3. Network Layer 10/6/24

1. Network layer Design issues:-

→ The data transmitted in the network layer in the form of Packets.

→ It is concerned with getting Packets from the source all the way to the destination (with min. cost)

→ The following are the design issues related to the network layer. They are.

a. Store-and-Forward Packet Switching
b. Services Provided to the Transport Layer.
c. Implementation of Connection-less Services.
d. Implementation of Connection-Oriented Services
e. Comparision of Virtual-Circuits and Datagram Subnets.

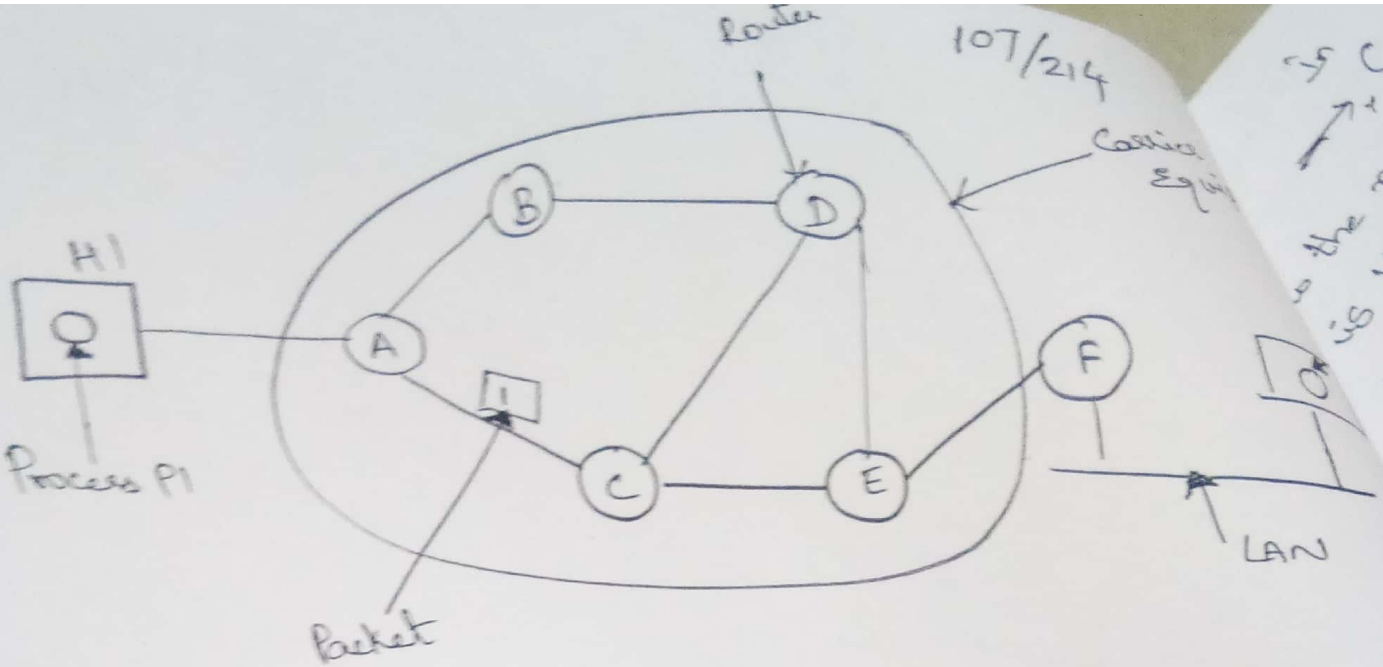
a. Store-and-Forward Packet Switching:-

→ A host with a Packet to send transmits it to the nearest router, either on its own LAN or Over a Point-to-Point link to the Carrier.

→ The Packet is then stored their Until it has fully arrived, so the checksum can be Verified.

→ Then it is forwarded to the next router along the Path until it reaches the destination host, where it is delivered.

→ This mechanism is called as Store-and-forward Packet Switching.



fig②: The Environment of network layer Protocol.

(b) Services Provided to the Transport layer:-

→ The network layer Provided Services to the Transport layer at the network layer / transport layer interface.

