Module 3

Introduction

Figure 4.1 shows a simple network with two hosts, H1 and H2, and several routers on the path between H1 and H2. Suppose that H1 is sending information to H2, and consider the role of the network layer in these hosts and in the intervening routers. The network layer in H1 takes segments from the transport layer in H1, encapsulates each segment into a datagram (that is, a network-layer packet), and then sends the datagram's to its nearby router, R1.

At the receiving host, H2, the network layer receives the datagram's from its nearby router R2, extracts the transport-layer segments, and delivers the segments up to the transport layer at H2. The primary role of the routers is to forward datagram's from input links to output links. Note that the routers in Figure 4.1 are shown with a truncated protocol stack, that is, with no upper layers above the network layer, because (except for control purposes) routers do not run application- and transport-layer protocols

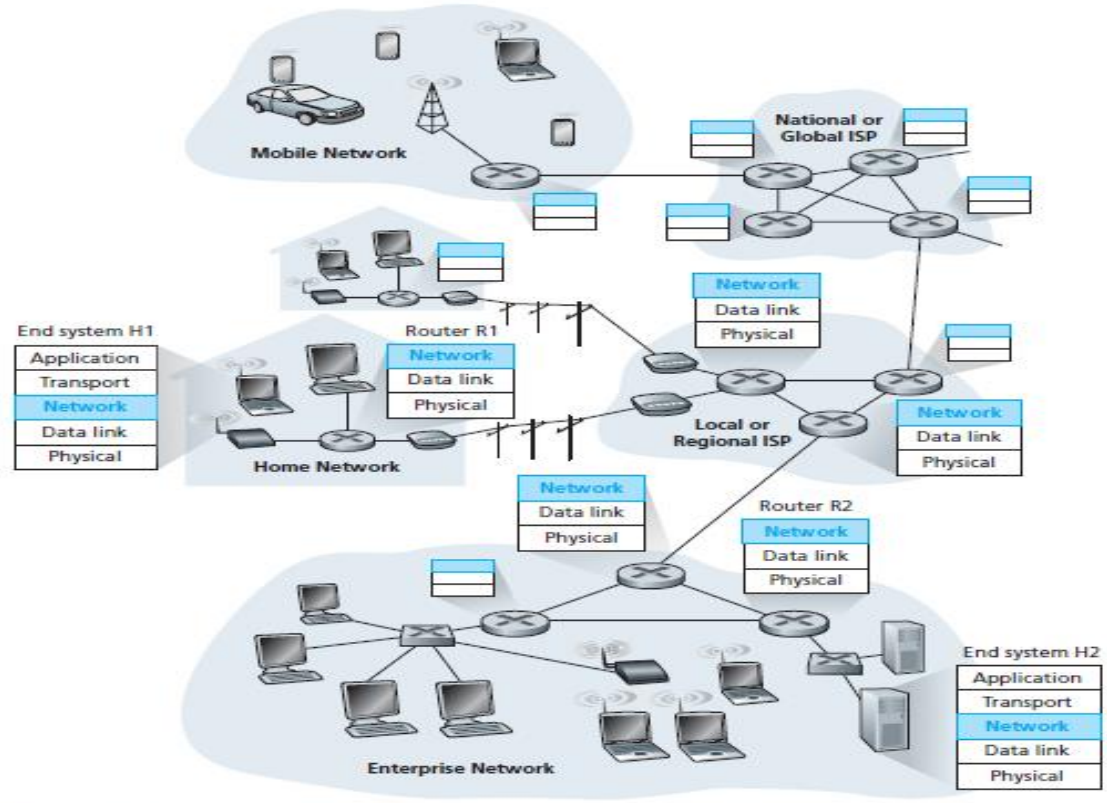


Figure 4.1 • The network layer

Forwarding and Routing

The role of the network layer is thus deceptively simple—to move packets from a sending host to a receiving host. To do so, two important network-layer functions can be identified:

Forwarding: When a packet arrives at a router's input link, the router must move the packet to the appropriate output link. For example, a packet arriving from Host H1 to Router R1 must be forwarded to the next router on a path to H2. In Section 4.3, we'll look inside a router and examine how a packet is actually for-warded from an input link to an output link within a router.

Routing. The network layer must determine the route or path taken by packets as they flow from a sender to a receiver. The algorithms that calculate these paths are referred to as **routing algorithms**. A routing algorithm would determine, for example, the path along which packets flow from H1 to H2.

Every router has a **forwarding table**. A router forwards a packet by examining the value of a field in the arriving packet's header, and then using this header value to index into the router's forwarding table. The value stored in the forwarding table entry for that header indicates the router's outgoing link interface to which that packet is to be forwarded. Depending on the network-layer protocol, the header value could be the destination address of the packet or an indication of the connection to which the packet belongs.

In Figure 4.2, a packet with a header field value of 0111 arrives to a router. The router indexes into its forwarding table and determines that the output link interface for this packet is interface 2. The router then internally forwards the packet to interface 2.

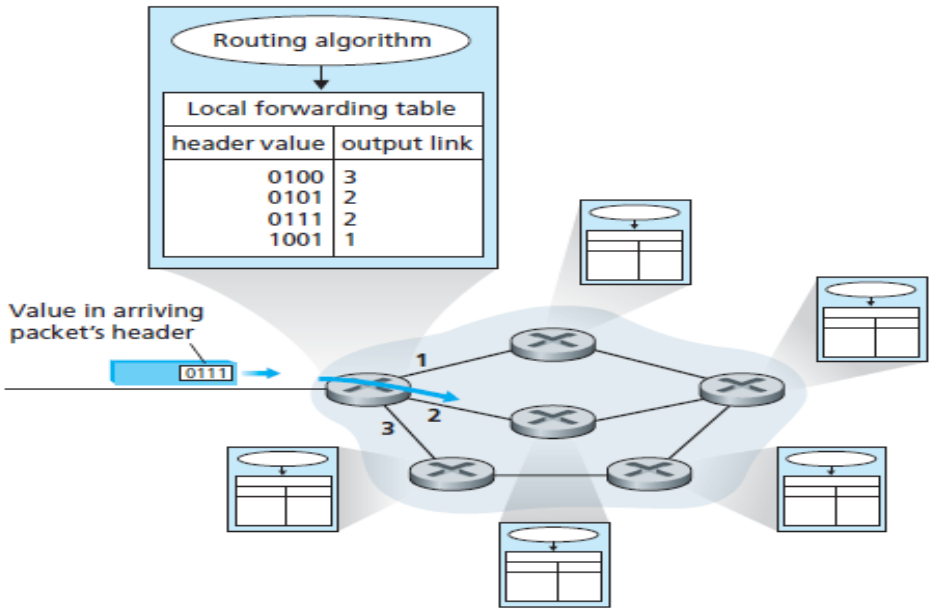


Figure 4.2 ♦ Routing algorithms determine values in forwarding tables

As shown in Figure 4.2, the routing algorithm determines the values that are inserted into the routers' forwarding tables. The routing algorithm may be centralized (e.g., with an algorithm executing on a central site and down-loading routing information to each of the routers) or decentralized (i.e., with a piece of the distributed routing algorithm running in each router).

What's Inside a Router?

Let's turn our attention to its **forwarding function**—the actual transfer of packets from a router's incoming links to the appropriate outgoing links at that router.
A high-level view of a generic router architecture is shown in the below Figure

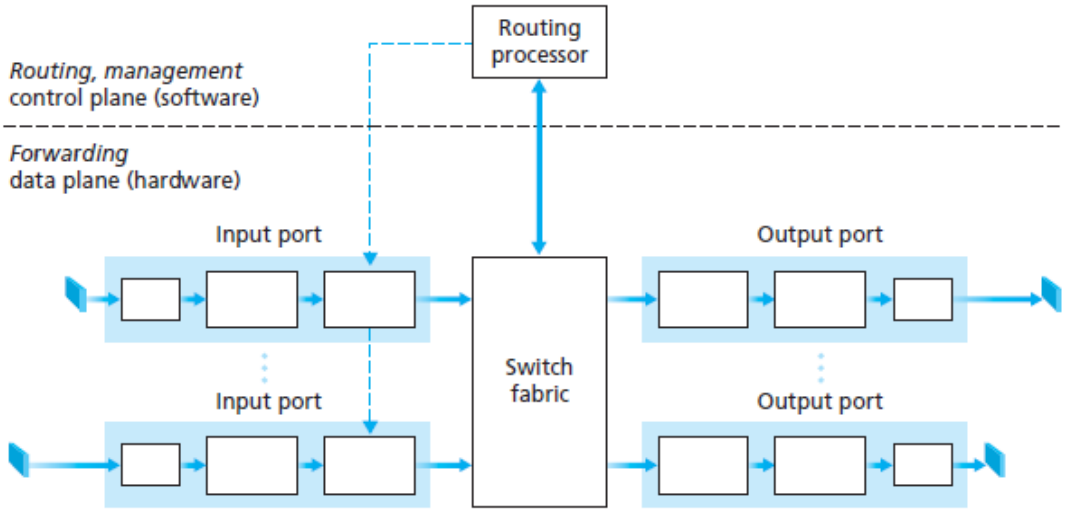


Figure 4.6 ♦ Router architecture

Four router components can be identified as

❖ Input ports:

- ✓ An input port performs several key functions. It performs the physical layer function of terminating an incoming physical link at a router; this is shown in the leftmost box of the input port and the rightmost box of the output port .
- ✓ An input port also performs link-layer functions needed to interoperate with the link layer at the other side of the incoming link; this is represented by the middle boxes in the input and output ports. Perhaps most crucially, the lookup function is also performed at the input port; this will occur in the rightmost box of the input port.
- ✓ It is here that the forwarding table is consulted to determine the router output port to which an arriving packet will be forwarded via the switching fabric.
  Control packets (for example, packets carrying routing protocol information) are forwarded from an input port to the routing processor.

❖ Switching fabric:

The switching fabric connects the router's input ports to its output ports. This switching fabric is completely contained within the router a network inside of a network router.

❖ Output ports:

- ✓ An output port stores packets received from the switching fabric and transmits these packets on the outgoing link by performing the necessary link-layer and physical-layer functions.
- ✓ When a link is bidirectional (that is, carries traffic in both directions), an output port will typically be paired with the input port for that link on the same line card (a printed circuit board containing one or more input ports, which is connected to the switching fabric).

❖ Routing processor:

- ✓ The routing processor executes the routing protocols maintains routing tables and attached link state information and computes the forwarding table for the router.
- ✓ A router's input ports, output ports, and switching fabric together implement the forwarding function and are almost always implemented in hardware.

- ✓ These forwarding functions are sometimes collectively referred to as the **router forwarding plane**. To appreciate why a hardware implementation is needed, consider that with a 10 Gbps input link and a 64-byte IP datagram, the input port has only 51.2 ns to process the datagram before another datagram may arrive.
- ✓ If *N* ports are combined on a line card the datagram-processing pipeline must operate *N* times faster—far too fast for software implementation. Forwarding plane hardware can be implemented either using a router vendor's own hardware designs, or constructed using purchased merchant-silicon chips
- ✓ **router control plane** functions are usually implemented in software and execute on the routing processor

Input Processing:

As discussed above, the input port's line termination function and link-layer processing implement the physical and link layers for that individual input link. The lookup performed in the input port is central to the router's operation—it is here that the router uses the forwarding table to look up the output port to which an arriving packet will be forwarded via the switching fabric.

The forwarding table is computed and updated by the routing processor, with a shadow copy typically stored at each input port.

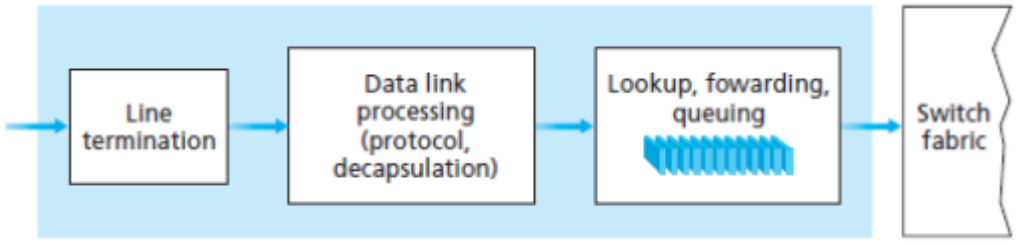


Fig: input port processing

Given the existence of a forwarding table, lookup is conceptually simple—we just search through the forwarding table looking for the longest prefix match, as described once a packet's output port has been determined via the lookup, the packet can be sent into the switching fabric.

In some designs, a packet may be temporarily blocked from entering the switching fabric if packets from other input ports are currently using the fabric. A blocked packet will be queued at the input port and then scheduled to cross the fabric at a later point in time.

