "best" (for example, closest, as determined by AS-hop counts) route to that IP address. For example, if one BGP route (corresponding to one location) is only one AS hop away from the router, and all other BGP routes (corresponding to other locations) are two or more AS hops away, then the BGP router would choose to route packets to the location that is one hop away. After this initial BGP address-advertisement phase, the CDN can do its main job of distributing content. When a client requests the video, the CDN returns to the client the common IP address used by the geographically dispersed servers, no matter where the client is located. When the client sends a request to that IP address, Internet routers then forward the request packet to the "closest" server, as defined by the BGP route-selection algorithm.

Although the above CDN example nicely illustrates how IP-anycast can be used, in practice CDNs generally choose not to use IP-anycast because BGP routing changes can result in different packets of the same TCP connection arriving at different instances of the Web server. But IP-anycast is extensively used by the DNS system to direct DNS queries to the closest root DNS server. Recall from **Section 2.4**, there are currently 13 IP addresses for root DNS servers. But corresponding

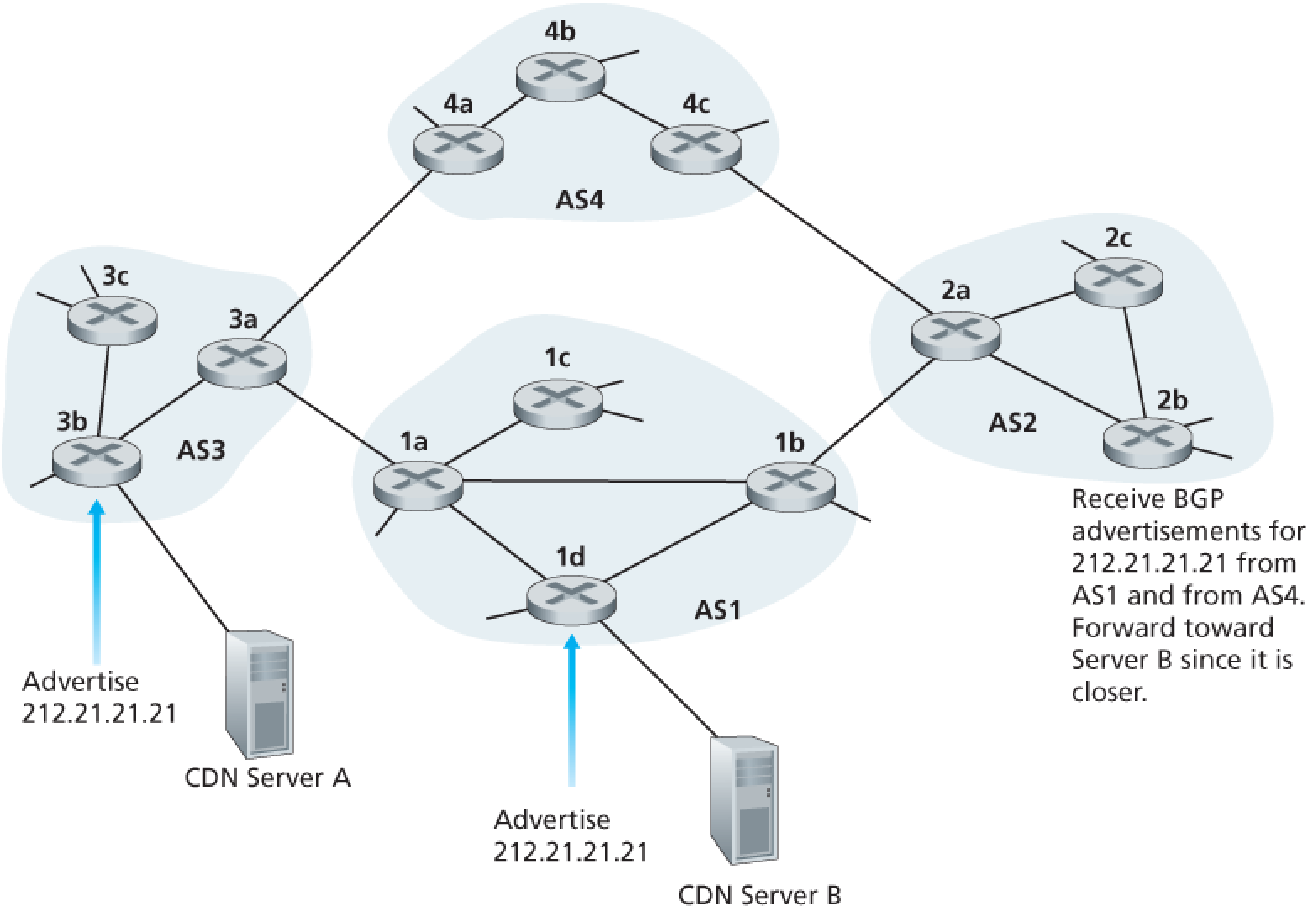

4b
4a
4c
AS4
3c
3a
2c
2a
1c
3b
AS3
1a
1b
2b
AS2
1d
AS1
Receive BGP advertisements for 212.21.21.21 from AS1 and from AS4. Forward toward Server B since it is closer.
Advertise 212.21.21.21
CDN Server A
Advertise 212.21.21.21
CDN Server B

Figure 5.12 Using IP-anycast to bring users to the closest CDN server

to each of these addresses, there are multiple DNS root servers, with some of these addresses having over 100 DNS root servers scattered over all corners of the world. When a DNS query is sent to one of these 13 IP addresses, IP anycast is used to route the query to the nearest of the DNS root servers that is responsible for that address.