→ The network layer Services have been designed with the following goals in mind:

1. The Services Should be independent of router Technology.
2. The transport layer should be Shielded from the number, type, and topology of the routers Present.
3. The network addresses must be available to the transport layer should use a Uniform numbering plan, even across LAN's & WAN's.

→ In Connection-Oriented Service, first Connection is Established for a network, & then data is transfered.

→ In Connectionless Services, the Packet is transferred to the nearest router. Here, Packet Switching Technology is Used. [i.e Each data-unit is addressed & routed based on information Carried in Each Unit; rather than in the setup information is Prefixed].

→ Packet Switching is a digital networking Communication method that groups all transmitted data into Suitably Sized blocks, Called Packets.

→ A datagram is a basic transfer unit associated with a Packet-Switched network. The delivery, arrival time, & Order of arrival need not be guaranteed by the network.

→ A Virtual Circuit is a means of transferring data Over a Packet-Switched Computer network in a Such a way that it appears as though there is a dedicated physical layer link between the Source & destination End System of this data.

note:-

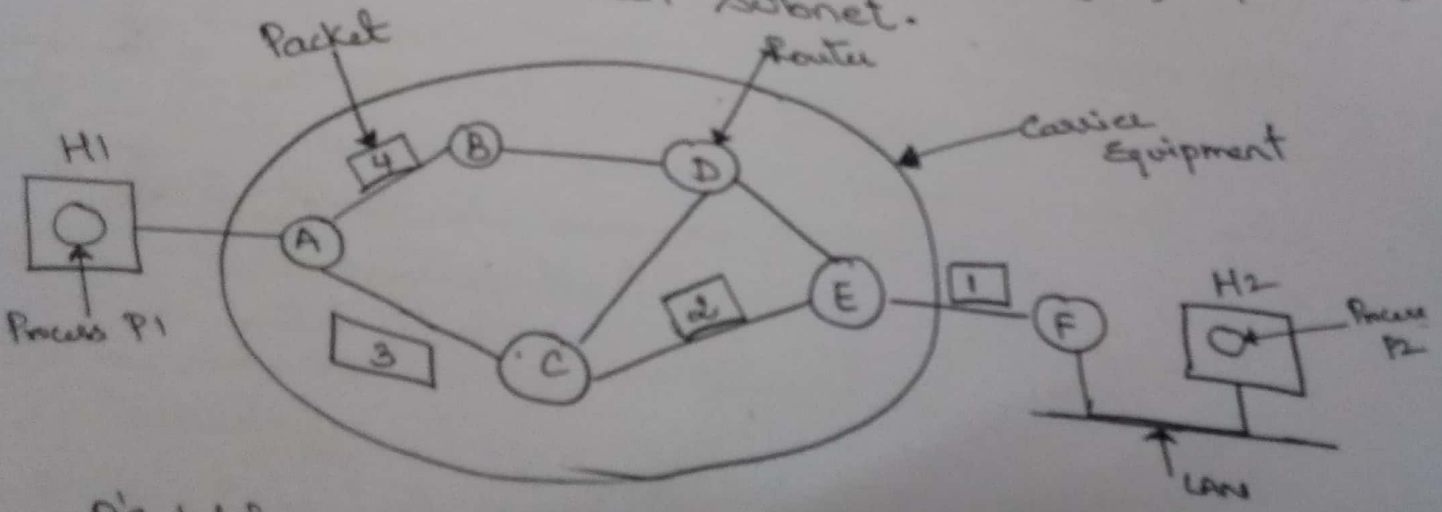
- n/w < Connection-Oriented / Connection-less
- routers job → Packet < moves / not move
- Telephone n/w → reliable – Connection-Oriented Service
- Connection-less Services → has grown Popularity.
  - Eg:- ATM & Internet

(c) Implementation of Connection-less Services :-

→ In Connection-less Services, Packets injected into the Subnet individually & routed independently of each other. (i.e No-advance setup needed)

→ So, Packets are frequently called datagrams and the Subnet is called a datagram Subnet.

→ If Connection-Oriented Savice is used, a Path from the Source router to the destination router must be Established before data Packets can be sent. This Connection is called Virtual Circuit (VC) & Subnet is called Virtual Circuit Subnet.