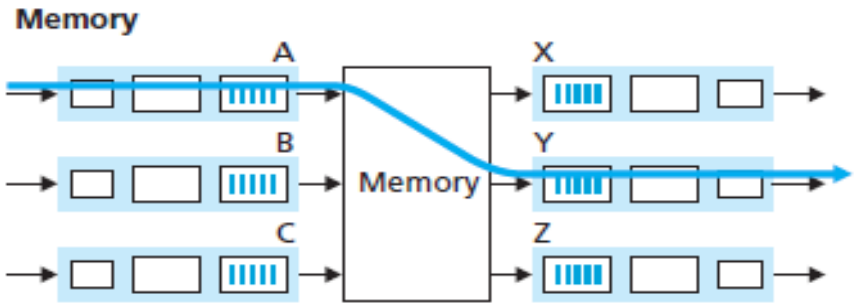
Switching

The switching fabric is at the very heart of a router, as it is through this fabric that the packets are actually switched (that is, forwarded) from an input port to an output port. **Switching can be accomplished in a number of ways.**

1. Switching via memory:

The simplest, earliest routers were traditional computers, with switching between input and output ports being done under direct control of the CPU (routing processor). Input and output ports functioned as traditional I/O devices in a traditional operating system.

An input port with an arriving packet first signalled the routing processor via an interrupt. The packet was then copied from the input port into processor memory. The routing processor then extracted the destination address from the header, looked up the appropriate output port in the forwarding table, and copied the packet to the output port's buffers.



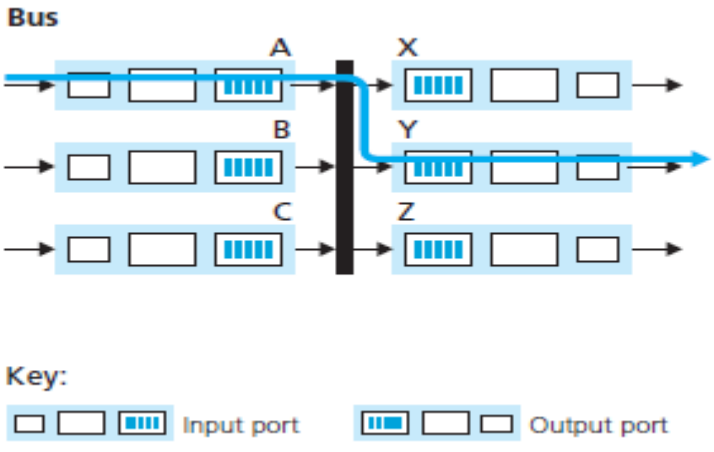
if the memory bandwidth is such that *B* packets per second can be written into, or read from, memory, then the overall forwarding throughput (the total rate at which packets are transferred from input ports to output ports) must be less than *B*/2.

Note also that two packets cannot be forwarded at the same time, even if they have different destination ports, since only one memory read/write over the shared system bus can be done at a time. Many modern routers switch via memory. A major difference from early routers, however, is that the lookup of the destination address and the storing of the packet into the appropriate memory location are performed by processing on the input line cards.
In some ways, routers that switch via memory look very much like shared-memory multiprocessors, with the processing on a line card switching (writing) packets into the memory of the appropriate output port.

2. Switching via a bus.

In this approach, an input port transfers a packet directly to the output port over a shared bus, without intervention by the routing processor. This is typically done by having the input port pre-pend a switch-internal label (header) to the packet indicating the local output port to which this packet is being transferred and transmitting the packet onto the bus.

The packet is received by all output ports, but only the port that matches the label will keep the packet. The label is then removed at the output port, as this label is only used within the switch to cross the bus.



If multiple packets arrive to the router at the same time, each at a different input port, all but one must wait since only one packet can cross the bus at a time. Because every packet must cross the single bus, the switching speed of the router is limited to the bus speed. Nonetheless, switching via a bus is often sufficient for routers that operate in small local area and enterprise networks.

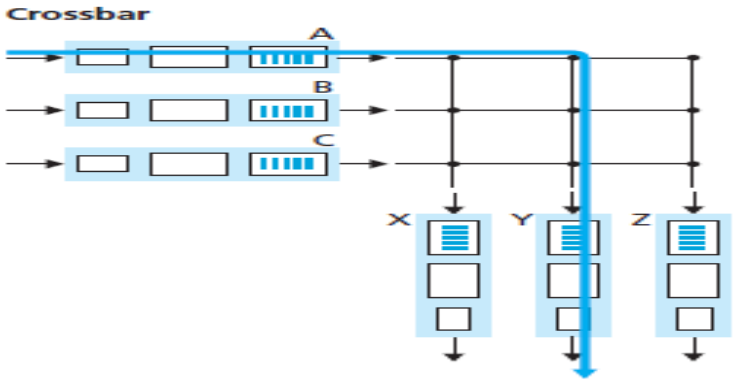
3. Switching via an interconnection network.

One way to overcome the bandwidth limitation of a single, shared bus is to use a more sophisticated interconnection network, such as those that have been used in the past to interconnect processors in a multiprocessor computer architecture. A crossbar switch is an interconnection network consisting of 2*N* buses that connect *N* input ports to *N* output ports

Each vertical bus intersects each horizontal bus at a cross point, which can be opened or closed at any time by the switch fabric controller. When a packet arrives from port A and needs to be forwarded to port Y, the switch controller closes the cross point at the intersection of busses A and Y, and port A then sends the packet onto its bus, which is picked up (only) by bus Y.

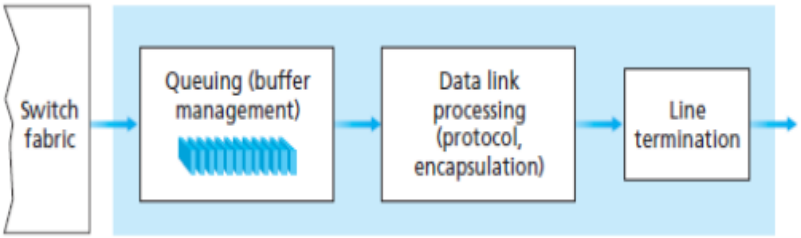
Note that a packet from port B can be forwarded to port X at the same time, since the A-to-Y and B-to-X packets use different input and output busses. Thus, unlike the previous two switching approaches, crossbar networks are capable of forwarding multiple packets in parallel.

However, if two packets from two different input ports are destined to the same output port, then one will have to wait at the input, since only one packet can be sent over any given bus at a time.
More sophisticated interconnection networks use multiple stages of switching elements to allow packets from different input ports to proceed towards the same output port at the same time through the switching fabric.



Output Processing

Output port processing, takes packets that have been stored in the output port's memory and transmits them over the output link. This includes selecting and de-queuing packets for transmission, and performing the needed link layer and physical-layer transmission functions.



Where Does Queuing Occur?

The packet queues may form at both the input ports *and* the output ports, The location and extent of queuing (either at the input port queues or the output port queues) will depend on the traffic load, the relative speed of the switching fabric, and the line speed.

Let's now consider these queues in a bit more detail, since as these queues grow large, the router's memory can eventually be exhausted and **packet loss** will occur when no memory is available to store arriving packets. Recall that in our earlier discussions, we said that packets were "lost within the network" or "dropped at a router." It is here, at these queues within a router, where such packets are actually dropped and lost.

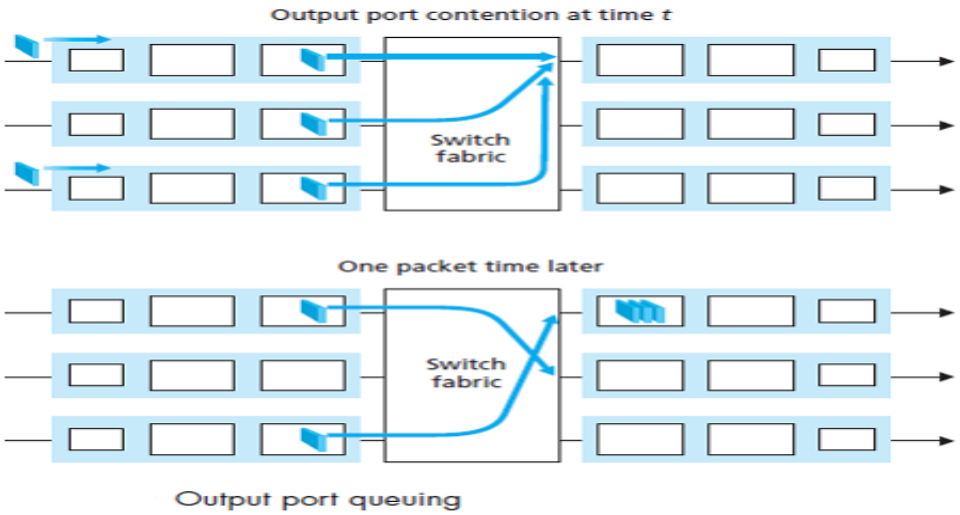
Suppose that the input and output line speeds (transmission rates) all have an identical transmission rate of ***Rline*** packets per second, and that there are *N* input ports and *N* output ports. To further simplify the discussion, let's assume that all packets have the same fixed length, and the packets arrive to input ports in a synchronous manner. That is, the time to send a packet on any link is equal to the time to receive a packet on any link, and during such an interval of time, either zero or one packet can arrive on an input link.

Define the switching fabric transfer rate **Rswitch** as the rate at which packets can be moved from input port to output port. If **Rswitch** is *N* times faster than **Rline**, then only negligible queuing will occur at the input ports. This is because even in the worst case, where all *N* input lines are receiving packets, and all packets are to be forwarded to the same output port, each batch of *N* packets (one packet per input port) can be cleared through the switch fabric before the next batch arrives.

Output port queuing :

Let's suppose that **Rswitch** is still *N* times faster than ***Rline*.** Once again, packets arriving at each of the *N* input ports are destined to the same output port. In this case, in the time it takes to send a single packet onto the outgoing link, *N* new packets will arrive at this output port. Since the output port can transmit only a single packet in a unit of time (the packet transmission *time*), the *N* arriving packets will have to queue (wait) for transmission over the outgoing link.

Then *N* more packets can possibly arrive in the time it takes to transmit just one of the *N* packets that had just previously been queued. And so on. Eventually, the number of queued packets can grow large enough to exhaust available memory at the output port, in which case packets are dropped.



In the following figure , At time *t,* a packet has arrived at each of the incoming input ports, each destined for the uppermost outgoing port. Assuming identical line speeds and a switch operating at three times the line speed. one time unit later, all three original packets have been transferred to the outgoing port and are queued awaiting transmission.

In the next time unit, one of these three packets will have been transmitted over the outgoing link. In our example, two *new* packets have arrived at the incoming side of the switch; one of these packets is destined for this uppermost output port.

A consequence of output port queuing is that a **packet scheduler** at the output port must choose one packet among those queued for transmission. This selection might be done on a simple basis, such as **first-come-first-served (FCFS) scheduling,** or a more sophisticated scheduling discipline such as **weighted fair queuing (WFQ),** which shares the outgoing link fairly among the different end-to-end connections that have packets queued for transmission.

Packet scheduling plays a crucial role in providing **quality-of-service guarantees**. Similarly, if there is not enough memory to buffer an incoming packet, a decision must be made to either drop the arriving packet or remove one or more already-queued packets to make room for the newly arrived packet. In some cases, it may be advantageous to drop a packet *before* the buffer is full in order to provide a congestion signal to the sender. A number of packet-dropping and -marking policies which collectively have become known as **active queue management** (**AQM**) algorithms .

One of the most widely studied and implemented AQM algorithms is the **Random Early Detection** (**RED**) algorithm. Under RED, a weighted average is maintained for the length of the output queue.

- ✓ If the average queue length is less than a minimum threshold, ***minth,*** when a packet arrives, the packet is admitted to the queue.
- ✓ Conversely, if the queue is full or the average queue length is greater than a maximum threshold, ***maxth,*** when a packet arrives, the packet is marked or dropped.
- ✓ Finally, if the packet arrives to find an average queue length in the **interval [*minth, maxth*],** the packet is marked or dropped with a probability that is typically some function of the average queue length, *minth*, and *maxth*.

