port number 5001. When generating a new source port number, the NAT router can select any source port number that is not currently in the NAT translation table. (Note that because a port number field is 16 bits long, the NAT protocol can support over 60,000 simultaneous connections with a single WAN-side IP address for the router!) NAT in the router also adds an entry to its NAT translation table. The Web server, blissfully unaware that the arriving datagram containing the HTTP request has been manipulated by the NAT router, responds with a datagram whose destination address is the IP address of the NAT router, and whose destination port number is 5001. When this datagram arrives at the NAT router, the router indexes the NAT translation table using the destination IP address and destination port number to obtain the appropriate IP address (10.0.0.1) and destination port number (3345) for the browser in the home network. The router then rewrites the datagram's destination address and destination port number, and forwards the datagram into the home network. NAT has enjoyed widespread deployment in recent years. But NAT is not without detractors. First, one might argue that, port numbers are meant to be used for addressing processes, not for addressing hosts. This violation can indeed cause problems for servers running on the home network, since, as we have seen in Chapter 2, server processes wait for incoming requests at well-known port numbers and peers in a P2P protocol need to accept incoming connections when acting as servers. Technical solutions to these problems include NAT traversal tools [RFC 5389] and Universal Plug and Play (UPnP), a protocol that allows a host to discover and configure a nearby NAT [UPnP Forum 2016]. More "philosophical" arguments have also been raised against NAT by architectural purists. Here, the concern is that routers are meant to be layer 3 (i.e., network-layer) devices, and should process packets only up to the network layer. NAT violates this principle that hosts should be talking directly with each other, without interfering nodes modifying IP addresses, much less port numbers. But like it or not, NAT has not become an important component of the Internet, as have other so-called middleboxes [Sekar 2011] that operate at the network layer but have functions that are quite different from routers. Middleboxes do not perform traditional datagram forwarding, but instead perform functions such as NAT, load balancing of traffic flows, traffic firewalling (see accompanying sidebar), and more. The generalized forwarding paradigm that we'll study shortly in Section 4.4 allows a number of these middlebox functions, as well as traditional router forwarding, to be accomplished in a common, integrated manner. FOCUS ON SECURITY INSPECTING DATAGRAMS: FIREWALLS AND INTRUSION DETECTION SYSTEMS Suppose you are assigned the task of administering a home, departmental, university, or corporate network. Attackers, knowing the IP address range of your network, can easily send IP datagrams to addresses in your range. These datagrams can do all kinds of devious things, including mapping your network with ping sweeps and port scans, crashing vulnerable hosts with malformed packets, scanning for open TCP/UDP ports on servers in your network, and infecting hosts by including malware in the packets. As the network administrator, what are you going to do about all those bad guys out there, each capable of sending malicious packets into your network? Two popular defense mechanisms to malicious packet attacks are firewalls and intrusion detection systems (IDSs). As a network administrator, you may first try installing a firewall between your network and the Internet. (Most access routers today have firewall capability.) Firewalls inspect the datagram and segment header fields, denying suspicious datagrams entry into the internal network. For example, a firewall may be configured to block all ICMP echo request packets (see Section 5.6), thereby preventing an attacker from doing a traditional port scan across your IP address range. Firewalls can also block packets based on source and destination IP addresses and port numbers. Additionally, firewalls can be configured to track TCP connections, granting entry only to datagrams that belong to approved connections. Additional protection can be

provided with an IDS. An IDS, typically situated at the network boundary, performs “deep packet inspection,” examining not only header fields but also the payloads in the datagram (including application-layer data). An IDS has a database of packet signatures that are known to be part of attacks. This database is automatically updated as new attacks are discovered. As packets pass through the IDS, the IDS attempts to match header fields and payloads to the signatures in its signature database. If such a match is found, an alert is created. An intrusion prevention system (IPS) is similar to an IDS, except that it actually blocks packets in addition to creating alerts. In Chapter 8, we’ll explore firewalls and IDSs in more detail. Can firewalls and IDSs fully shield your network from all attacks? The answer is clearly no, as attackers continually find new attacks for which signatures are not yet available. But firewalls and traditional signature-based IDSs are useful in protecting your network from known attacks. 4.3.5 IPv6 In the early 1990s, the Internet Engineering Task Force began an effort to develop a successor to the IPv4 protocol. A prime motivation for this effort was the realization that the 32-bit IPv4 address space was beginning to be used up, with new subnets and IP nodes being attached to the Internet (and being allocated unique IP addresses) at a breathtaking rate. To respond to this need for a large IP address space, a new IP protocol, IPv6, was developed. The designers of IPv6 also took this opportunity to tweak and augment other aspects of IPv4, based on the accumulated operational experience with IPv4. The point in time when IPv4 addresses would be completely allocated (and hence no new networks could attach to the Internet) was the subject of considerable debate. The estimates of the two leaders of the IETF’s Address Lifetime Expectations working group were that addresses would become exhausted in 2008 and 2018, respectively [Solensky 1996]. In February 2011, IANA allocated out the last remaining pool of unassigned IPv4 addresses to a regional registry. While these registries still have available IPv4 addresses within their pool, once these addresses are exhausted, there are no more available address blocks that can be allocated from a central pool [Huston 2011a]. A recent survey of IPv4 address-space exhaustion, and the steps taken to prolong the life of the address space is [Richter 2015]. Although the mid-1990s estimates of IPv4 address depletion suggested that a considerable amount of time might be left until the IPv4 address space was exhausted, it was realized that considerable time would be needed to deploy a new technology on such an extensive scale, and so the process to develop IP version 6 (IPv6) [RFC 2460] was begun [RFC 1752]. (An often-asked question is what happened to IPv5? It was initially envisioned that the ST-2 protocol would become IPv5, but ST-2 was later dropped.) An excellent source of information about IPv6 is [Huitema 1998]. IPv6 Datagram