5.4.5 Routing Policy

When a router selects a route to a destination, the AS routing policy can trump all other considerations, such as shortest AS path or hot potato routing. Indeed, in the route-selection algorithm, routes are first selected according to the local-preference attribute, whose value is fixed by the policy of the local AS.

Let’s illustrate some of the basic concepts of BGP routing policy with a simple example. Figure 5.13 shows six interconnected autonomous systems: A, B, C, W, X, and Y. It is important to note that A, B, C, W, X, and Y are ASs, not routers. Let’s

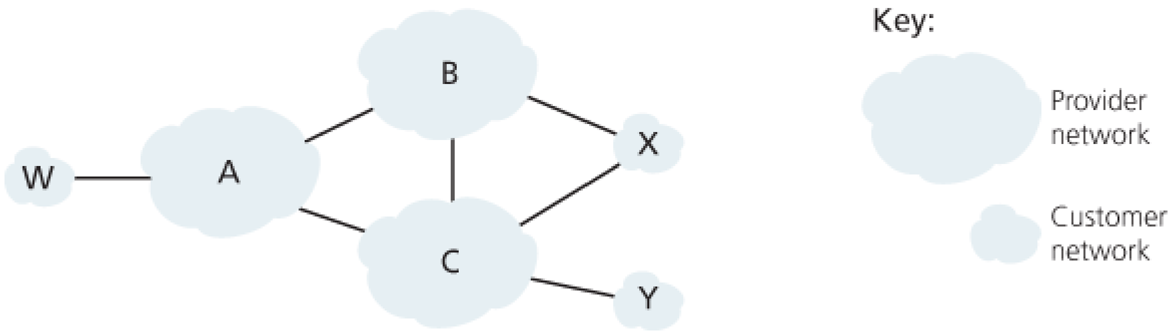


Figure 5.13 A simple BGP policy scenario

assume that autonomous systems W, X, and Y are access ISPs and that A, B, and C are backbone provider networks. We'll also assume that A, B, and C, directly send traffic to each other, and provide full BGP information to their customer networks. All traffic entering an ISP access network must be destined for that network, and all traffic leaving an ISP access network must have originated in that network. W and Y are clearly access ISPs. X is a **multi-homed access ISP**, since it is connected to the rest of the network via two different providers (a scenario that is becoming increasingly common in practice). However, like W and Y, X itself must be the source/destination of all traffic leaving/entering X. But how will this stub network behavior be implemented and enforced? How will X be prevented from forwarding traffic between B and C? This can easily be accomplished by controlling the manner in which BGP routes are advertised. In particular X will function as an access ISP network if it advertises (to its neighbors B and C) that it has no paths to any other destinations except itself. That is, even though X may know of a path, say XCY, that reaches network Y, it will not advertise this path to B. Since B is unaware that X has a path to Y, B would never forward traffic destined to Y (or C) via X. This simple example illustrates how a selective route advertisement policy can be used to implement customer/provider routing relationships.

Let's next focus on a provider network, say AS B. Suppose that B has learned (from A) that A has a path AW to W. B can thus install the route AW into its routing information base. Clearly, B also wants to advertise the path BAW to its customer, X, so that X knows that it can route to W via B. But should B advertise the path BAW to C? If it does so, then C could route traffic to W via BAW. If A, B, and C are all backbone providers, than B might rightly feel that it should not have to shoulder the burden (and cost!) of carrying transit traffic between A and C. B might rightly feel that it is A's and C's job (and cost!) to make sure that C can route to/from A's customers via a direct connection between A and C. There are currently no official standards that govern how backbone ISPs route among themselves. However, a rule of thumb followed by commercial ISPs is that any traffic flowing across an ISP's backbone network must have either a source or a destination (or both) in a network that is a customer of that ISP; otherwise the traffic would be getting a free ride on the ISP's network. Individual peering agreements (that would govern questions such as