Input port queuing:

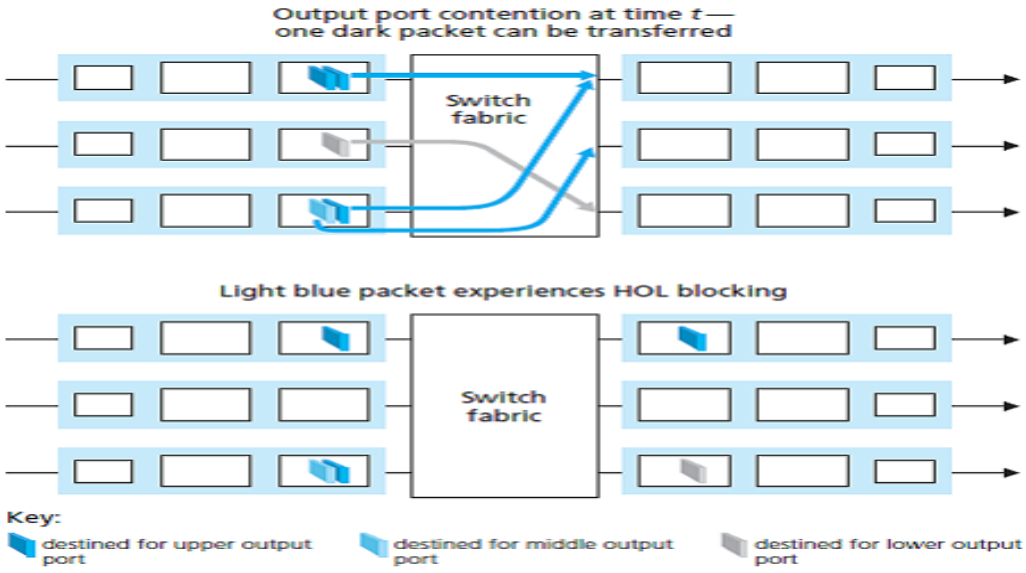
If the switch fabric is not fast enough (relative to the input line speeds) to transfer *all* arriving packets through the fabric without delay, then packet queuing can also occur at the input ports, as packets must join input port queues to wait their turn to be transferred through the switching fabric to the output port.

To illustrate an important consequence of this queuing, consider a crossbar switching fabric and suppose that (1) all link speeds are identical,

(2) that one packet can be transferred from any one input port to a given output port in the same amount of time it takes for a packet to be received on an input link,

(3) packets are moved from a given input queue to their desired output queue in an FCFS manner.

Multiple packets can be transferred in parallel, as long as their output ports are different. However, if two packets at the front of two input queues are destined for the same output queue, then one of the packets will be blocked and must wait at the input queue—the switching fabric can transfer only one packet to a given output port at a time.



HOL blocking at an input queued switch

In the above Figure shows an example in which two packets (darkly shaded) at the front of their input queues are destined for the same upper-right output port. Suppose that the switch fabric chooses to transfer the packet from the front of the upper-left queue.

In this case, the darkly shaded packet in the lower-left queue must wait. But not only must this darkly shaded packet wait, so too must the lightly shaded packet that is queued behind that packet in the lower-left queue, even though there is *no* contention for the middle-right output port This phenomenon is known as "**Head-of-the-line** (**HOL**) **blocking"**

HOL blocking: A queued packet in an input queue must wait for transfer through the fabric even though its output port is free because it is blocked by another packet at the head of the line.

The Routing Control Plane

we've implicitly assumed that the routing control plane fully resides and executes in a routing processor within the router. The network-wide routing control plane is thus decentralized—with different pieces (e.g., of a routing algorithm) executing at different routers and interacting by sending control messages to each other.

Additionally, router and switch vendors bundle their hardware data plane and **software control plane** together into closed (but inter-operable) platforms in a vertically integrated product.
These researchers argue that separating the software control plane from the **hardware data plane** (with a minimal router-resident control plane) can simplify routing by replacing distributed routing calculation with centralized routing calculation, and enable network innovation by allowing different customized control planes to operate over fast hardware data planes.

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 IP 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 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. In addition to unicast and multicast addresses, IPv6 has introduced a new type of address, called an **anycast address**, which allows a datagram to be delivered to any one of a group of hosts.

- **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. A new encoding of options allows for more flexible options processing.
- **Flow labeling and priority:** IPv6 has an elusive definition of a **flow**. RFC 1752 and RFC 2460 state that this allows "labelling of packets belonging to particular flows for which the sender 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.

IPv6 Datagram Format

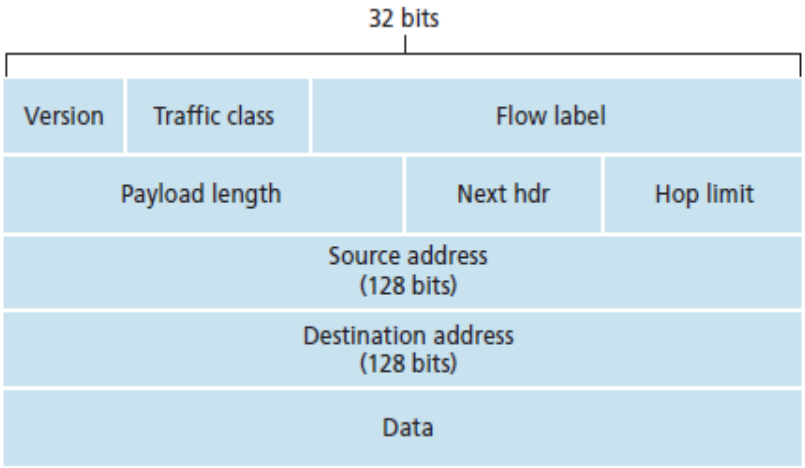


Fig: IPV6 datagram format

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.

- **Traffic class.** This 8-bit field of traffic class to allow different types of IP datagram's i.e for example, datagram's particularly requiring low delay, high throughput, or reliability to be distinguished from each other. For example, it might be useful to distinguish real-time datagrams (such as those used by an IP telephony application) from non-real-time traffic (for example, FTP).

- **Flow label.** 20-bit field is used to identify a flow of datagram's.identify datagram's in same flow.
- **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.
- ***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.

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 below) 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.

❖ **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. The removal of the options field results in a fixed-length, 40 byte IP header.

Transitioning from IPv4 to 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 datagram's, already deployed IPv4-capable systems are not capable of handling IPv6 datagram's. Several options are possible

- 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. 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 hundreds of millions of machines and millions of network administrators and users is even more unthinkable today.

- Probably the most straightforward way to introduce IPv6-capable nodes is a **dual-stack approach,** where IPv6 nodes also have a complete IPv4 implementation. Such a node, referred to as an IPv6/IPv4 node, has the ability to send and receive both IPv4 and IPv6 datagram's.

When interoperating with an IPv4 node, an IPv6/IPv4 node can use IPv4 datagram's; when interoperating with an IPv6 node, an IPv6/IPv4 node can use IPv4 datagram's. IPv6/IPv4 nodes must have both IPv6 and IPv4 addresses.

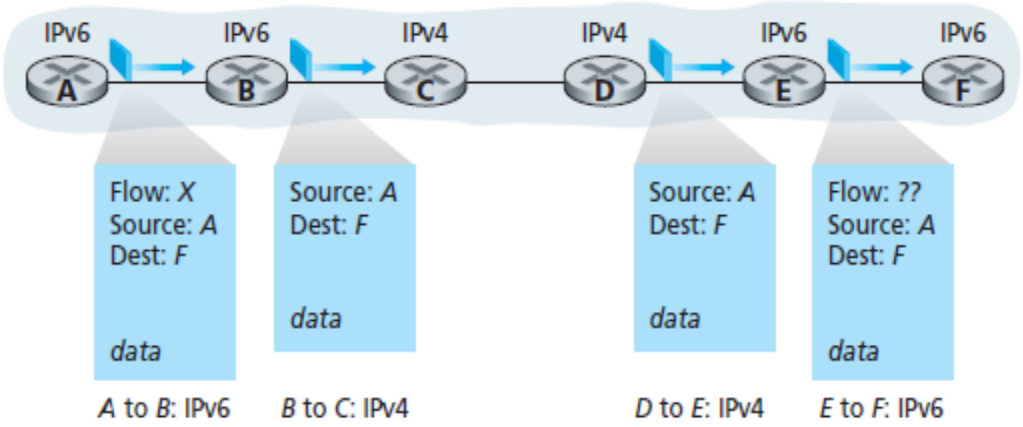


Fig: Dual stack approach

In the dual-stack approach, if either the sender or the receiver is only IPv4- capable, an IPv4 datagram must be used. As a result, it is possible that two IPv6- capable nodes can end up, in essence, sending IPv4 datagram's to each other.

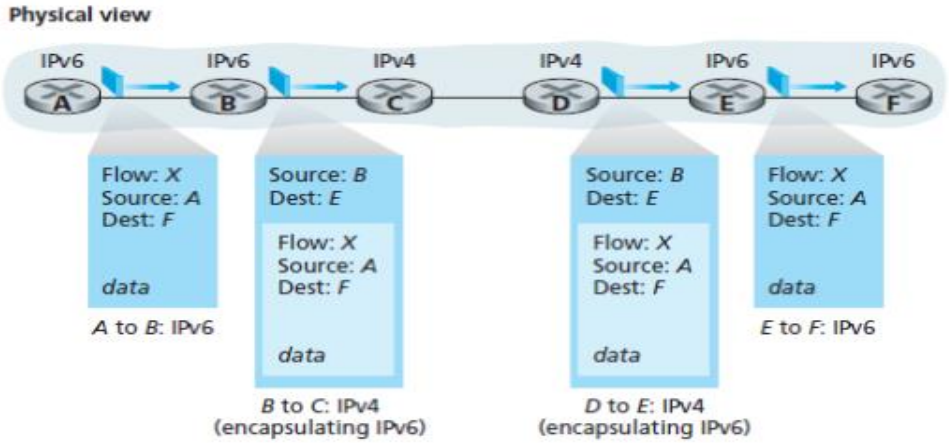
This is illustrated in the below Figure

- ✓ Suppose **Node A** is IPv6-capable and wants to send an IP datagram to **Node F**, which is also IPv6-capable.
- ✓ Nodes A and B can exchange an IPv6 datagram. However, Node B must create an IPv4 datagram to send to C. Certainly, the data field of the IPv6 datagram can be copied into the data field of the IPv4 datagram and appropriate address mapping can be done.
- ✓ However, in performing the conversion from IPv6 to IPv4, there will be IPv6-specific fields in the IPv6 datagram (**for example, the flow label field**) that have no counterpart in IPv4. The information in these fields will be lost.
- ✓ Thus, even though E and F can exchange IPv6 datagrams, the arriving IPv4 datagrams at E from D do not contain all of the fields that were in the original IPv6 datagram sent from A.

❖ An alternative to the dual-stack approach is known as **tunneling**. Tunneling can solve the problem noted above, allowing, for example, E to receive the IPv6 datagram originated by A. The basic idea behind tunneling is the following.

Suppose two IPv6 nodes 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 the below figure With tunnelling.



- ✓ The IPv6 node on the sending side of the tunnel (for 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 (for example, E) and sent to the first node in the tunnel (for 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, 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 neighbour.

A Brief Foray into IP Security

Indeed, IPv4 was designed in an era (the 1970s) when the Internet was primarily used among mutually-trusted networking researchers. Creating a computer network that integrated a multitude of link-layer technologies was already challenging enough, without having to worry about security. But with security being a major concern today, Internet researchers have moved on to design new network-layer protocols that provide a variety of security services.

IPSEC:

- ✓ **IPSec** one of the more popular secure network-layer protocols and also widely deployed in Virtual Private Networks (VPNs).
- ✓ Ip security provides the capability to secure communication across LAN ,across private and Public WANS and across the internet.
- ✓ Ip security is one of the most powerful secure network layer protocol IPsec has been designed to be backward compatible with IPv4 and IPv6.
- ✓ In particular,in order to reap the benefits of IPsec, we don't need to replace the protocol stacks in *all* the routers and hosts in the Internet.

2 modes in IPsec

❖ Transport mode:

- ✓ In this mode, two hosts first establish an IPSec session between themselves. With the session in place, all TCP and UDP segments sent between the transport layer passes a segment to IPSec.

- ✓ IPSec then encrypts the segment, appends additional security fields to the segment, and encapsulates the resulting payload in an ordinary IP datagram. Encrypting only payload section in the datagram not header section .
- ✓ The sending host then sends the datagram into the Internet, which transports it to the destination host. There, IPsec decrypts the segment and passes the unencrypted segment to the transport layer.

❖ **Tunnelling mode:**

In this mode, IPSec then encrypts the complete IP datagram including header section at sender side and at reciver side complete datagram will be decrypted and sent to transport layer

The services provided by an IPsec session include:

- **Cryptographic agreement.** Mechanisms that allow the two communicating hosts to agree on cryptographic algorithms and keys.
- **Encryption of IP datagram payloads.** When the sending host receives a segment from the transport layer, IPSec encrypts the payload. The payload can only be decrypted by IPsec in the receiving host.
- **Data integrity.** IPSec allows the receiving host to verify that the datagram's header fields and encrypted payload were not modified while the datagram was en route from source to destination.
- **Origin authentication.** When a host receives an IPsec datagram from a trusted source the host is assured that the source IP address in the datagram is the actual source of the datagram.

When two hosts have an IPsec session established between them, all TCP and UDP segments sent between them will be encrypted and authenticated. IPsec therefore provides blanket coverage, securing all communication between the two hosts for all network applications.

Routing Algorithms

We learned that when a packet arrives to a router, the router indexes a forwarding table and determines the link interface to which the packet is to be directed. We also learned that routing algorithms, operating in network routers, exchange and compute the information that is used to configure these forwarding tables.