Format The format of the IPv6 datagram is shown in Figure 4.26. The most important changes introduced in IPv6 are evident in the datagram format: Expanded addressing capabilities. IPv6 increases the size of the IP address from 32 to 128 bits. This ensures that the world won’t run out of IP addresses. Now, every grain of sand on the planet can be IP-addressable. In addition to unicast and multicast addresses, IPv6 has introduced a new type of address, called an anycast address, that allows a datagram to be delivered to any one of a group of hosts. (This feature could be used, for example, to send an HTTP GET to the nearest of a number of mirror sites that contain a given document.) A streamlined 40-byte header. As discussed below, a number of IPv4 fields have been dropped or made optional. The resulting 40-byte fixed-length header allows for faster processing of the IP datagram by a router. A new encoding of options allows for more flexible options processing. Flow labeling. IPv6 has an elusive definition of a flow. RFC 2460 states that this allows “labeling of packets belonging to particular flows for which the sender Figure 4.26 IPv6 datagram format requests special handling, such as a non-default quality of service or real-time service.” For example, audio and video transmission might likely be treated as a flow. On the other hand, the more traditional applications, such as file transfer and e-mail, might not

be treated as flows. It is possible that the traffic carried by a high-priority user (for example, someone paying for better service for their traffic) might also be treated as a flow. What is clear, however, is that the designers of IPv6 foresaw the eventual need to be able to differentiate among the flows, even if the exact meaning of a flow had yet to be determined. As noted above, a comparison of Figure 4.26 with Figure 4.16 reveals the simpler, more streamlined structure of the IPv6 datagram. The following fields are defined in IPv6: Version. This 4-bit field identifies the IP version number. Not surprisingly, IPv6 carries a value of 6 in this field. Note that putting a 4 in this field does not create a valid IPv4 datagram. (If it did, life would be a lot simpler—see the discussion below regarding the transition from IPv4 to IPv6.) Traffic class. The 8-bit traffic class field, like the TOS field in IPv4, can be used to give priority to certain datagrams within a flow, or it can be used to give priority to datagrams from certain applications (for example, voice-over-IP) over datagrams from other applications (for example, SMTP e-mail). Flow label. As discussed above, this 20-bit field is used to identify a flow of datagrams. Payload length. This 16-bit value is treated as an unsigned integer giving the number of bytes in the IPv6 datagram following the fixed-length, 40-byte datagram header. Next header. This field identifies the protocol to which the contents (data field) of this datagram will be delivered (for example, to TCP or UDP). The field uses the same values as the protocol field in the IPv4 header. Hop limit. The contents of this field are decremented by one by each router that forwards the datagram. If the hop limit count reaches zero, the datagram is discarded. Source and destination addresses. The various formats of the IPv6 128-bit address are described in RFC 4291. Data. This is the payload portion of the IPv6 datagram. When the datagram reaches its destination, the payload will be removed from the IP datagram and passed on to the protocol specified in the next header field. The discussion above identified the purpose of the fields that are included in the IPv6 datagram. Comparing the IPv6 datagram format in Figure 4.26 with the IPv4 datagram format that we saw in Figure 4.16, we notice that several fields appearing in the IPv4 datagram are no longer present in the IPv6 datagram: Fragmentation/reassembly. IPv6 does not allow for fragmentation and reassembly at intermediate routers; these operations can be performed only by the source and destination. If an IPv6 datagram received by a router is too large to be forwarded over the outgoing link, the router simply drops the datagram and sends a “Packet Too Big” ICMP error message (see Section 5.6) back to the sender. The sender can then resend the data, using a smaller IP datagram size. Fragmentation and reassembly is a time-consuming operation; removing this functionality from the routers and placing it squarely in the end systems considerably speeds up IP forwarding within the network. Header checksum. Because the transport-layer (for example, TCP and UDP) and link-layer (for example, Ethernet) protocols in the Internet layers perform checksumming, the designers of IP probably felt that this functionality was sufficiently redundant in the network layer that it could be removed. Once again, fast processing of IP packets was a central concern. Recall from our discussion of IPv4 in Section 4.3.1 that since the IPv4 header contains a TTL field (similar to the hop limit field in IPv6), the IPv4 header checksum needed to be recomputed at every router. As with fragmentation and reassembly, this too was a costly operation in IPv4. Options. An options field is no longer a part of the standard IP header. However, it has not gone away. Instead, the options field is one of the possible next headers pointed to from within the IPv6 header. That is, just as TCP or UDP protocol headers can be the next header within an IP packet, so too can an options field. The removal of the options field results in a fixed-length, 40-byte IP header. Transitioning from IPv4 to IPv6 Now that we have seen the technical details of IPv6, let us consider a very practical matter: How will the public Internet, which is based on IPv4, be transitioned to IPv6? The problem is that while new IPv6- capable systems can be made backward-compatible, that is, can send, route, and receive IPv4 datagrams, already deployed

IPv4-capable systems are not capable of handling IPv6 datagrams. Several options are possible [Huston 2011b, RFC 4213]. One option would be to declare a flag day—a given time and date when all Internet machines would be turned off and upgraded from IPv4 to IPv6. The last major technology transition (from using NCP to using TCP for reliable transport service) occurred almost 35 years ago. Even back then [RFC 801], when the Internet was tiny and still being administered by a small number of "wizards," it was realized that such a flag day was not possible. A flag day involving billions of devices is even more unthinkable today. The approach to IPv4-to-IPv6 transition that has been most widely adopted in practice involves tunneling [RFC 4213]. The basic idea behind tunneling—a key concept with applications in many other scenarios beyond IPv4-to-IPv6 transition, including wide use in the all-IP cellular networks that we'll cover in Chapter 7—is the following. Suppose two IPv6 nodes (in this example, B and E in Figure 4.27) want to interoperate using IPv6 datagrams but are connected to each other by intervening IPv4 routers. We refer to the intervening set of IPv4 routers between two IPv6 routers as a tunnel, as illustrated in Figure 4.27. With tunneling, the IPv6 node on the sending side of the tunnel (in this example, B) takes the entire IPv6 datagram and puts it in the data (payload) field of an IPv4 datagram. This IPv4 