PRINCIPLES IN PRACTICE

> *WHY ARE THERE DIFFERENT INTER-AS AND INTRA-AS ROUTING PROTOCOLS?*
>
> Having now studied the details of specific inter-AS and intra-AS routing protocols deployed in today’s Internet, let’s conclude by considering perhaps the most fundamental question we could ask about these protocols in the first place (hopefully, you have been wondering this all along, and have not lost the forest for the trees!): Why are different inter-AS and intra-AS routing protocols used?
>
> The answer to this question gets at the heart of the differences between the goals of routing within an AS and among ASs:
>
> - **Policy.** Among ASs, policy issues dominate. It may well be important that traffic originating in a given AS not be able to pass through another specific AS. Similarly, a given AS may well want to control what transit traffic it carries between other ASs. We have seen that BGP carries path attributes and provides for controlled distribution of routing information so that such policy-based routing decisions can be made. Within an AS, everything is nominally under the same administrative control, and thus policy issues play a much less important role in choosing routes within the AS.
> - **Scale.** The ability of a routing algorithm and its data structures to scale to handle routing to/among large numbers of networks is a critical issue in inter-AS routing. Within an AS, scalability is less of a concern. For one thing, if a single ISP becomes too large, it is always possible to divide it into two ASs and perform inter-AS routing between the two new ASs. (Recall that OSPF allows such a hierarchy to be built by splitting an AS into areas.)
> - **Performance.** Because inter-AS routing is so policy oriented, the quality (for example, performance) of the routes used is often of secondary concern (that is, a longer or more costly route that satisfies certain policy criteria may well be taken over a route that is shorter but does not meet that criteria). Indeed, we saw that among ASs, there is not even the notion of cost (other than AS hop count) associated with routes. Within a single AS, however, such policy concerns are of less importance, allowing routing to focus more on the level of performance realized on a route.

those raised above) are typically negotiated between pairs of ISPs and are often confidential; **[Huston 1999a]** provides an interesting discussion of peering agreements. For a detailed description of how routing policy reflects commercial relationships among ISPs, see **[Gao 2001**; **Dmitiropoulos 2007]**. For a discussion of BGP routing polices from an ISP standpoint, see **[Caesar 2005b]**.

This completes our brief introduction to BGP. Understanding BGP is important because it plays a central role in the Internet. We encourage you to see the references **[Griffin 2012**; **Stewart 1999**; **Labovitz 1997**; **Halabi 2000**; **Huitema 1998**; **Gao 2001**; **Feamster 2004**; **Caesar 2005b**; **Li 2007]** to learn more about BGP.

5.4.6 Putting the Pieces Together: Obtaining Internet Presence

Although this subsection is not about BGP *per se*, it brings together many of the protocols and concepts we've seen thus far, including IP addressing, DNS, and BGP.

Suppose you have just created a small company that has a number of servers, including a public Web server that describes your company's products and services, a mail server from which your employees obtain their e-mail messages, and a DNS server. Naturally, you would like the entire world to be able to visit your Web site in order to learn about your exciting products and services. Moreover, you would like your employees to be able to send and receive e-mail to potential customers throughout the world.

To meet these goals, you first need to obtain Internet connectivity, which is done by contracting with, and connecting to, a local ISP. Your company will have a gateway router, which will be connected to a router in your local ISP. This connection might be a DSL connection through the existing telephone infrastructure, a leased line to the ISP's router, or one of the many other access solutions described in **Chapter 1**. Your local ISP will also provide you with an IP address range, e.g., a /24 address range consisting of 256 addresses. Once you have your physical connectivity and your IP address range, you will assign one of the IP addresses (in your address range) to your Web server, one to your mail server, one to your DNS server, one to your gateway router, and other IP addresses to other servers and networking devices in your company's network.

In addition to contracting with an ISP, you will also need to contract with an Internet registrar to obtain a domain name for your company, as described in **Chapter 2**. For example, if your company's name is, say, Xanadu Inc., you will naturally try to obtain the domain name **xanadu.com**. Your company must also obtain presence in the DNS system. Specifically, because outsiders will want to contact your DNS server to obtain the IP addresses of your servers, you will also need to provide your registrar with the IP address of your DNS server. Your registrar will then put an entry for your DNS server (domain name and corresponding IP address) in the .com top-level-domain servers, as described in **Chapter 2**. After this step is completed, any user who knows your domain name (e.g., **xanadu.com**) will be able to obtain the IP address of your DNS server via the DNS system.

So that people can discover the IP addresses of your Web server, in your DNS server you will need to include entries that map the host name of your Web server (e.g., **www.xanadu.com**) to its IP address. You will want to have similar entries for other publicly available servers in your company, including your mail server. In this manner, if Alice wants to browse your Web server, the DNS system will contact your DNS server, find the IP address of your Web server, and give it to Alice. Alice can then establish a TCP connection directly with your Web server.

However, there still remains one other necessary and crucial step to allow outsiders from around the world to access your Web server. Consider what happens when Alice, who knows the IP address of your Web server, sends an IP datagram (e.g., a TCP SYN segment) to that IP address. This datagram will be routed through the Internet, visiting a series of routers in many different ASs, and eventually reach your Web server. When any one of the routers receives the datagram, it is going to look for an entry in its forwarding table to determine on which outgoing port it should forward the datagram. Therefore, each of the routers needs to know about the existence of your company’s /24 prefix (or some aggregate entry). How does a router become aware of your company’s prefix? As we have just seen, it becomes aware of it from BGP! Specifically, when your company contracts with a local ISP and gets assigned a prefix (i.e., an address range), your local ISP will use BGP to advertise your prefix to the ISPs to which it connects. Those ISPs will then, in turn, use BGP to propagate the advertisement. Eventually, all Internet routers will know about your prefix (or about some aggregate that includes your prefix) and thus be able to appropriately forward datagrams destined to your Web and mail servers.