The job of routing is to determine good paths (equivalently, routes), from senders to receivers, through the network of routers. Typically a host is attached directly to one router, the **default router** for the host (also called the **first-hop router** for the host). Whenever a host sends a packet,
the packet is transferred to its default router. We refer to the default router of the source host as the **source router** and the default router of the destination host as the **destination router**.

The problem of routing a packet from source host to destination host clearly boils down to the problem of routing the packet from source router to destination router, which is the focus of this section. The purpose of a routing algorithm is then simple: given a set of routers, with links connecting the routers, a routing algorithm finds a "good" path from source router to destination router. Typically, a good path is one that has the least cost.

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.
As shown in the below Figure an edge also has a value representing its cost. Typically, an edge's cost may reflect the physical length of the corresponding link 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, throughout we consider only undirected graphs (i.e., graphs whose edges do not have a direction), so that edge (x,y) is the

same as edge (*y*,*x*) and that *c*(*x*,*y*) = *c*(*y*,*x*). Also, a node *y* is said to be a **neighbour** of node *x* if (*x*,*y*) belongs to *E*.

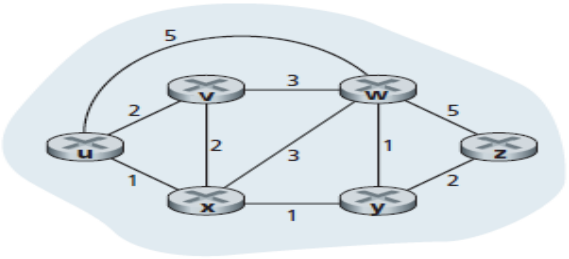


Figure Abstract graph model of a computer network

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 ($x1$, $x2$,..., xp) such that each of the pairs ($x1$,$x2$), ($x2$,$x3$),...,(xp-1,xp) are edges in E. The cost of a path ($x1$,$x2$,..., xp) is simply the sum of all the edge costs along the path, that is, $c(x1,x2) + c(x2,x3) + ... + c(xp\text{-}1,xp)$. 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.

Routing Algorithm Classification

1. Global routing algorithm V/S Decentralized routing algorithm:

Global routing algorithm :

- ✓ It 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 key distinguishing feature here, however, is that a global algorithm has complete information about connectivity and link costs.
- ✓ In practice, 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.

Decentralized routing algorithm:

- ✓ The calculation of the least-cost path is carried out in an iterative, distributed manner. 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 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.

2. Static V/S Dynamic routing algorithms

Static routing algorithms

routes change very slowly over time, often as a result of human intervention (for example, a human manually editing a router's forwarding table).

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 oscillation in routes.

3. Load-sensitive algorithm V/S Load-insensitive

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.

Load-insensitive algorithm

link's cost does not explicitly reflect its current (or recent past) level of congestion.
Today's Internet routing algorithms (such as RIP, OSPF, and BGP) are **load-insensitive**, as a

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.
The link-state routing algorithm we present below is known as **Dijkstra's algorithm,** named after its inventor.
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:

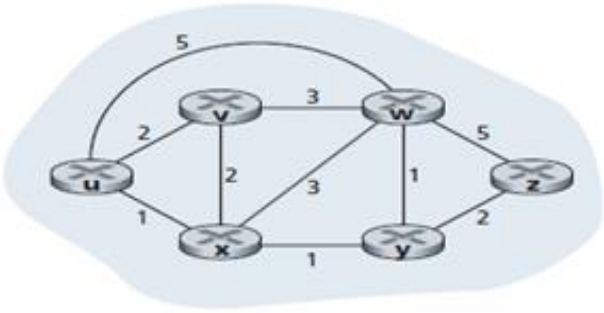
- ***D*(*v*):** cost of the least-cost path from the source node to destination *v* as of this iteration of the algorithm.
- ***p*(*v*):** previous node (neighbor of *v*) along the current least-cost path from the source to *v.*

- ***N*:** subset of nodes; *v* is in *N*_ if the least-cost path from the source to *v* is definitively known.

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
```

Example: consider a network and compute the least-cost paths from ***u*** to all possible destinations.



Solution:

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					

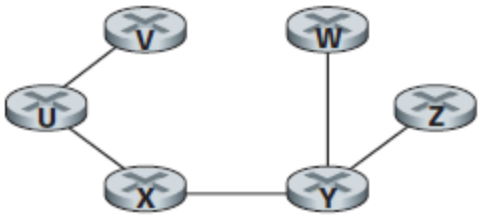
Let's consider the few first steps in detail.

• **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*.

• **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*.. 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 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.



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

Complexity: 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)**

Example problems: follow the class notes

Distance-Vector (DV) Routing Algorithm

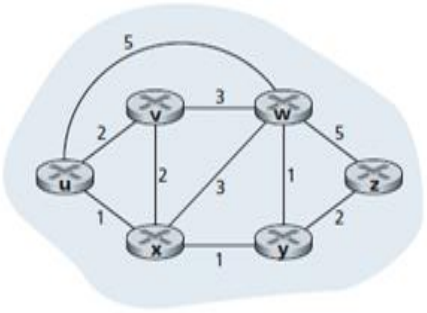
Whereas the LS algorithm is an algorithm using global information, the **distancevector** (**DV**) algorithm is iterative, asynchronous, and distributed.

- ✓ 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.

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,

$$d_x(y) = \min_v\{c(x,v) + d_v(y)\} \quad \text{---------- 4.1}$$

where
minv*:* in the equation is taken over all of *x*'s neighbors.
Dx(y):minimum distance from x to y
C(x,v):cost from x to v
D(v,y):distance from v to y



let's check it for source node *u* and destination node *z* in this figure. 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 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.

The Bellman-Ford equation is not just an intellectual curiosity. It actually has significant practical importance. In particular, 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 4.1. 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 in *N*.
Let $\mathbf{D}x = [D_x(y): y \text{ in } N]$ 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, $\mathbf{D}x = [D_x(y): y \text{ in } N]$, containing *x*'s estimate of its cost to all destinations, *y*, in *N*
• The distance vectors of each of its neighbors, that is, $\mathbf{D}v = [D_v(y): y \text{ in } N]$ 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 *v*, it saves *v*'s distance vector, and then uses the Bellman-Ford equation to update its own distance vector as follows: $D_x(y) \leftarrow \min_v\{c(x,v) + D_v(y)\}$ for each node *y* 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.

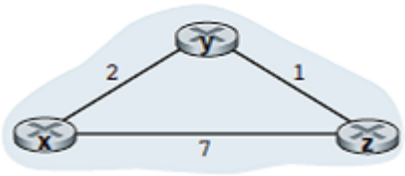
Distance-Vector (DV) Algorithm

At each node, x:

```
1  Initialization:
2     for all destinations y in N:
3        Dx(y) = c(x,y)    /* if y is not a neighbor then c(x,y) = ∞ */
4     for each neighbor w
5        Dw(y) = ? for all destinations y in N
6     for each neighbor w
7        send distance vector Dx = [Dx(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       Dx(y) = minv{c(x,v) + Dv(y)}
15
16    if Dx(y) changed for any destination y
17       send distance vector Dx = [Dx(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.

Example: consider a network with simple three node

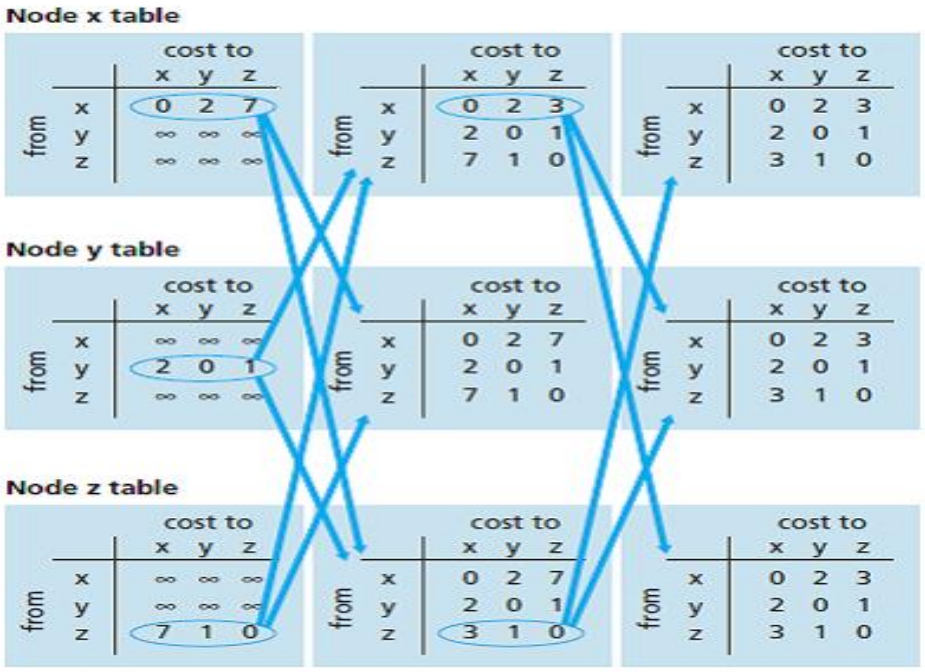


Solution:

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.

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 $\mathbf{D}x = [Dx(x), Dx(y), Dx(z)] = [0, 2, 7]$. 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 above figure 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

$\boldsymbol{Dx(x) = 0}$

$\boldsymbol{Dx(y) = \min\{c(x,y) + Dy(y), c(x,z) + Dz(y)\} = \min\{2 + 0, 7 + 1\} = 2}$

$\boldsymbol{Dx(z) = \min\{c(x,y) + Dy(z), c(x,z) + Dz(z)\} = \min\{2 + 1, 7 + 0\} = 3}$

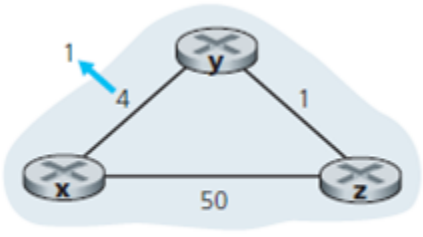
The second column therefore displays, for each node, the node's new distance vector along with distance vectors just received from its neighbors.

Note, for example,that node *x*'s estimate for the least cost to node *z*, $Dx(z)$, has changed from 7 to 3. After the nodes recompute their distance vectors, they again send their updated distance vectors to their neighbours (if there has been a change).

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 it updates its distance vector and, if there's a change in the cost of the least-cost path, informs its neighbors of its new distance vector.

- Figure 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:

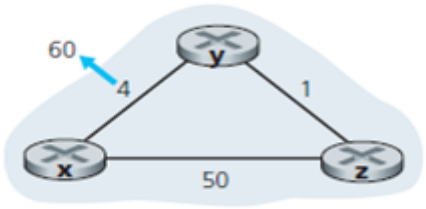
• **At time *t*0**, *y* detects the link-cost change (the cost has changed from 4 to 1), updates its distance vector, and informs its neighbors of this change since its distance vector has changed.

• **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.

❖ **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



1. Before the link cost changes, ***Dy*(*x*) = 4, *Dy*(*z*) = 1, *Dz*(*y*) = 1, and *Dz*(*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 *Dy*(*x*) = min{*c*(*y*,*x*) + *Dx*(*x*), *c*(*y*,*z*) + *Dz*(*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*.

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 *Dz*(*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.

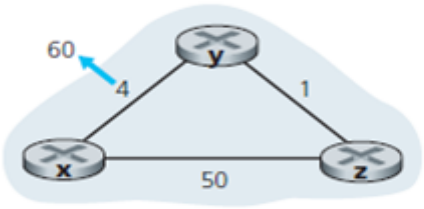
4. In a similar manner, after receiving *z*'s new distance vector, *y* determines *Dy*(*x*) = 8 and sends *z* its distance vector. *z* then determines *Dz*(*x*) = 9 and sends *y* its distance vector, and so on.
How long will the process continue? You should convince yourself that the loop will persist 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. the problem we have seen is sometimes referred to as the **count to-infinity problem.**

Solution to 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.



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 $D_y(x) = 51$ in truth).

A Comparison of LS and DV Routing Algorithms

The DV and LS algorithms take complementary approaches towards 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.
- ✓ In the LS algorithm, each node talks 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. 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 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.

Hierarchical Routing

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 a bit simplistic for at least two important reasons:

• **Scale.** As the number of routers becomes large, the overhead involved in computing, storing, and communicating routing information (for example, LS updates or least-cost path changes) becomes prohibitive. Today's public Internet consists of hundreds of millions of hosts. Storing routing information at each of these hosts would clearly require enormous amounts of memory. Clearly, something must be done to reduce the complexity of route computation in networks as large as the public Internet.