A's table initially

Dest. line	
A	-
B	B
C	C
D	B
E	C
F	C

Dest.	
A	-
B	B
C	C
D	B
E	B
F	B

C's table

A	A
B	A
C	-
D	D
E	F
F	E

E's table

A	C
B	D
C	C
D	D
E	-
F	F

fig (b) :- Routing with datagram Subnet

Implementation of Connection-Oriented Services:-

→ Connection-Oriented Services. Uses Virtual-Circuit Subnet.

→ The idea behind Virtual circuit is to avoid having to choose a new route for every Pack sent.

→ When a Connection is Established, a route from the source machine to destination machine is choosen as Part of Connection Set-up & stored in tables inside the routers.

→ When the Connection is released, the Virtual Circuit is also terminated. With Connection-Oriented Service, Each Packet Carries an identifier telling which Virtual Circuit it belongs to.

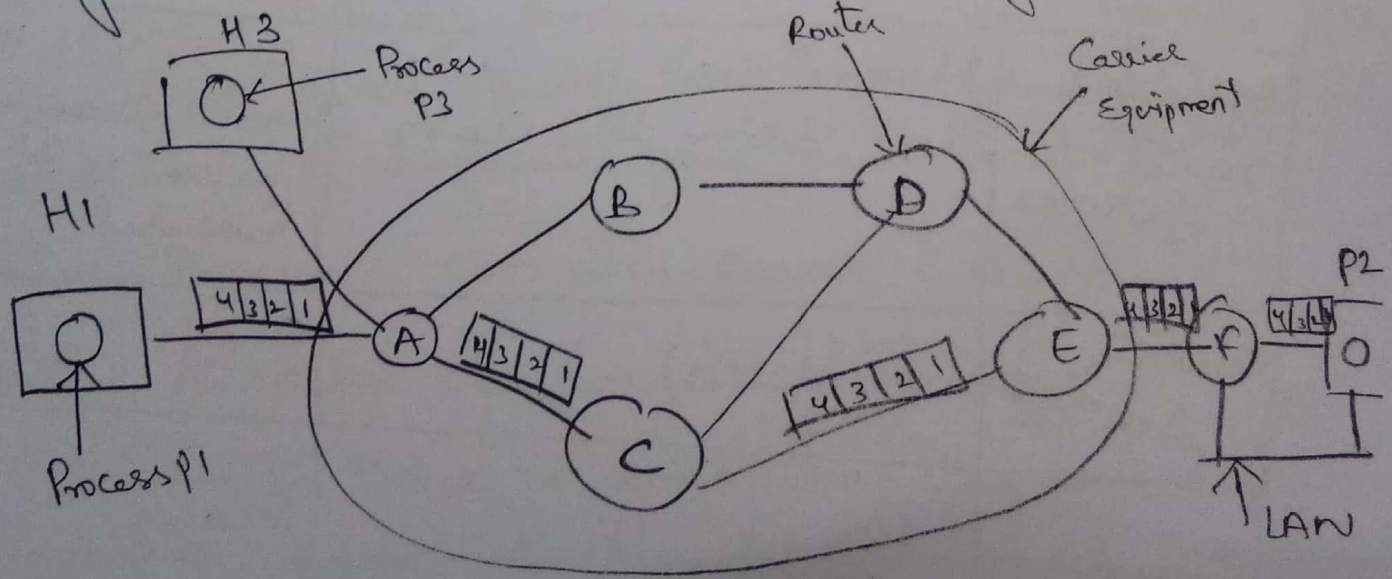


fig ⓒ :- Routing within a Virtual-Circuit Subnet

⑥ Comparision of Virtual-Circuit & Datagram

Issue	Datagram Subnet	Virtual-Circuit
Circuit Setup	not needed	Required
Addressing	Each Packet Contains the full Source & destination address	Each Vc requires router table space for Connection
State information	Router do not hold State info. about Connection	Each Packet Contains a short Vc number
Routing	Each Packet is routed independently	Route choosen when Vc is setup, all Packets follow it
Effect of router failures	None, Except for Packet lost during the Crash	All the Vc's that Passed through the failed route are terminated
Quality of Service	Difficult	Easy
Congestion Control	Difficult	Easy

CONGESTION.