datagram is then addressed to the IPv6 node on the receiving side of the tunnel (in this example, E) and sent to the first node in the tunnel (in this example, C). The intervening IPv4 routers in the tunnel route this IPv4 datagram among themselves, just as they would any other datagram, blissfully unaware that the IPv4 datagram itself contains a complete IPv6 datagram. The IPv6 node on the receiving side of the tunnel eventually receives the IPv4 datagram (it is the destination of the IPv4 datagram!), determines that the IPv4 datagram contains an IPv6 datagram (by observing that the protocol number field in the IPv4 datagram is 41 [RFC 4213], indicating that the IPv4 payload is a IPv6 datagram), extracts the IPv6 datagram, and then routes the IPv6 datagram exactly as it would if it had received the IPv6 datagram from a directly connected IPv6 neighbor. We end this section by noting that while the adoption of IPv6 was initially slow to take off [Lawton 2001; Huston 2008b], momentum has been building. NIST [NIST IPv6 2015] reports that more than a third of US government second-level domains are IPv6-enabled. On the client side, Google reports that only about 8 percent of the clients accessing Google services do so via IPv6 [Google IPv6 2015]. But other recent measurements [Czyz 2014] indicate that IPv6 adoption is accelerating. The proliferation of devices such as IP-enabled phones and other portable devices Figure 4.27 Tunneling provides an additional push for more widespread deployment of IPv6. Europe's Third Generation Partnership Program [3GPP 2016] has specified IPv6 as the standard addressing scheme for mobile multimedia. One important lesson that we can learn from the IPv6 experience is that it is enormously difficult to change network-layer protocols. Since the early 1990s, numerous new network-layer protocols have been trumpeted as the next major revolution for the Internet, but most of these protocols have had limited penetration to date. These protocols include IPv6, multicast protocols, and resource reservation protocols; a discussion of these latter two protocols can be found in the online supplement to this text. Indeed, introducing new protocols into the network layer is like replacing the foundation of a house—it is difficult to do without tearing the whole house down or at least temporarily relocating the house's residents. On the other hand, the Internet has witnessed rapid deployment of new protocols at the application layer. The classic examples, of course, are the Web, instant messaging, streaming media, distributed games, and various forms of social media. Introducing new application-layer protocols is like adding a new layer of paint to a house—it is relatively easy to do, and if you choose an attractive color, others in the neighborhood will copy you. In summary, in the future we can certainly expect to see changes in the Internet's network layer, but these changes will likely occur on a time scale that is much

slower than the changes that will occur at the application layer. 4.4 Generalized Forwarding and SDN In Section 4.2.1, we noted that an Internet router's forwarding decision has traditionally been based solely on a packet's destination address. In the previous section, however, we've also seen that there has been a proliferation of middleboxes that perform many layer-3 functions. NAT boxes rewrite header IP addresses and port numbers; firewalls block traffic based on header-field values or redirect packets for additional processing, such as deep packet inspection (DPI). Load-balancers forward packets requesting a given service (e.g., an HTTP request) to one of a set of a set of servers that provide that service. [RFC 3234] lists a number of common middlebox functions. This proliferation of middleboxes, layer-2 switches, and layer-3 routers [Qazi 2013]—each with its own specialized hardware, software and management interfaces—has undoubtedly resulted in costly headaches for many network operators. However, recent advances in software-defined networking have promised, and are now delivering, a unified approach towards providing many of these network-layer functions, and certain link-layer functions as well, in a modern, elegant, and integrated manner. Recall that Section 4.2.1 characterized destination-based forwarding as the two steps of looking up a destination IP address ("match"), then sending the packet into the switching fabric to the specified output port ("action"). Let's now consider a significantly more general "match-plus-action" paradigm, where the "match" can be made over multiple header fields associated with different protocols at different layers in the protocol stack. The "action" can include forwarding the packet to one or more output ports (as in destination-based forwarding), load balancing packets across multiple outgoing interfaces that lead to a service (as in load balancing), rewriting header values (as in NAT), purposefully blocking/dropping a packet (as in a firewall), sending a packet to a special server for further processing and action (as in DPI), and more. In generalized forwarding, a match-plus-action table generalizes the notion of the destination-based forwarding table that we encountered in Section 4.2.1. Because forwarding decisions may be made using network-layer and/or link-layer source and destination addresses, the forwarding devices shown in Figure 4.28 are more accurately described as "packet switches" rather than layer 3 "routers" or layer 2 "switches." Thus, in the remainder of this section, and in Section 5.5, we'll refer Figure 4.28 Generalized forwarding: Each packet switch contains a match-plus-action table that is computed and distributed by a remote controller to these devices as packet switches, adopting the terminology that is gaining widespread adoption in SDN literature. Figure 4.28 shows a match-plus-action table in each packet switch, with the table being computed, installed, and updated by a remote controller. We note that while it is possible for the control components at the individual packet switch to interact with each other (e.g., in a manner similar to that in Figure 4.2), in practice generalized match-plus-action capabilities are implemented via a remote controller that computes, installs, and updates these tables. You might take a minute to compare Figures 4.2, 4.3 and 4.28—what similarities and differences do you notice between destination-based forwarding shown in Figure 4.2 and 4.3, and generalized forwarding shown in Figure 4.28? Our following discussion of generalized forwarding will be based on OpenFlow [McKeown 2008, OpenFlow 2009, Casado 2014, Tourrilhes 2014]—a highly visible and successful standard that has pioneered the notion of the match-plus-action forwarding abstraction and controllers, as well as the SDN revolution more generally [Feamster 2013]. We'll primarily consider OpenFlow 1.0, which introduced key SDN abstractions and functionality in a particularly clear and concise manner. Later versions of OpenFlow introduced additional capabilities as a result of experience gained through implementation and use; current and earlier versions of the OpenFlow standard can be found at [ONF 2016]. Each entry in the match-plus-action forwarding table, known as a flow table in OpenFlow, includes: A set of header field values to which an incoming packet will be matched.