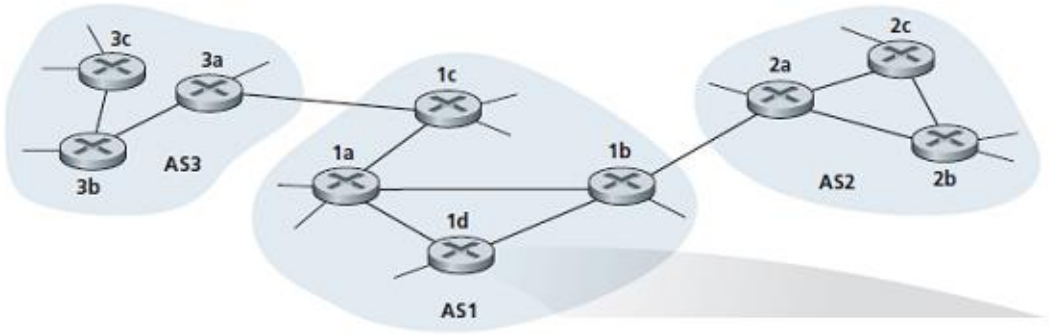
• **Administrative autonomy.** Although researchers tend to ignore issues such as a company's desire to run its routers as it pleases (for example, to run whatever routing algorithm it chooses) or to hide aspects of its network's internal organization from the outside, these are important considerations. Ideally, an organization should be able to run 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 typically under the same administrative control (e.g., operated by the same ISP or belonging to the same company network).

Routers within the same AS all run the same routing algorithm (for example, an LS or DV algorithm) and have information about each other.

The routing algorithm running within an autonomous system is called an **intra autonomous system routing protocol**. It will be necessary, of course, to connect ASs to each other, and thus one or more of the routers in an AS will have the added task of being responsible for forwarding packets to destinations outside the AS; these routers are called **gateway routers**.

Figure below provides a simple example with three ASs: AS1, AS2, and AS3.



In this figure, the heavy lines represent direct link connections between pairs of routers. The thinner lines hanging from the routers represent subnets that are directly connected to the routers.

AS1 has four routers—1a, 1b, 1c, and 1d— which run the intra-AS routing protocol used within AS1. Thus, each of these four routers knows how to forward packets along the optimal path to any destination within AS1.

Similarly, autonomous systems AS2 and AS3 each have three routers. Note that the intra-AS routing protocols running in AS1, AS2, and AS3 need not be the same.

Also note that the **routers 1b, 1c, 2a, and 3a are all gateway routers.**

How does a router, within some AS, know how to route a packet to a destination that is outside the AS?

It's easy to answer this question if the AS has only one gateway router that connects to only one other AS. In this case, because the AS's intra-AS routing algorithm has determined the least-cost path from each internal router to the gateway router, each internal router knows how it should forward the packet. The gateway router, upon receiving the packet, forwards the packet on the one link that leads outside the AS. The AS on the other side of the link then takes over the responsibility of routing the packet to its ultimate destination.

Example:

suppose router 2b in Figure 4.32 receives a packet whose destination is outside of AS2. Router 2b will then forward the packet to either router 2a or 2c, as specified by router 2b's forwarding table, which was configured by AS2's intra-AS routing protocol. The packet will eventually arrive to the gateway router 2a, which will forward the packet to 1b. Once the packet has left 2a, AS2's job is done with this one packet.

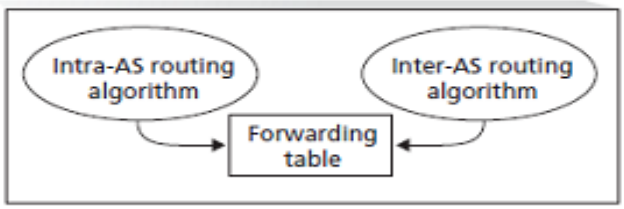
What if the source AS has two or more links (through two or more gateway routers) that lead outside the AS?

Then the problem of knowing where to forward the packet becomes significantly more challenging.

Example:

consider a router in AS1 and suppose it receives a packet whose destination is outside the AS. The router should clearly forward the packet to one of its two gateway routers, 1b or 1c, but which one? To solve this problem, AS1 needs (1) to learn which destinations are reachable via AS2 and which destinations are reachable via AS3, and (2) to propagate this reachability information to all the routers within AS1, so that each router can configure its forwarding table to handle external-AS destinations. These two tasks—obtaining reachability information from neighboring ASs and propagating the reachability information to all routers internal to the AS—are handled by the **inter-AS routing protocol**.

The two communicating ASs must run the same inter-AS routing protocol. In fact, in the Internet all ASs run the same inter-AS routing protocol, each router receives information from an intra-AS routing protocol and an inter-AS routing protocol, and uses the information from both protocols to configure its forwarding table.

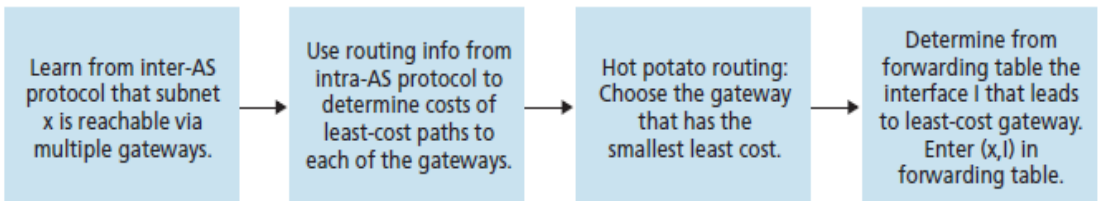


As an example, consider a subnet *x* (identified by its CIDRized address), and suppose that AS1 learns from the inter-AS routing protocol that subnet *x* is reachable from AS3 but is *not* reachable from AS2. AS1 then propagates this information to all of its routers. When router 1d learns that subnet *x* is reachable from AS3, and hence from gateway 1c, it then determines, from the information provided by the intra-AS routing protocol, the router interface that is on the least-cost path from router 1d to gateway router 1c. Say this is interface *I*. The router 1d can then put the
entry (*x*, *I*) into its forwarding table.

Following up on the previous example, now suppose that AS2 and AS3 connect to other ASs, which are not shown in the diagram. Also suppose that AS1 learns from the inter-AS routing protocol that subnet *x* is reachable both from AS2, via gateway 1b, and from AS3, via gateway 1c. AS1 would then propagate this information to all its routers, including router 1d. In order to configure its forwarding table, router 1d would have to determine to which gateway router, 1b or 1c, it should direct packets that are destined for subnet *x*. One approach, which is often employed in practice, is to use **hot-potato routing**.

In hot-potato routing, the AS gets rid of the packet (the hot potato) as quickly as possible (more precisely, as inexpensively as possible). This is done by having a router send the packet to the
gateway router that has the smallest router-to-gateway cost among all gateways with a path to the destination.

In the context of the current example, hot-potato routing, running in 1d, would use information from the intra-AS routing protocol to determine the path costs to 1b and 1c, and then choose the path with the least cost. Once this path is chosen, router 1d adds an entry for subnet *x* in its forwarding table.
Figure below summarizes the actions taken at router 1d for adding the new entry for *x* to the forwarding table.



Routing in the Internet

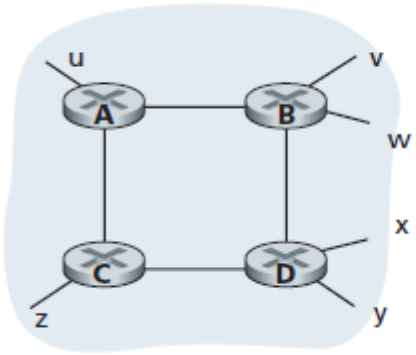
The Internet's routing protocols job is to determine the path taken by a datagram between source and destination.

Intra-AS Routing in the Internet: RIP

An intra-AS routing protocol is used to determine how routing is performed within an autonomous system (AS). Intra-AS routing protocols are also known as **interior gateway protocols**.
Two routing protocols have been used extensively for routing within an autonomous system in the Internet: the **Routing Information Protocol (RIP)** and **Open Shortest Path First (OSPF)**.
A routing protocol closely related to OSPF is the **IS-IS** protocol.

RIP was one of the earliest intra-AS Internet routing protocols and is still in widespread use today.
RIP is a distance-vector protocol that operates in a manner very close to the idealized DVprotocol.
The version of RIP specified in RFC 1058 uses hop count as a cost metric; that is, costs are actually from source router to a destination subnet.
RIP uses the term **hop,** which is the number of subnets traversed along the shortest path from source router to destination subnet, including the destination subnet.

Figure below illustrates an AS with six leaf subnets.

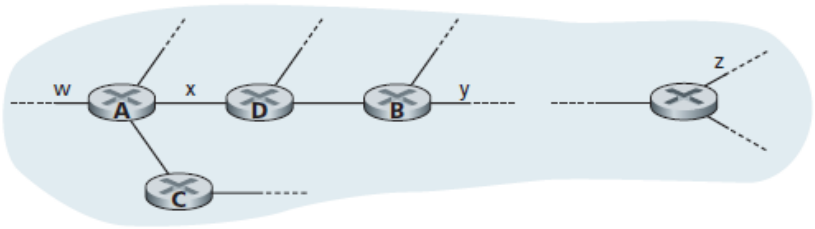


Destination	Hops
u	1
v	2
w	2
x	3
y	3
z	2

The table in the figure indicates the number of hops from the source A to each of the leaf subnets. The maximum cost of a path is limited to 15, thus limiting the use of RIP to autonomous systems that are fewer than 15 hops in diameter. Recall that in DV protocols, neighboring routers exchange distance vectors with each other. The distance vector for any one router is the current estimate of the shortest path distances from that router to the subnets in the AS.

In RIP, routing updates are exchanged between neighbors approximately every 30 seconds using a **RIP response message**. The response message sent by a router or host contains a list of up to 25 destination subnets within the AS, as well as the sender's distance to each of those subnets. Response messages are also known as **RIP advertisements**.

Let's take a look at a simple example of how RIP advertisements work. Consider the portion of an AS shown in Figure



In this figure, lines connecting the routers denote subnets. Only selected routers **(*A, B, C,* and *D*)** and subnets **(*w, x, y,* and *z*)** are labelled. Dotted lines indicate that the AS continues on; thus this autonomous system has many more routers and links than are shown. Each router maintains a RIP table known as a **routing table**.

A router's routing table includes both the router's distance vector and the router's forwarding table. Figure 4.36 shows the routing table for router *D*.

Destination Subnet	Next Router	Number of Hops to Destination
w	A	2
y	B	2
z	B	7
x	—	1
....	....	....

Note that the routing table has three columns.

- ✓ **first column** is for the destination subnet
- ✓ **second column** indicates the identity of the next router along the shortest path to the destination subnet
- ✓ **third** column indicates the number of hops (that is, the number of subnets that have to be traversed, including the destination subnet) to get to the destination subnet along the shortest path.

For this example, the table indicates that to send a datagram from router *D* to destination subnet *w*, the datagram should first be forwarded to neighbouring router *A;* the table also indicates that destination subnet *w* is two hops away along the shortest path. Similarly, the table indicates that subnet *z* is seven hops away via router *B*.

Now suppose that 30 seconds later, router *D* receives from router *A* the advertisement shown in Figure

Destination Subnet	Next Router	Number of Hops to Destination
z	C	4
w	—	1
x	—	1
....	....	....

Fig: advertisement from router A

This information indicates, in particular, that subnet *z* is only four hops away from router *A*. Router *D*, upon receiving this advertisement, merges the advertisement with the old routing table. In particular, router *D* learns that there is now a path through router *A* to subnet *z* that is shorter than the path through

router *B*. Thus, router *D* updates its routing table to account for the shorter shortest path, as shown in Figure

Destination Subnet	Next Router	Number of Hops to Destination
w	A	2
y	B	2
z	A	5
....	....	....

RIP routers exchange advertisements approximately every 30 seconds. If a router does not hear from its neighbour at least once every 180 seconds, that neighbour is considered to be no longer reachable; that is, either the neighbour has died or the connecting link has gone down. When this happens, RIP modifies the local routing table and then propagates this information by sending advertisements to its neighbouring routers (the ones that are still reachable).
A router can also request information about its neighbour's cost to a given destination using RIP's request message. Routers send RIP request and response messages to each other over UDP using port number 520.

Figure 4.39 sketches how RIP is typically implemented in a UNIX system, for example, a UNIX workstation serving as a router. A process called **routed** executes RIP, that is, maintains routing information and exchanges messages with ***routed*** processes running in neighbouring routers. Because RIP is implemented as an application-layer process, it can send and receive messages over a standard socket and use a standard transport protocol. As shown, RIP is implemented as an application-layer protocol running over UDP.