112/214

Introduction:-

→ When too many Packets are Present in the Subnet, Performance degrades. This situation is called Congestion.

→ When the no. of Packets dumped into the Subnet by the hosts within its Carrying Capacity, they are all delivered & the number delivered is Proportional to the number sent.

no. of delivered Packets	$\propto$	no. of Packets sent.

→ Congestion can be occurred in different ways & Situations. Such as:-

① If routers have an infinite amount of memory, Congestion gets worse, not better, because by the time Packets gets to the front of the queue, they have already timed out (repeatedly) & duplicate have been sent.

② Slow Processors can also cause Congestion.

① General Principles of Congestion Control:-

→ In Order to avoid Congestion, ~~Control~~ few Approaches came into Existance known as Congestion Control.

→ Congestion Control approach leads to dividing all Solutions into two groups. They are

(a) Open loop

(b) closed loop.

② Open loop:-

→ (Open loop solutions attempt to solve the Problem by good design, in essence, to make sure it does not occur in the first place.)

→ In this method, Policies are used to Prevent the Congestion before it happens.

→ Congestion Control is handled either by the Source or by the destination.

→ The Various methods used for Open loop Congestion Control are:

* Retransmission policy:-

↳ Sender retransmits a Packet, if it feels that Packet it has sent is lost or Corrupted.

→ In general, retransmission may increase the Congestion in the network. But we need to implement good retransmission policy to Prevent Congestion.

* Window Policy:-

↳ To implement this policy, Selective ~~repeat~~ reject window method is used for Congestion Control.

→ Select rejects sends only the specific lost or damaged Packets.

* Acknowledgement Policy:-

→ The acknowledgement Policy imposed by the receiver may also affect Congestion.

→ If the receiver doesn't acknowledge every Packets it receives it may slow down the sender to help Prevent Congestion.

* Discarding Policy:-

→ A router may discard less sensitive Packets when Congestion is likely to happen.

→ Such a discarding policy may Prevent Congestion at the same time may not harm the integrity of the transmission.

* Admission Policy:-

→ Here, which is a quality-of-Service mechanism, can also Prevent Congestion in Virtual Circuit networks.

→ Switches in a flow first check the resource requirements of flow before admitting it to the n/w.

→ A router can deny Establishing a Virtual Circuit Connection if there is Congestion in the n/w or if there is a Possibility of future Congestion.

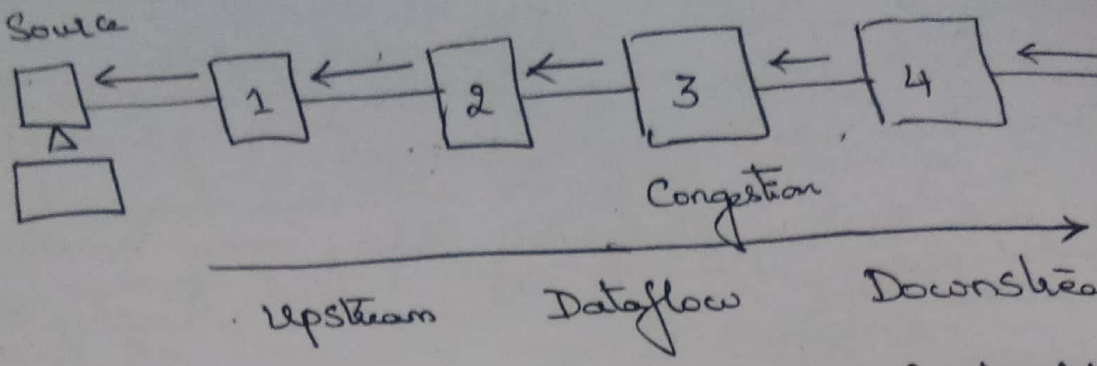
(b) Closed loop:-

→ closed loop Congestion Control mechanisms try to removes the Congestion after it happens.

→ Various methods used for closed loop are:-

1. Back Pressure:-

→ It is a node-to-node Congestion Control that starts with a node & Propagates, in the Opposite direction of data flow.



→ This technique can be applied only to Virtual Circuit networks. In such Circuits Each node knows the upstream node from which a data flow is coming.