As in the case of destination-based forwarding, hardware-based matching is most rapidly performed in TCAM memory, with more than a million destination address entries being possible [Bosshart 2013]. A packet that matches no flow table entry can be dropped or sent to the remote controller for more processing. In practice, a flow table may be implemented by multiple flow tables for performance or cost reasons [Bosshart 2013], but we'll focus here on the abstraction of a single flow table. A set of counters that are updated as packets are matched to flow table entries. These counters might include the number of packets that have been matched by that table entry, and the time since the table entry was last updated. A set of actions to be taken when a packet matches a flow table entry. These actions might be to forward the packet to a given output port, to drop the packet, makes copies of the packet and sent them to multiple output ports, and/or to rewrite selected header fields. We'll explore matching and actions in more detail in Sections 4.4.1 and 4.4.2, respectively. We'll then study how the network-wide collection of per-packet switch matching rules can be used to implement a wide range of functions including routing, layer-2 switching, firewalling, load-balancing, virtual networks, and more in Section 4.4.3. In closing, we note that the flow table is essentially an API, the abstraction through which an individual packet switch's behavior can be programmed; we'll see in Section 4.4.3 that network-wide behaviors can similarly be programmed by appropriately programming/configuring these tables in a collection of network packet switches [Casado 2014]. 4.4.1 Match Figure 4.29 shows the eleven packet-header fields and the incoming port ID that can be matched in an OpenFlow 1.0 match-plus-action rule. Recall from Figure 4.29 Packet matching fields, OpenFlow 1.0 flow table Section 1.5.2 that a link-layer (layer 2) frame arriving to a packet switch will contain a network-layer (layer 3) datagram as its payload, which in turn will typically contain a transport-layer (layer 4) segment. The first observation we make is that OpenFlow's match abstraction allows for a match to be made on selected fields from three layers of protocol headers (thus rather brazenly defying the layering principle we studied in Section 1.5). Since we've not yet covered the link layer, suffice it to say that the source and destination MAC addresses shown in Figure 4.29 are the link-layer addresses associated with the frame's sending and receiving interfaces; by forwarding on the basis of Ethernet addresses rather than IP addresses, we can see that an OpenFlow-enabled device can equally perform as a router (layer-3 device) forwarding datagrams as well as a switch (layer-2 device) forwarding frames. The Ethernet type field corresponds to the upper layer protocol (e.g., IP) to which the frame's payload will be demultiplexed, and the VLAN fields are concerned with so-called virtual local area networks that we'll study in Chapter 6. The set of twelve values that can be matched in the OpenFlow 1.0 specification has grown to 41 values in more recent OpenFlow specifications [Bosshart 2014]. The ingress port refers to the input port at the packet switch on which a packet is received. The packet's IP source address, IP destination address, IP protocol field, and IP type of service fields were discussed earlier in Section 4.3.1. The transport-layer source and destination port number fields can also be matched. Flow table entries may also have wildcards. For example, an IP address of 128.119.*.* in a flow table will match the corresponding address field of any datagram that has 128.119 as the first 16 bits of its address. Each flow table entry also has an associated priority. If a packet matches multiple flow table entries, the selected match and corresponding action will be that of the highest priority entry with which the packet matches. Lastly, we observe that not all fields in an IP header can be matched. For example OpenFlow does not allow matching on the basis of TTL field or datagram length field. Why are some fields allowed for matching, while others are not? Undoubtedly, the answer has to do with the tradeoff between functionality and complexity. The "art" in choosing an abstraction is to provide for enough functionality to accomplish a task (in this case to implement, configure, and manage a wide range of network-

layer functions that had previously been implemented through an assortment of network-layer devices), without over-burdening the abstraction with so much detail and generality that it becomes bloated and unusable. Butler Lampson has famously noted [Lampson 1983]: Do one thing at a time, and do it well. An interface should capture the minimum essentials of an abstraction. Don't generalize; generalizations are generally wrong. Given OpenFlow's success, one can surmise that its designers indeed chose their abstraction well. Additional details of OpenFlow matching can be found in [OpenFlow 2009, ONF 2016]. 4.4.2 Action As shown in Figure 4.28, each flow table entry has a list of zero or more actions that determine the processing that is to be applied to a packet that matches a flow table entry. If there are multiple actions, they are performed in the order specified in the list. Among the most important possible actions are: Forwarding. An incoming packet may be forwarded to a particular physical output port, broadcast over all ports (except the port on which it arrived) or multicast over a selected set of ports. The packet may be encapsulated and sent to the remote controller for this device. That controller then may (or may not) take some action on that packet, including installing new flow table entries, and may return the packet to the device for forwarding under the updated set of flow table rules. Dropping. A flow table entry with no action indicates that a matched packet should be dropped. Modify-field. The values in ten packet header fields (all layer 2, 3, and 4 fields shown in Figure 4.29 except the IP Protocol field) may be re-written before the packet is forwarded to the chosen output port. 4.4.3 OpenFlow Examples of Match-plus-action in Action Having now considered both the match and action components of generalized forwarding, let's put these ideas together in the context of the sample network shown in Figure 4.30. The network has 6 hosts (h1, h2, h3, h4, h5 and h6) and three packet switches (s1, s2 and s3), each with four local interfaces (numbered 1 through 4). We'll consider a number of network-wide behaviors that we'd like to implement, and the flow table entries in s1, s2 and s3 needed to implement this behavior. Figure 4.30 OpenFlow match-plus-action network with three packet switches, 6 hosts, and an OpenFlow controller A First Example: Simple Forwarding As a very simple example, suppose that the desired forwarding behavior is that packets from h5 or h6 destined to h3 or h4 are to be forwarded from s3 to s1, and then from s1 to s2 (thus completely avoiding the use of the link between s3 and s2). The flow table entry in s1 would be: s1 Flow Table (Example 1) Match Action Ingress Port = 1 ; IP Src = 10.3.*.* ; IP Dst = 10.2.*.* Forward(4) … … Of course, we'll also need a flow table entry in s3 so that datagrams sent from h5 or h6 are forwarded to s1 over outgoing interface 3: s3 Flow Table (Example 1) Match Action IP Src = 10.3.*.* ; IP Dst = 10.2.*.