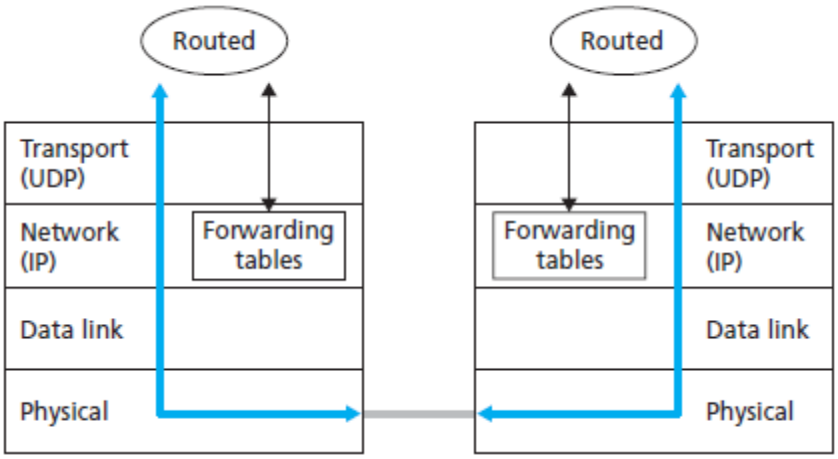


Fig: Implementation of RIP as the routed daemon

Intra-AS Routing in the Internet: OSPF

Like RIP, OSPF routing is widely used for intra-AS routing in the Internet. At its heart, however, OSPF is a link-state protocol that uses flooding of link-state information and a Dijkstra least-cost path algorithm.

With OSPF, a router constructs a complete topological map (that is, a graph) of the entire autonomous system. The router then locally runs Dijkstra's shortest-path algorithm to determine a shortest-path tree to all *subnets*, with itself as the root node.

With OSPF, a router broadcasts routing information to *all* other routers in the autonomous system, not just to its neighbouring 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. 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 and allows an OSPF router to obtain a neighbouring 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 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**
  - ✓ **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. 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.

- **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) 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 routing domain:** Perhaps the most significant advance in OSPF is the ability to structure an autonomous system hierarchically.

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 nonborder 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.

Inter-AS Routing: BGP

Let's now examine how paths are determined for source-destination pairs that span multiple ASs. The **Border Gateway Protocol** version 4, specified in RFC 4271 It is commonly referred to as BGP4 or simply as **BGP**.

As an inter-AS routing protocol BGP provides each AS a means to
1. Obtain subnet reachability information from neighbouring ASs.
2. Propagate the reachability information to all routers internal to the AS.
3. Determine "good" routes to subnets based on the reachability information and on AS policy

Most importantly, 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 ASs in the Internet know about the subnet and how to get there.

BGP Basics

In BGP, pairs of routers exchange routing information over semi permanent TCP connections using port 179. There is typically one such BGP TCP connection for each link that directly connects two routers in two different Ass. Shown in below figure

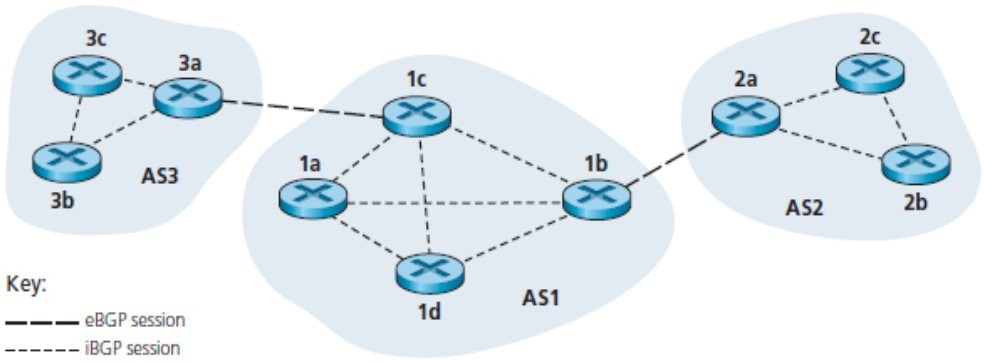


Fig: eBGP and iBGP sessions

Thus, in this Figure, there is a TCP connection between gateway routers 3a and 1c and another TCP connection between gateway routers 1b and 2a. There are also semi-permanent BGP TCP connections between routers within an AS.

In particular, Figure 4.40 displays a common configuration of one TCP connection for each pair of routers internal to an AS, creating a mesh of TCP connections within each AS. For each TCP connection, the two routers at the end of the connection are called **BGP peers**, and the TCP connection

along with all the BGP messages sent over the connection is called a **BGP session**. Furthermore, a BGP session that spans two Ass is called an **external BGP** (**eBGP**) **session**, and a BGP session between routers in the same AS is called an **internal BGP** (**iBGP**) **session**.
In Figure the eBGP sessions are shown with the long dashes; the iBGP sessions are shown with the short dashes.

BGP allows each AS to learn which destinations are reachable via its neighbouring ASs. In BGP, destinations are not hosts but instead are CIDRized **prefixes**, with each prefix representing a subnet or a collection of subnets. Thus, for example, suppose there are four subnets attached to **AS2: 138.16.64/24, 138.16.65/24, 138.16.66/24, and 138.16.67/24.** And AS3 attached 1 subnet **138.16.67/24** Then AS2 could aggregate the prefixes for these fourth subnets and use BGP to advertise the single prefix to **138.16.64/22** to AS1. because routers use longest-prefix matching for forwarding datagrams, AS3 could advertise to AS1 the more specific prefix **138.16.67/24** and AS2 could *still* advertise to AS1 the aggregated prefix **138.16.64/22.**

BGP would distribute prefix reachability information over the BGP sessions shown in Figure 4.40. As you might expect, using the **eBGP** session between the gateway routers 3a and 1c, AS3 sends AS1 the list of prefixes that are reachable from AS3; and AS1 sends AS3 the list of prefixes that are reachable from AS1. Similarly, AS1 and AS2 exchange prefix reachability information
through their gateway routers 1b and 2a.

Also as you may expect, when a gateway router (in any AS) receives eBGP-learned prefixes, the gateway router uses its iBGP sessions to distribute the prefixes to the other routers in the AS. Thus, all the routers in AS1 learn about AS3 prefixes, including the gateway router 1b. The gateway router 1b (in AS1) can therefore re-advertise AS3's prefixes to AS2. When a router
(gateway or not) learns about a new prefix, it creates an entry for the prefix in its forwarding table.

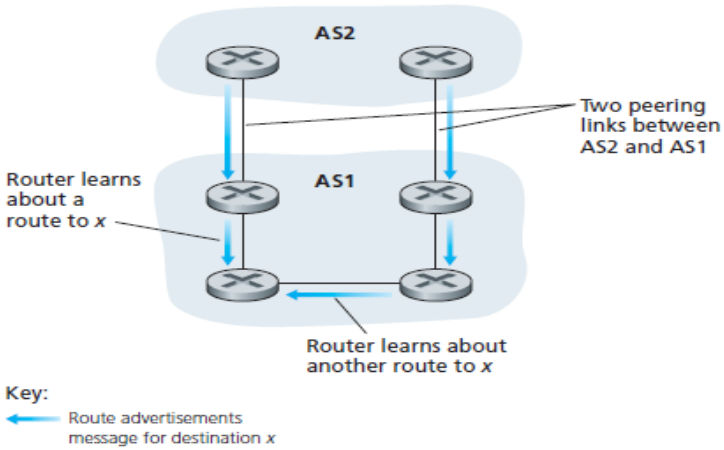
Path Attributes and BGP Routes:

In BGP, an autonomous system is identified by its globally unique **autonomous system number** (**ASN**) In particular, a so called stub AS that carries only traffic for which it is a source or destination will not typically have an ASN .
When a router advertises a prefix across a BGP session, it includes with the prefix a number of **BGP attributes**. In BGP jargon, a prefix along with its attributes is called a **route**. Thus, BGP peers advertise routes to each other.
Two of the more important attributes are **AS-PATH and NEXT-HOP**

- **AS-PATH:**
  - ✓ This attribute contains the ASs through which the advertisement for the prefix has passed. When a prefix is passed into an AS, the AS adds its ASN to the ASPATH attribute.
  - ✓ For example, consider Figure 4.40 and suppose that prefix 138.16.64/24 is first advertised from AS2 to AS1; if AS1 then advertises the prefix to AS3, **AS-PATH would be AS2 AS1**.
  - ✓ Routers use the AS-PATH attribute to detect and prevent looping advertisements; specifically, if a router sees that its AS is contained in the path list, it will reject the advertisement.
- **NEXT-HOP:**
  - ✓ *The NEXT-HOP is the router interface that begins the AS-PATH*. To gain insight into this attribute, let's again refer to Figure 4.40. Consider what happens when the gateway router 3a in AS3 advertises a route to gateway router 1c in AS1 using **eBGP.** The route includes the advertised prefix, which we'll call *x*, and an AS-PATH to the prefix. This advertisement also includes the NEXT-HOP, which is the IP address of the router 3a interface that leads to 1c.

- ✓ Now consider what happens when router 1d learns about this route from iBGP. After learning about this route to *x*, router 1d may want to forward packets to *x* along the route, that is, router 1d may want to include the entry (*x*, *l*) in its forwarding table, where *l* is its interface that begins the least-cost path from 1d towards the gateway router 1c.

In this Figure illustrates another situation where the NEXT-HOP is needed.



In this figure, AS1 and AS2 are connected by two peering links. A router in AS1 could learn about two different routes to the same **prefix *x*.** These two routes could have the same AS-PATH to ***x*,**
But could have different NEXT-HOP values corresponding to the different peering links. Using the NEXT-HOP values and the intra-AS routing algorithm, the router can determine the cost of the path to each peering link, and then apply hot-potato routing to determine the appropriate interface.

BGP Route Selection

If there are two or more routes to the same prefix, then BGP sequentially invokes the following elimination rules until one route remains:

- ✓ Routes are assigned a local preference value as one of their attributes. The local preference of a route could have been set by the router or could have been learned by another router in the same AS. This is a policy decision that is left up to the AS's network administrator The routes with the highest local preference values are selected.

- ✓ From the remaining routes (all with the same 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.

- ✓ From the remaining routes (all with the same local preference value and the same AS-PATH length), the route with the closest NEXT-HOP router is selected. Here, closest means the router for which the cost of the least-cost path, determined by the intra-AS algorithm, is the smallest. this process is called **hot-potato routing.**

- ✓ If more than one route still remains, the router uses BGP identifiers to select the route

Routing Policy

Let's illustrate some of the basic concepts of BGP routing policy with a simple example.

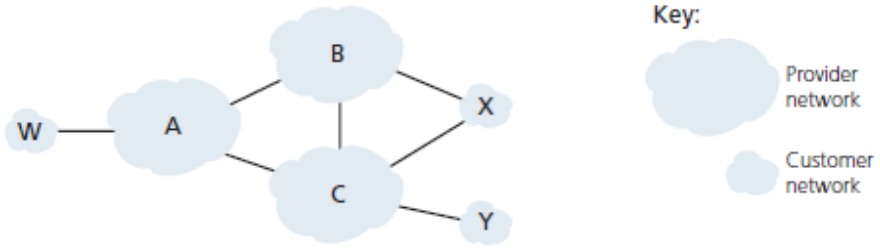


Fig: A simple BGP scenario

Figure 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 assume that autonomous systems W**, X, and Y**are stub networks and that **A, B, and C are backbone provider networks**.

We'll also assume that A, B, and C, all peer with each other, and provide full BGP information to their customer networks. All traffic entering a **stub network** must be destined for that network, and all traffic leaving a stub network must have originated in that network. W and Y are clearly stub networks. X is a **multi-homed stub network,** since it is connected to the rest of the network via two different providers.

However, like W and Y, X itself must be the source/destination of all traffic leaving/entering X.
But how will this stub network behaviour 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 a stub 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 BAW 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?

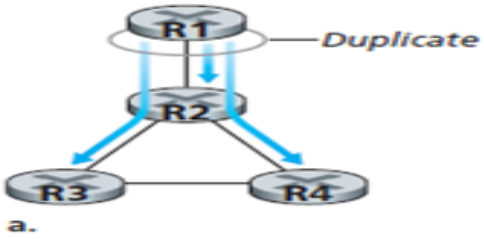
If it does so, then C could route traffic to W via CBAW. If A, B, and C are all backbone providers, than B might rightly feel that it should not have to shoulder the burden (and cost!) of carrying transit traffic between A and C. B might rightly feel that it is A's and C's job (and cost!) to make sure 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.

Broadcast and Multicast Routing

Broadcast routing, the network layer provides a service of delivering a packet sent from a source node to all other nodes in the network.
Multicast routing enables a single source node to send a copy of a packet to a subset of the other network nodes.

Broadcast Routing Algorithms

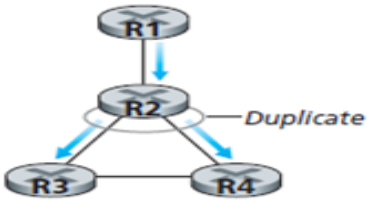
Perhaps the most straightforward way to accomplish broadcast communication is for the sending node to send a separate copy of the packet to each destination, as shown in Figure 4.43(a).



a.