* Forward(3) … … Lastly, we'll also need a flow table entry in s2 to complete this first example, so that datagrams arriving from s1 are forwarded to their destination, either host h3 or h4: s2 Flow Table (Example 1) Match Action Ingress port = 2 ; IP Dst = 10.2.0.3 Forward(3) Ingress port = 2 ; IP Dst = 10.2.0.4 Forward(4) … … A Second Example: Load Balancing As a second example, let's consider a load-balancing scenario, where datagrams from h3 destined to 10.1.*.* are to be forwarded over the direct link between s2 and s1, while datagrams from h4 destined to 10.1.*.* are to be forwarded over the link between s2 and s3 (and then from s3 to s1). Note that this behavior couldn't be achieved with IP's destination-based forwarding. In this case, the flow table in s2 would be: s2 Flow Table (Example 2) Match Action Ingress port = 3; IP Dst = 10.1.*.* Forward(2) Ingress port = 4; IP Dst = 10.1.*.* Forward(1) … … Flow table entries are also needed at s1 to forward the datagrams received from s2 to either h1 or h2; and flow table entries are needed at s3 to forward datagrams received on interface 4 from s2 over interface 3 towards s1. See if you can figure out these flow table entries at s1 and s3. A Third Example: Firewalling As a third example, let's consider a firewall scenario in which s2 wants only to receive (on any of its interfaces) traffic sent from hosts attached to s3. s2 Flow Table (Example 3) Match Action IP Src = 10.3.*.* IP Dst

= 10.2.0.3 Forward(3) IP Src = 10.3.*.* IP Dst = 10.2.0.4 Forward(4) … … If there were no other entries in s2's flow table, then only traffic from 10.3.*.* would be forwarded to the hosts attached to s2. Although we've only considered a few basic scenarios here, the versatility and advantages of generalized forwarding are hopefully apparent. In homework problems, we'll explore how flow tables can be used to create many different logical behaviors, including virtual networks—two or more logically separate networks (each with their own independent and distinct forwarding behavior)—that use the same physical set of packet switches and links. In Section 5.5, we'll return to flow tables when we study the SDN controllers that compute and distribute the flow tables, and the protocol used for communicating between a packet switch and its controller. 4.5 Summary In this chapter we've covered the data plane functions of the network layer—the per-router functions that determine how packets arriving on one of a router's input links are forwarded to one of that router's output links. We began by taking a detailed look at the internal operations of a router, studying input and output port functionality and destination-based forwarding, a router's internal switching mechanism, packet queue management and more. We covered both traditional IP forwarding (where forwarding is based on a datagram's destination address) and generalized forwarding (where forwarding and other functions may be performed using values in several different fields in the datagram's header) and seen the versatility of the latter approach. We also studied the IPv4 and IPv6 protocols in detail, and Internet addressing, which we found to be much deeper, subtler, and more interesting than we might have expected. With our newfound understanding of the network-layer's data plane, we're now ready to dive into the network layer's control plane in Chapter 5! Homework Problems and Questions Chapter 4 Review Questions SECTION 4.1 SECTION 4.2 R1. Let's review some of the terminology used in this textbook. Recall that the name of a transport-layer packet is segment and that the name of a link-layer packet is frame. What is the name of a network-layer packet? Recall that both routers and link-layer switches are called packet switches. What is the fundamental difference between a router and link-layer switch? R2. We noted that network layer functionality can be broadly divided into data plane functionality and control plane functionality. What are the main functions of the data plane? Of the control plane? R3. We made a distinction between the forwarding function and the routing function performed in the network layer. What are the key differences between routing and forwarding? R4. What is the role of the forwarding table within a router? R5. We said that a network layer's service model "defines the characteristics of end-to-end transport of packets between sending and receiving hosts." What is the service model of the Internet's network layer? What guarantees are made by the Internet's service model regarding the host-to-host delivery of datagrams? R6. In Section 4.2 , we saw that a router typically consists of input ports, output ports, a switching fabric and a routing processor. Which of these are implemented in hardware and which are implemented in software? Why? Returning to the notion of the network layer's data plane and control plane, which are implemented in hardware and which are implemented in software? Why? R7. Discuss why each input port in a high-speed router stores a shadow copy of the forwarding table. R8. What is meant by destination-based forwarding? How does this differ from generalized forwarding (assuming you've read Section 4.4 , which of the two approaches are adopted by Software-Defined Networking)? R9. Suppose that an arriving packet matches two or more entries in a router's forwarding table. With traditional destination-based forwarding, what rule does a router apply to determine which SECTION 4.3 of these rules should be applied to determine the output port to which the arriving packet should be switched? R10. Three types of switching fabrics are discussed in Section 4.2 . List and briefly describe each type. Which, if any, can send multiple packets across the fabric in parallel? R11. Describe how packet loss can occur at input ports. Describe how packet loss at input ports can

be eliminated (without using infinite buffers). R12. Describe how packet loss can occur at output ports. Can this loss be prevented by increasing the switch fabric speed? R13. What is HOL blocking? Does it occur in input ports or output ports? R14. In Section 4.2 , we studied FIFO, Priority, Round Robin (RR), and Weighted Fair Queueing (WFQ) packet scheduling disciplines? Which of these queueing disciplines ensure that all packets depart in the order in which they arrived? R15. Give an example showing why a network operator might want one class of packets to be given priority over another class of packets. R16. What is an essential different between RR and WFQ packet scheduling? Is there a case (Hint: Consider the WFQ weights) where RR and WFQ will behave exactly the same? R17. Suppose Host A sends Host B a TCP segment encapsulated in an IP datagram. When Host B receives the datagram, how does the network layer in Host B know it should pass the segment (that is, the payload of the datagram) to TCP rather than to UDP or to some other upper-layer protocol? R18. What field in the IP header can be used to ensure that a packet is forwarded through no more than N routers? R19. Recall that we saw the Internet checksum being used in both transport-layer segment (in UDP and TCP headers, Figures 3.7 and 3.29 respectively) and in network-layer datagrams (IP header, Figure 4.16 ). Now consider a transport layer segment encapsulated in an IP datagram. Are the checksums in the segment header and datagram header computed over any common bytes in the IP datagram? Explain your answer. R20. When a large datagram is fragmented into multiple smaller datagrams, where are these smaller datagrams reassembled into a single larger datagram? R21. Do routers have IP addresses? If so, how many? R22. What is the 32-bit binary equivalent of the IP address 223.1.3.27? R23. Visit a host that uses DHCP to obtain its IP address, network mask, default router, and IP address of its local DNS server. List these values. R24. Suppose there are three routers between a source host and a destination host. Ignoring fragmentation, an IP datagram sent from the source host to the destination host will travel over how many interfaces? How many forwarding tables will be indexed to move the datagram from the source to the destination? SECTION 4.4 Problems R25. Suppose an application generates chunks of 40 bytes of data every 20 msec, and each chunk gets encapsulated in a TCP segment and then an IP datagram. What percentage of each datagram will be overhead, and what percentage will be application data? R26. Suppose you purchase a wireless router and connect it to your cable modem. Also suppose that your ISP dynamically assigns your connected device (that is, your wireless router) one IP address. Also suppose that you have five PCs at home that use 802.11 to wirelessly connect to your wireless router. How are IP addresses assigned to the five PCs? Does the wireless router use NAT? Why or why not? R27. What is meant by the term “route aggregation”? Why is it useful for a router to perform route aggregation? R28. What is meant by a “plug-and-play” or “zeroconf” protocol? R29. What is a private network address? Should a datagram with a private network address ever be present in the larger public Internet? Explain. R30. Compare and contrast the IPv4 and the IPv6 header fields. Do they have any fields in common? R31. It has been said that when IPv6 tunnels through IPv4 routers, IPv6 treats the IPv4 tunnels as link-layer protocols. Do you agree with this statement? Why or why not? R32. How does generalized forwarding differ from destination-based forwarding? R33. What is the difference between a forwarding table that we encountered in destinationbased forwarding in Section 4.1 and OpenFlow’s flow table that we encountered in Section 4.4 ? R34. What is meant by the “match plus action” operation of a router or switch? In the case of destination-based forwarding packet switch, what is matched and what is the action taken? In the case of an SDN, name three fields that can be matched, and three actions that can be taken. R35. Name three header fields in an IP datagram that can be “matched” in OpenFlow 1.0 generalized forwarding. What are three IP datagram header fields that cannot be “matched” in OpenFlow? P1. Consider the network below. a. Show the forwarding table in router A, such that all traffic destined to host

H3 is forwarded through interface 3. b. Can you write down a forwarding table in router A, such that all traffic from H1 destined to host H3 is forwarded through interface 3, while all traffic from H2 destined to host H3 is forwarded through interface 4? (Hint: This is a trick question.) P2. Suppose two packets arrive to two different input ports of a router at exactly the same time. Also suppose there are no other packets anywhere in the router. a. Suppose the two packets are to be forwarded to two different output ports. Is it possible to forward the two packets through the switch fabric at the same time when the fabric uses a shared bus? b. Suppose the two packets are to be forwarded to two different output ports. Is it possible to forward the two packets through the switch fabric at the same time when the fabric uses switching via memory? c. Suppose the two packets are to be forwarded to the same output port. Is it possible to forward the two packets through the switch fabric at the same time when the fabric uses a crossbar? P3. In Section 4.2 , we noted that the maximum queuing delay is (n–1)D if the switching fabric is n times faster than the input line rates. Suppose that all packets are of the same length, n packets arrive at the same time to the n input ports, and all n packets want to be forwarded to different output ports. What is the maximum delay for a packet for the (a) memory, (b) bus, and (c) crossbar switching fabrics? P4. Consider the switch shown below. Suppose that all datagrams have the same fixed length, that the switch operates in a slotted, synchronous manner, and that in one time slot a datagram can be transferred from an input port to an output port. The switch fabric is a crossbar so that at most one datagram can be transferred to a given output port in a time slot, but different output ports can receive datagrams from different input ports in a single time slot. What is the minimal number of time slots needed to transfer the packets shown from input ports to their output ports, assuming any input queue scheduling order you want (i.e., it need not have HOL blocking)? What is the largest number of slots needed, assuming the worst-case scheduling order you can devise, assuming that a non-empty input queue is never idle? P5. Consider a datagram network using 32-bit host addresses. Suppose a router has four links, numbered 0 through 3, and packets are to be forwarded to the link interfaces as follows: Destination Address Range Link Interface 11100000 00000000 00000000 00000000 through 11100000 00111111 11111111 11111111 0 11100000 01000000 00000000 00000000 through 11100000 01000000 11111111 11111111 1 11100000 01000001 00000000 00000000 through 11100001 01111111 11111111 11111111 2 otherwise 3 a. Provide a forwarding table that has five entries, uses longest prefix matching, and forwards packets to the correct link interfaces. b. Describe how your forwarding table determines the appropriate link interface for datagrams with destination addresses: 11001000 10010001 01010001 01010101 11100001 01000000 11000011 00111100 11100001 10000000 00010001 01110111 P6. Consider a datagram network using 8-bit host addresses. Suppose a router uses longest prefix matching and has the following forwarding table: Prefix Match Interface 00 0 010 1 011 2 10 2 11 3 For each of the four interfaces, give the associated range of destination host addresses and the number of addresses in the range. P7. Consider a datagram network using 8-bit host addresses. Suppose a router uses longest prefix matching and has the following forwarding table: Prefix Match Interface 1 0 10 1 111 2 otherwise 3 For each of the four interfaces, give the associated range of destination host addresses and the number of addresses in the range. P8. Consider a router that interconnects three subnets: Subnet 1, Subnet 2, and Subnet 3. Suppose all of the interfaces in each of these three subnets are required to have the prefix 223.1.17/24. Also suppose that Subnet 1 is required to support at least 60 interfaces, Subnet 2 is to support at least 90 interfaces, and Subnet 3 is to support at least 12 interfaces. Provide three network addresses (of the form a.b.c.d/x) that satisfy these constraints. P9. In Section 4.2.2 an example forwarding table (using longest prefix matching) is given. Rewrite this forwarding table using the a.b.c.d/x notation instead of the binary string