Given *N* destination nodes, the source node simply makes *N* copies of the packet, addresses each copy to a different destination, and then transmits the *N* copies to the *N* destinations using unicast routing.
This **N-way unicast** approach to broadcasting is simple—no new network-layer routing protocol, Packet-duplication, or forwarding functionality is needed. There are, however, several drawbacks to this approach. The first drawback is its inefficiency. If the source node is connected to the rest of the network via a single link, then *N* separate copies of the (same) packet will traverse this single link.

It would clearly be more efficient to send only a single copy of a packet over this first hop and then have the node at the other end of the first hop make and forward any additional needed copies. That is, it would be more efficient for the network nodes themselves (rather than just the source node) to create duplicate copies of a packet. For example, in Figure 4.43(b), only a single copy of a packet traverses the R1-R2 link. That packet is then duplicated at R2, with a single copy being sent over links R2-R3 and R2-R4.



An implicit assumption of *N*-way-unicast is that broadcast recipients, and their addresses, are known to the sender. But how is this information obtained? Most likely, additional protocol mechanisms (such as a broadcast membership or destination-registration protocol) would be required. This would add more overhead and, importantly, additional complexity to a protocol that had initially seemed quite simple.

A final drawback of *N*-way-unicast relates to the purposes for which broadcast is to be used. Given the several drawbacks of *N*-way-unicast broadcast, approaches in which the network nodes themselves play

an active role in packet duplication, packet forwarding, and computation of the broadcast routes are clearly of interest.

Uncontrolled Flooding

The most obvious technique for achieving broadcast is a **flooding** approach in which the source node sends a copy of the packet to all of its neighbors.

When a node receives a broadcast packet, it duplicates the packet and forwards it to all of its neighbors (except the neighbor from which it received the packet). Clearly, if the graph is connected, this scheme will eventually deliver a copy of the broadcast packet to all nodes in the graph. Although this scheme is simple and elegant, it has a fatal flaw

If the graph has cycles, then one or more copies of each broadcast packet will cycle indefinitely.
For example, in this Figure

R2 will flood to R3, R3 will flood to R4, R4 will flood to R2, and R2 will flood (again!) to R3, and so on. This simple scenario results in the endless cycling of two broadcast packets, one clockwise, and one counter clockwise.

But there can be an even more calamitous fatal flaw: When a node is connected to more than two other nodes, it will create and forward multiple copies of the broadcast packet, each of which will create multiple copies of itself (at other nodes with more than two neighbours), and so on.

This **broadcast storm**, resulting from the endless multiplication of broadcast packets, would eventually result in so many broadcast packets being created that the network would be rendered useless.

Controlled Flooding

The key to avoiding a broadcast storm is for a node to judiciously choose when to flood a packet and (e.g., if it has already received and flooded an earlier copy of a packet) when not to flood a packet.
In practice, this can be done in one of several ways.

1. **In sequence-number-controlled flooding:**

A source node puts its address as well as a broadcast sequence number into a broadcast packet, and then sends the packet to all of its neighbours.
Each node maintains a list of the source address and sequence number of each broadcast packet it has already received, duplicated, and forwarded. When a node receives a broadcast packet, it first checks whether the packet is in this list. If so, the packet is dropped; if not, the packet is duplicated and forwarded to all the node's neighbors (except the node from
which the packet has just been received).

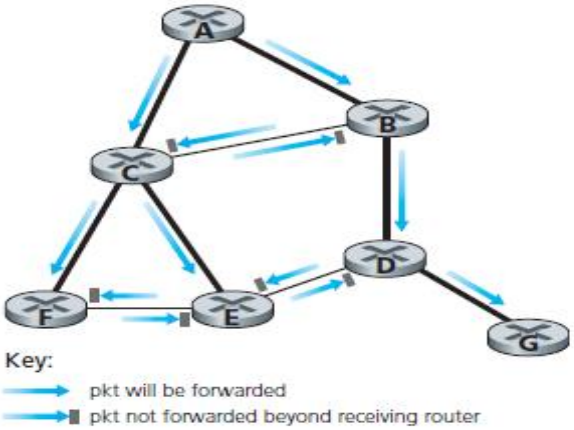
2. **Reverse path forwarding (RPF):**

Also sometimes referred to as reverse path broadcast (RPB). The idea behind RPF is simple, yet elegant. When a router receives a broadcast packet with a given source address, it transmits the packet on all of

its outgoing links only if the packet arrived on the link that is on its own shortest unicast path back to the source.

Otherwise, the router simply discards the incoming packet without forwarding it on any of its outgoing links. Such a packet can be dropped because the router knows it either will receive or has already received a copy of this packet on the link that is on its own shortest path back to the sender.
Note that RPF does not use unicast routing to actually deliver a packet to a destination, nor does it require that a router know the complete shortest path from itself to the source. RPF need only know the next neighbour on its unicast shortest path to the sender; it uses this neighbor's identity only to determine whether or not to flood a received broadcast packet.

Figure 4.44 illustrates RPF.



Suppose that the links drawn with thick lines represent the least-cost paths from the receivers to the source (*A*). Node *A* initially broadcasts a source-*A* packet to nodes *C* and *B*.

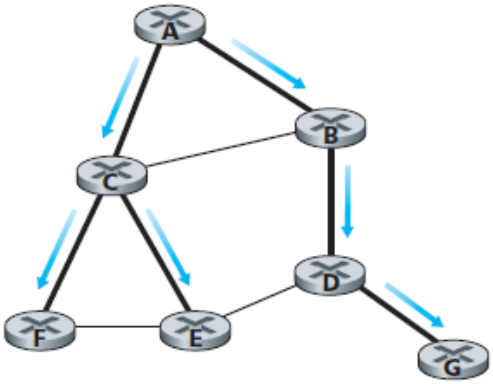
Node *B* will forward the source-*A* packet it has received from *A* (since *A* is on its least-cost path to *A*) to both *C* and *D*. *B* will ignore (drop, without forwarding) any source-*A* packets it receives from any other nodes (for example, from routers *C* or *D*).

Let us now consider node *C*, which will receive a source-*A* packet directly from *A* as well as from *B*. Since *B* is not on *C*'s own shortest path back to *A*, *C* will ignore any source-*A* packets it receives from *B*. On the other hand, when *C* receives a source-*A* packet directly from *A*, it will forward the packet to nodes *B*, *E*, and *F*.

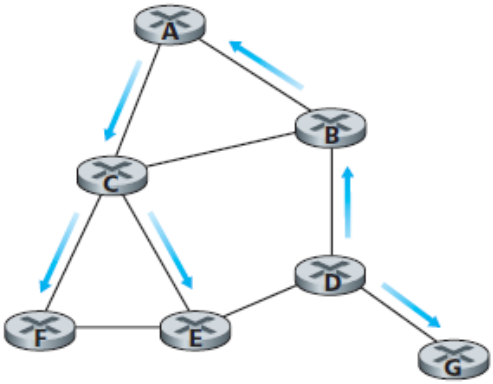
Spanning-Tree Broadcast

While sequence-number-controlled flooding and RPF avoid broadcast storms, they do not completely avoid the transmission of redundant broadcast packets. For example, in Figure 4.44, nodes *B*, *C*, *D*, *E*, and *F* receive either one or two redundant packets. Ideally, every node should receive only one copy of the broadcast packet.

Examining the tree consisting of the nodes connected by thick lines in the below figure you can see that if broadcast packets were forwarded only along links within this tree, each and every network node would receive exactly one copy of the broadcast packet exactly the solution we were looking for this tree is an example of a **spanning tree**s—a tree that contains each and every node in a graph.



a. Broadcast initiated at A



b. Broadcast initiated at D

More formally, a spanning tree of a graph $G = (N,E)$ is a graph $G\_ = (N,E\_)$ such that $E\_$ is a Subset of E, $G\_$ is connected, $G\_$ contains no cycles, and $G\_$ contains all the original nodes in G. If each link has an associated cost and the cost of a tree is the sum of the link costs, then a spanning tree whose cost is the minimum of all of the graph's spanning trees is called a **minimum spanning tree**.

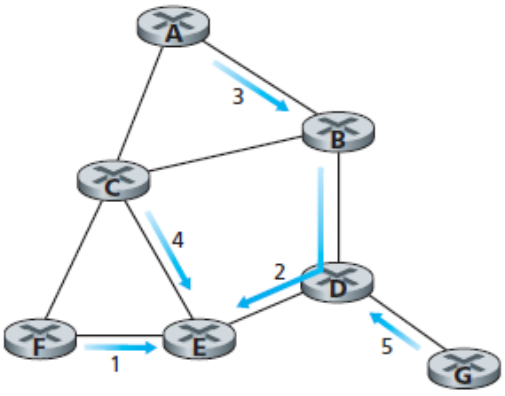
Thus, another approach to providing broadcast is for the network nodes to first construct a spanning tree. When a source node wants to send a broadcast packet, it sends the packet out on all of the incident links that belong to the spanning tree. A node receiving a broadcast packet then forwards the packet to all its neighbours in the spanning tree (except the neighbour from which it received the packet

We consider only one simple algorithm here. In the **center-based approach** to building a spanning tree, a center node is defined. Nodes then unicast tree-join messages addressed to the center node.

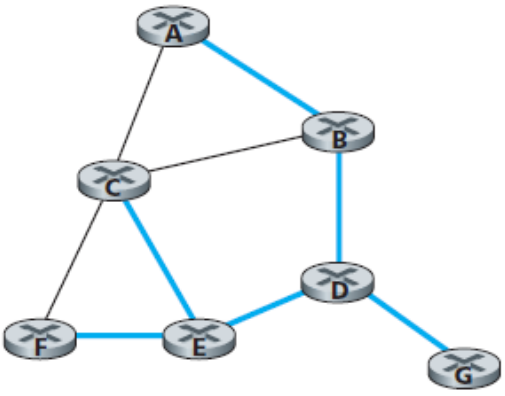
A tree-join message is forwarded using unicast routing toward the center until it either arrives at a node that already belongs to the spanning tree or arrives at the center.

In either case, the path that the tree-join message has followed defines the branch of the spanning tree between the edge node that initiated the tree-join message and the center. One can think of this new path as being grafted onto the existing spanning tree.

Figure illustrates the construction of a center-based spanning tree.



a. Stepwise construction of spanning tree



b. Constructed spanning tree

- ✓ Suppose that node *E* is selected as the center of the tree. Suppose that node *F* first joins the tree and forwards a tree-join message to *E*. The single link *EF* becomes the initial spanning tree.

- ✓ Node *B* then joins the spanning tree by sending its tree-join message to *E*. Suppose that the unicast path route to *E* from *B* is via *D*. In this case, the tree-join message results in the path *BDE* being grafted onto the spanning tree.

- ✓ Node *A* next joins the spanning group by forwarding its tree-join message towards *E*. If *A*'s unicast path to *E* is through *B*, then since *B* has already joined the spanning tree, the arrival of *A*'s tree-join message at *B* will result in the *AB* link being immediately grafted onto the spanning tree.

- ✓ Node *C* joins the spanning tree next by forwarding its tree-join message directly to *E*. Finally, because the unicast routing from *G* to *E* must be via node *D*, when *G* sends its tree-join message to *E*, the *GD* link is grafted onto the spanning tree at node *D*.

Multicast routing

We've seen in the previous section that with broadcast service, packets are delivered to each and every node in the network. In this section we turn our attention to **multicast** service, in which a multicast packet is delivered to only a *subset* of network nodes.

A number of emerging network applications requires the delivery of packets from one or more senders to a group of receivers.
These applications include

- ✓ **bulk data transfer** (for example, the transfer of a software upgrade from the software developer to users needing the upgrade),
- ✓ **streaming continuous media** (for example, the transfer of the audio, video, and text of a live lecture to a set of distributed lecture participants),
- ✓ **shared data applications** (for example, a whiteboard or teleconferencing application that is shared among many distributed participants),
- ✓ **data feeds** (for example, stock quotes),
- ✓ **Web cache updating, and interactive gaming** (for example, distributed interactive virtual environments or multiplayer games).

In multicast communication, we are immediately faced with two problems

1. How to identify the receivers of a multicast packet
2. How to address a packet sent to these receivers.

In the case of unicast communication, the IP address of the receiver (destination) is carried in each IP unicast datagram and identifies the single recipient; in the case of broadcast, *all* nodes need to receive the broadcast packet, so no destination addresses are needed. But in the case of multicast, we now have multiple receivers.

Does it make sense for each multicast packet to carry the IP addresses of all of the multiple recipients? While this approach might be workable with a small number of recipients, it would not scale well to the case of hundreds or thousands of receivers; the amount of addressing information in the datagram would swamp the amount of data actually carried in the packet's payload field.

A multicast packet is addressed using **address indirection**. that is, a single identifier is used for the group of receivers, and a copy of the packet that is addressed to the group using this single identifier is delivered to all of the multicast receivers associated with that group. In the Internet, the single identifier that represents a group of receivers is a class D multicast IP address. The group of receivers associated with a class D address is referred to as a **multicast group**.