notation. P10. In Problem P5 you are asked to provide a forwarding table (using longest prefix matching). Rewrite this forwarding table using the a.b.c.d/x notation instead of the binary string notation. P11. Consider a subnet with prefix 128.119.40.128/26. Give an example of one IP address (of form xxx.xxx.xxx.xxx) that can be assigned to this network. Suppose an ISP owns the block of addresses of the form 128.119.40.64/26. Suppose it wants to create four subnets from this block, with each block having the same number of IP addresses. What are the prefixes (of form a.b.c.d/x) for the four subnets? P12. Consider the topology shown in Figure 4.20 . Denote the three subnets with hosts (starting clockwise at 12:00) as Networks A, B, and C. Denote the subnets without hosts as Networks D, E, and F. a. Assign network addresses to each of these six subnets, with the following constraints: All addresses must be allocated from 214.97.254/23; Subnet A should have enough addresses to support 250 interfaces; Subnet B should have enough addresses to support 120 interfaces; and Subnet C should have enough addresses to support 120 interfaces. Of course, subnets D, E and F should each be able to support two interfaces. For each subnet, the assignment should take the form a.b.c.d/x or a.b.c.d/x – e.f.g.h/y. b. Using your answer to part (a), provide the forwarding tables (using longest prefix matching) for each of the three routers. P13. Use the whois service at the American Registry for Internet Numbers (http://www.arin.net/ whois) to determine the IP address blocks for three universities. Can the whois services be used to determine with certainty the geographical location of a specific IP address? Use www.maxmind.com to determine the locations of the Web servers at each of these universities. P14. Consider sending a 2400-byte datagram into a link that has an MTU of 700 bytes. Suppose the original datagram is stamped with the identification number 422. How many fragments are generated? What are the values in the various fields in the IP datagram(s) generated related to fragmentation? P15. Suppose datagrams are limited to 1,500 bytes (including header) between source Host A and destination Host B. Assuming a 20-byte IP header, how many datagrams would be required to send an MP3 consisting of 5 million bytes? Explain how you computed your answer. P16. Consider the network setup in Figure 4.25 . Suppose that the ISP instead assigns the router the address 24.34.112.235 and that the network address of the home network is 192.168.1/24. a. Assign addresses to all interfaces in the home network. b. Suppose each host has two ongoing TCP connections, all to port 80 at host 128.119.40.86. Provide the six corresponding entries in the NAT translation table. P17. Suppose you are interested in detecting the number of hosts behind a NAT. You observe that the IP layer stamps an identification number sequentially on each IP packet. The identification number of the first IP packet generated by a host is a random number, and the identification numbers of the subsequent IP packets are sequentially assigned. Assume all IP packets generated by hosts behind the NAT are sent to the outside world. a. Based on this observation, and assuming you can sniff all packets sent by the NAT to the outside, can you outline a simple technique that detects the number of unique hosts behind a NAT? Justify your answer. b. If the identification numbers are not sequentially assigned but randomly assigned, would your technique work? Justify your answer. P18. In this problem we'll explore the impact of NATs on P2P applications. Suppose a peer with username Arnold discovers through querying that a peer with username Bernard has a file it wants to download. Also suppose that Bernard and Arnold are both behind a NAT. Try to devise a technique that will allow Arnold to establish a TCP connection with Bernard without applicationspecific NAT configuration. If you have difficulty devising such a technique, discuss why. P19. Consider the SDN OpenFlow network shown in Figure 4.30 . Suppose that the desired forwarding behavior for datagrams arriving at s2 is as follows: any datagrams arriving on input port 1 from hosts h5 or h6 that are destined to hosts h1 or h2 should be forwarded over output port 2; any datagrams arriving on input port 2 from hosts h1 or h2 that are destined to hosts h5 or h6 should be

forwarded over output port 1; any arriving datagrams on input ports 1 or 2 and destined to hosts h3 or h4 should be delivered to the host specified; hosts h3 and h4 should be able to send datagrams to each other. Specify the flow table entries in s2 that implement this forwarding behavior. P20. Consider again the SDN OpenFlow network shown in Figure 4.30 . Suppose that the desired forwarding behavior for datagrams arriving from hosts h3 or h4 at s2 is as follows: any datagrams arriving from host h3 and destined for h1, h2, h5 or h6 should be forwarded in a clockwise direction in the network; any datagrams arriving from host h4 and destined for h1, h2, h5 or h6 should be forwarded in a counter-clockwise direction in the network. Specify the flow table entries in s2 that implement this forwarding behavior. P21. Consider again the scenario from P19 above. Give the flow tables entries at packet switches s1 and s3, such that any arriving datagrams with a source address of h3 or h4 are routed to the destination hosts specified in the destination address field in the IP datagram. (Hint: Your forwarding table rules should include the cases that an arriving datagram is destined for a directly attached host or should be forwarded to a neighboring router for eventual host delivery there.) P22. Consider again the SDN OpenFlow network shown in Figure 4.30 . Suppose we want switch s2 to function as a firewall. Specify the flow table in s2 that implements the following firewall behaviors (specify a different flow table for each of the four firewalling behaviors below) for delivery of datagrams destined to h3 and h4. You do not need to specify the forwarding behavior in s2 that forwards traffic to other routers. Only traffic arriving from hosts h1 and h6 should be delivered to hosts h3 or h4 (i.e., that arriving traffic from hosts h2 and h5 is blocked). Only TCP traffic is allowed to be delivered to hosts h3 or h4 (i.e., that UDP traffic is blocked). Wireshark Lab In the Web site for this textbook, www.pearsonhighered.com/cs-resources, you'll find a Wireshark lab assignment that examines the operation of the IP protocol, and the IP datagram format in particular. AN INTERVIEW WITH... Vinton G. Cerf Vinton G. Cerf is Vice President and Chief Internet Evangelist for Google. He served for over 16 years at MCI in various positions, ending up his tenure there as Senior Vice President for Technology Strategy. He is widely known as the co-designer of the TCP/IP protocols and the architecture of the Internet. During his time from 1976 to 1982 at the US Department of Defense Advanced Research Projects Agency (DARPA), he played a key role leading the development of Internet and Internet-related data packet and security techniques. He received the US Presidential Medal of Freedom in 2005 and the US National Medal of Technology in 1997. He holds a BS in Mathematics from Stanford University and an MS and PhD in computer science from UCLA. What brought you to specialize in networking? I was working as a programmer at UCLA in the late 1960s. My job was supported by the US Defense Advanced Research Projects Agency (called ARPA then, called DARPA now). I was working in the laboratory of Professor Leonard Kleinrock on the Network Measurement Center of the newly created ARPAnet. The first node of the ARPAnet was installed at UCLA on September 1, 1969. I was responsible for programming a computer that was used to capture performance information about the ARPAnet and to report this information back for comparison with mathematical models and predictions of the performance of the network. Several of the other graduate students and I were made responsible for working on the so-called Only traffic destined to h3 is to be delivered (i.e., all traffic to h4 is blocked). Only UDP traffic from h1 and destined to h3 is to be delivered. All other traffic is blocked. host-level protocols of the ARPAnet—the procedures and formats that would allow many different kinds of computers on the network to interact with each other. It was a fascinating exploration into a new world (for me) of distributed computing and communication. Did you imagine that IP would become as pervasive as it is today when you first designed the protocol? When Bob Kahn and I first worked on this in 1973, I think we were mostly very focused on the central question: How can we make heterogeneous packet networks interoperate with one another, assuming we cannot actually

change the networks themselves? We hoped that we could find a way to permit an arbitrary collection of packet-switched networks to be interconnected in a transparent fashion, so that host computers could communicate end-to-end without having to do any translations in between. I think we knew that we were dealing with powerful and expandable technology, but I doubt we had a clear image of what the world would be like with hundreds of millions of computers all interlinked on the Internet. What do you now envision for the future of networking and the Internet? What major challenges/obstacles do you think lie ahead in their development? I believe the Internet itself and networks in general will continue to proliferate. Already there is convincing evidence that there will be billions of Internet-enabled devices on the Internet, including appliances like cell phones, refrigerators, personal digital assistants, home servers, televisions, as well as the usual array of laptops, servers, and so on. Big challenges include support for mobility, battery life, capacity of the access links to the network, and ability to scale the optical core of the network up in an unlimited fashion. Designing an interplanetary extension of the Internet is a project in which I am deeply engaged at the Jet Propulsion Laboratory. We will need to cut over from IPv4 [32-bit addresses] to IPv6 [128 bits]. The list is long! Who has inspired you professionally? My colleague Bob Kahn; my thesis advisor, Gerald Estrin; my best friend, Steve Crocker (we met in high school and he introduced me to computers in 1960!); and the thousands of engineers who continue to evolve the Internet today. Do you have any advice for students entering the networking/Internet field? Think outside the limitations of existing systems—imagine what might be possible; but then do the hard work of figuring out how to get there from the current state of affairs. Dare to dream: A half dozen colleagues and I at the Jet Propulsion Laboratory have been working on the design of an interplanetary extension of the terrestrial Internet. It may take decades to implement this, mission by mission, but to paraphrase: “A man’s reach should exceed his grasp, or what are the heavens for?” 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 welldefined 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 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 Figure 5.3 Abstract graph model of a computer network 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]. 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 G=(N, E) 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 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 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 is a sequence of nodes such that each of the pairs are edges in E. The cost of a path is simply the sum of all the edge costs along the path, that is, 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 c(x, y)=∞. c(x, y)=c(y, x); G=(N, E) (x1,x2,⋯,xp) (x1,x2),(x2,x3),⋯,(xp−1,xp) (x1,x2,⋯, xp) c(x1,x2)+c(x2,x3)+⋯+c(xp−1,xp). 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 loadinsensitive. 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 kth 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 1 Initialization: 2 N' = {u} 3 for all nodes v 4 if v is a neighbor of u 5 then D(v) = c(u, v) 6 else D(v) = ∞ 7 8 Loop 9 find w not in N' such that D(w) is a minimum 10 add w to N' 11 update D(v) for each neighbor v of w and not in N': 12 D(v) = min(D(v), D(w)+ c(w, v) ) 13 /* new cost to v is either old cost to v or known 14 least path cost to w plus cost from w to v */ 15 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 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 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. 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. 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 nodes to determine the minimum cost; in the third iteration nodes, and so on. Overall, the total number of nodes we need to search through over all the iterations is and thus we say that the preceding implementation of the LS algorithm has worst-case complexity of order n squared: O(n ). (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 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 n−1 n−2 n(n+1)/2, 2 1+e. Figure 5.5 Oscillations with congestion-sensitive routing 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, selfterminating, 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 (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, x where the min 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 Since we must begin by traveling to some neighbor v, the least cost from x to y is the minimum of 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 and Plugging these values into Equation 5.1, along with the costs and gives 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 neighborto-neighbor communication that will take place in the DV algorithm. The basic idea is as follows. Each node x begins with D (y), an estimate of the cost of the least-cost path from itself to node y, for all nodes, y, in N. Let 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, , containing x's estimate of its cost to all destinations, y, in N The distance vectors of each of its neighbors, that is, 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: If node x's distance vector has changed as a result of this update step, node x will then send its updated dx(y)=minv{c(x,v)+dv(y)}, (5.1) v c(x,v)+dv(y). c(x,v)+dv(y) dv(z)=5, dx(z)=3, dw(z)=3. c(u,v)=2, c(u,x)=1, c(u,w)=5, du(z)=min{2+5,5+3,1+3}=4, x Dx=[Dx(y): y in N] Dx=[Dx(y): y in N] Dv=[Dv(y): y in N] Dx(y)=minv{c(x,v)+Dv(y)} for each node y in N 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 (y) converges to d (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 (y)= c(x, y)/* if y is not a neighbor then c(x, y)= ∞ */ 4 for each neighbor w 5 D (y) = ? for all destinations y in N 6 for each neighbor w 7 send distance vector D = [D (y): y 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 (y) = min {c(x, v) + D (y)} 15 16 if Dx(y) changed for any destination y 17 send distance vector D = [D (y): y 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. x x x w x x x v v x x 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 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 D [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 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 Dx=[Dx(x),Dx(y),Dx(z)]=[0,2,7]. x = Dx(x)=0Dx(y)=min{c(x,y)+Dy(y),c(x,z)+Dz(y)}=min{2+0, 7+1}=2Dx(z)=min{c(x,y)+Dy(z),c(x,z)+Dz(z)}=min{2+1,7+0}=3 Figure 5.6 Distance-vector (DV) algorithm in operation node x's estimate for the least cost to node z, D (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 and 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. x v*(y)=y v*(z)=y. 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 , 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 , 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 , 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, and At time t , y detects the link0 1 2 Dy(x)=4, Dy(z)=1, Dz(y)=1, Dz(x)=5. 0 cost change (the cost has changed from 4 to 60). y computes its new minimum-cost path to x to have a cost of 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 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