The multicast group abstraction is illustrated in Figure 4.47.

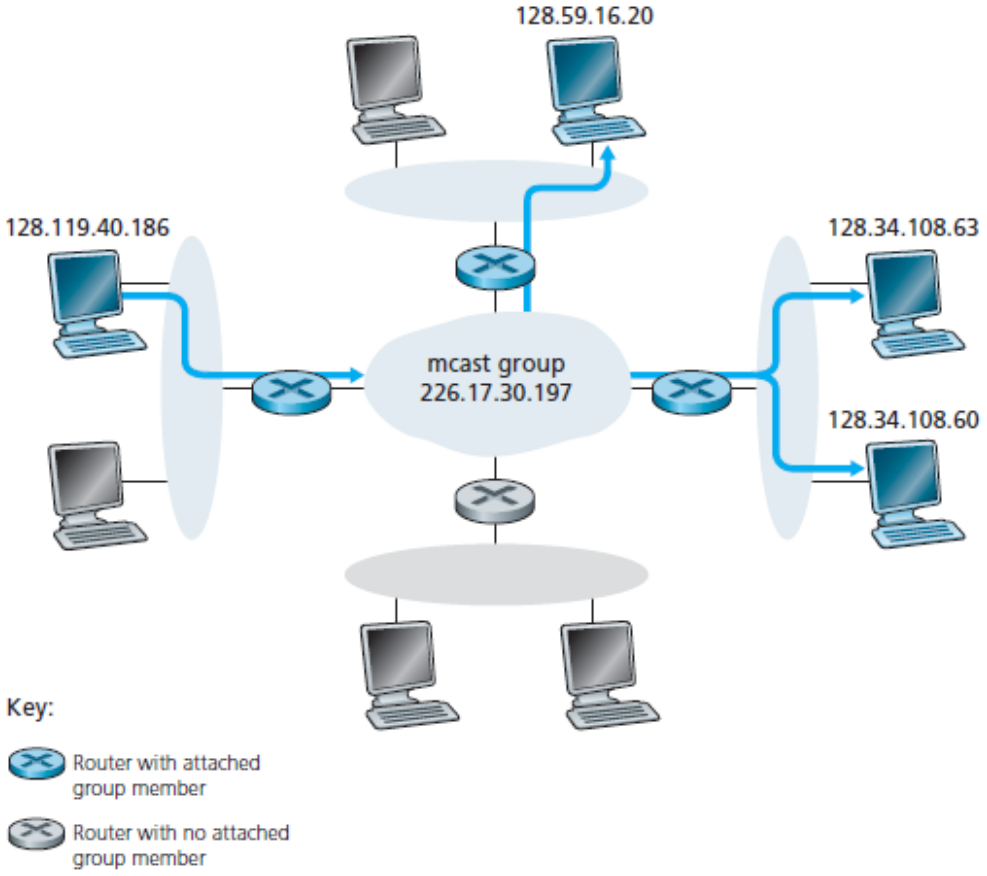


Figure The multicast group: A datagram addressed to the group is delivered to all members of the multicast group

Here, four hosts (shown in shaded color) are associated with the multicast group address of **226.17.30.197** and will receive all datagram's addressed to that multicast address. The difficulty that we must still address is the fact that each host has a unique IP unicast address that is completely independent of the address of the multicast group in which it is participating.

Internet Group Management Protocol(IGMP)

The IGMP protocol version 3 [RFC 3376] operates between a host and its directly attached router (informally, we can think of the directly attached router as the first hop router that a host would see on a path to any other host outside its own local network, or the last-hop router on any path to that host), as shown in Figure 4.48

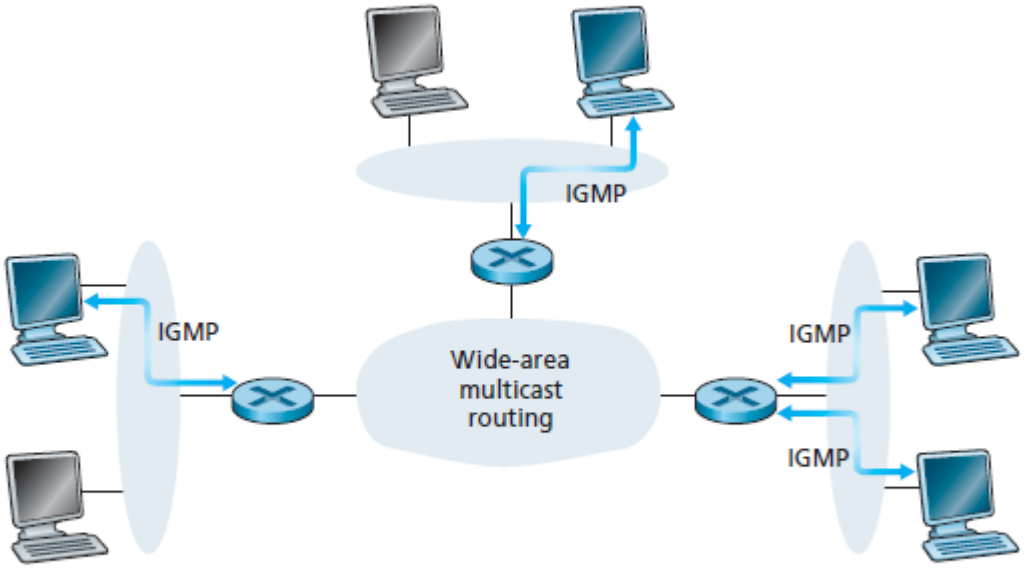


Figure 4.48 ♦ The two components of network-layer multicast in the Internet: IGMP and multicast routing protocols

Figure 4.48 shows three first-hop multicast routers, each connected to its attached hosts via one outgoing local interface. This local interface is attached to a LAN in this example, and while each LAN has multiple attached hosts, at most a few of these hosts will typically belong to a given multicast group at any given time.

IGMP provides the means for a host to inform its attached router that an application running on the host wants to join a specific multicast group. Given that the scope of IGMP interaction is limited to a host and its attached router, another protocol is clearly required to coordinate the multicast routers (including the attached routers) throughout the Internet, so that multicast datagrams are routed to their final destinations.

Network-layer multicast in the Internet thus consists of two complementary components: **IGMP and multicast routing protocols.**

IGMP has only three message types.

- The **membership_ query message** is sent by a router to all hosts on an attached interface (for example, to all hosts on a local area network) to determine the set of all multicast groups that have been joined by the hosts on that interface.
- Hosts respond to a membership_ query message with an IGMP **membership report message.** membership report messages can also be generated by a host when an application first joins a multicast group without waiting for a membership query message from the router.
- The final type of IGMP message is the **leave group message.** Interestingly, this message is optional. But if it is optional, how does a router detect when a host leaves the multicast group? The answer to this question is that the router *infers* that a host is no longer in the multicast group if it no longer responds to a membership_query message with the given group address.

Multicast Routing Algorithms

The **multicast routing problem** is illustrated in Figure 4.49. Hosts joined to the multicast group are shaded in color; their immediately attached router is also shaded in color. As shown in Figure 4.49, only a subset of routers (those with attached hosts that are joined to the multicast group) actually needs to receive the multicast traffic. In Figure 4.49, only routers *A, B, E,* and *F* need to receive the multicast traffic.

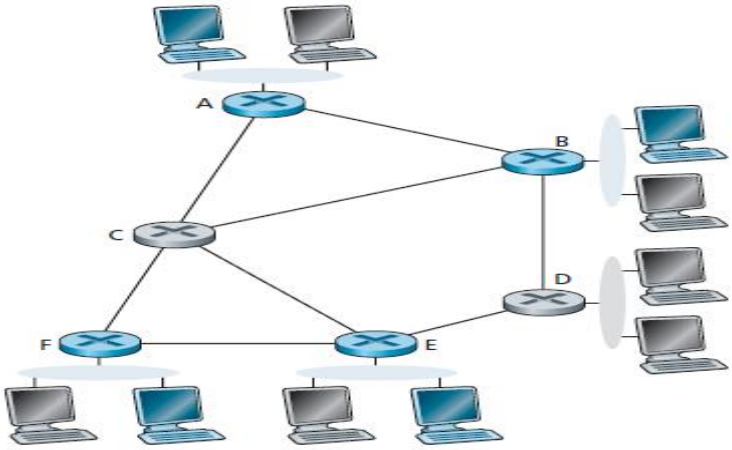


Figure 4.49 ♦ Multicast hosts, their attached routers, and other routers

Since none of the hosts attached to router *D* are joined to the multicast group and since router *C* has no attached hosts, neither *C* nor *D* needs to receive the multicast group traffic.

The goal of multicast routing, then, is to find a tree of links that connects all of the routers that have attached hosts belonging to the multicast group. Multicast packets will then be routed along this tree from the sender to all of the hosts belonging to the multicast tree. Of course, the tree may contain routers that do not have attached hosts belonging to the multicast group (for example, in Figure 4.49, it is impossible to connect routers *A, B, E,* and *F* in a tree without involving either router *C* or *D*).

In practice, two approaches have been adopted for determining the multicast routing tree,. The two approaches differ according to whether **a single group-shared tree** is used to distribute the traffic for *all* senders in the group, or whether a **source-specific routing tree** is constructed for each individual sender.

- **Multicast routing using a group-shared tree:** As in the case of spanning-tree broadcast, multicast routing over a group-shared tree is based on building a tree that includes all edge routers with attached hosts belonging to the multicast group.

  In practice, a center-based approach is used to construct the multicast routing tree, with edge routers with attached hosts belonging to the multicast group sending (via unicast) join messages addressed to the center node. As in the broadcast case, a join message is forwarded using unicast routing toward the center until it either arrives at a router that already belongs to the multicast tree or arrives at the center. All routers along the path that the join message follows will then forward received multicast packets to the edge router that initiated the multicast join..

- **Multicast routing using a source-based tree.** While group-shared tree multicast routing constructs a single, shared routing tree to route packets from *all* senders, the second approach constructs a multicast routing tree for *each* source in the multicast group. In practice, an RPF algorithm (with source node *x*) is used to construct a multicast forwarding tree for multicast datagrams originating at source *x*.

The RPF broadcast algorithm we studied earlier requires a bit of tweaking for use in multicast. To see why, consider router *D* in Figure 4.50. Under broadcast RPF, it would forward packets to router *G*, even though router *G* has no attached hosts that are joined to the multicast group.

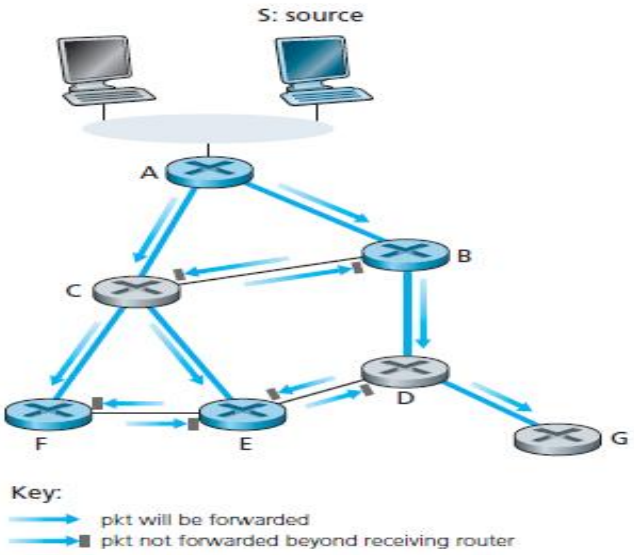


Figure 4.50 ♦ Reverse path forwarding, the multicast case

While this is not so bad for this case where *D* has only a single downstream router, *G*, imagine what would happen if there were thousands of routers downstream from *D*! Each of these thousands of routers would receive unwanted multicast packets.

The solution to the problem of receiving unwanted multicast packets under RPF is known as **pruning**. A multicast router that receives multicast packets and has no attached hosts joined to that group will send a prune message to its upstream router. If a router receives prune messages from each of its downstream routers, then it can forward a prune message upstream.

Multicast Routing in the Internet

The first multicast routing protocol used in the Internet was the **Distance-Vector Multicast Routing Protocol (DVMRP)** DVMRP implements source-based trees with reverse path forwarding and pruning.

DVMRP uses an RPF algorithm with pruning, as discussed above. Perhaps the most widely used Internet multicast routing protocol is the **Protocol-Independent Multicast (PIM) routing protocol**, which explicitly recognizes two multicast distribution scenarios.

- ✓ In **dense mode** multicast group members are densely located; that is, many or most of the routers in the area need to be involved in routing multicast datagrams. PIM dense mode is a flood-and-prune reverse path forwarding technique similar in spirit to DVMRP.
- ✓ In **sparse mode** the number of routers with attached group members is small with respect to the total number of routers; group members are widely dispersed. PIM sparse mode uses rendezvous points to set up the multicast distribution tree.