→ In this method of Congestion Control, the Congested node stops receiving data from the immediate upstream node or nodes.

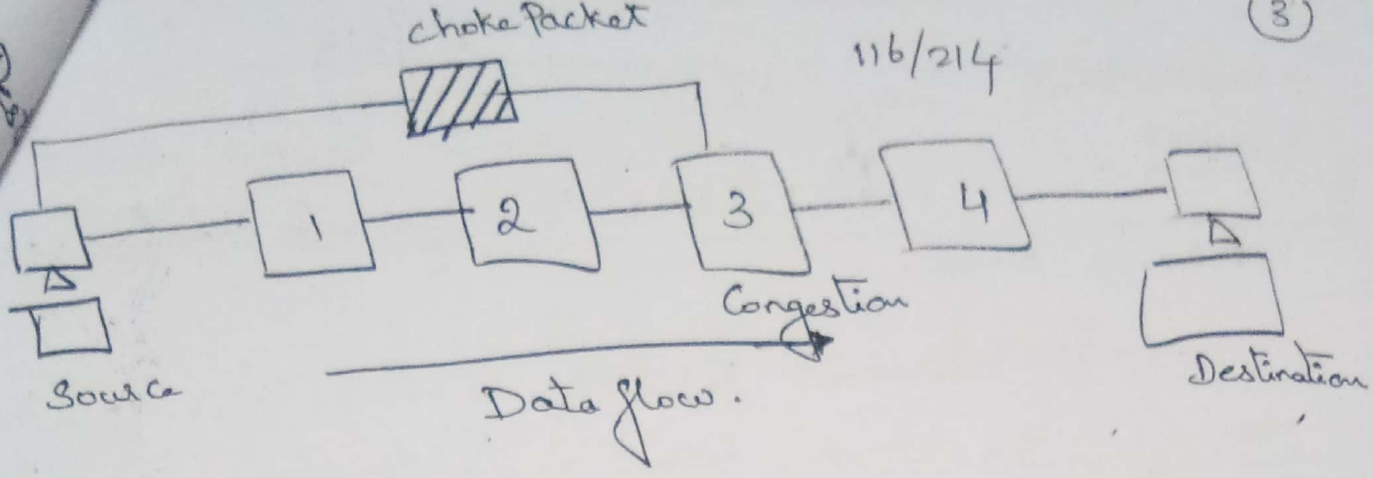
→ This may cause the upstream node or nodes to become Congested, & they in turn reject data from their upstream node or nodes.

② choke Packet:-

→ In this method, Congested router or node sends a special type of Packet called choke Packet from source to inform it about the Congestion.

→ Here, Congested node doesn't inform its upstream node about the Congestion as in backPressure method.

→ In this method, Congested node sends a warning directly to the source station i.e the intermediate nodes through which the packet has traveled or not warned.



(3) Implicit Signaling:-

→ Here, there is no Communication between the Congested node or nodes & the Source.

→ The Source guesses that there is Congestion Somewhere in the network when it does not receive any acknowledgement. Therefore the delay in receiving an acknowledgement is interpreted as Congestion in the network.

→ On Sensing this Congestion, the Source slows down.

→ This type of Congestion Policy is used by Tcp.

(4) Explicit Signaling:-

→ Here, the Congested nodes Explicitly Sends a Signal to the Source or destination to information about the Congestion.

→ This method is different from the choke Packet method. In choke Packet, a Seperate Packet is used for this Purpose whereas in Explicit Signaling method, the Signal is included in the

Packets that carry data.

→ It can occur in either the forward direction backward direction.

→ In backward signaling, a bit is sent in a packet moving in the opposite direction to the congestion. This bit warns the source about the congestion & informs the source to slow down.

→ In forward signaling, a bit sent in a packet moving in the direction of congestion. This bit warns the destination about the congestion. The receiver in this case use policies such as slowing down the acknowledgements to remove the congestion

2. Congestion Prevention Policies:-

Layer	Policies
Transport	• Retransmission Policy, • Out-of-order caching Policy, • Acknowledgement Policy, • Flow-control Policy • Time-out determination
Network	• Virtual circuit Vs datagram inside the subnet • Packet queing & service policy • Packet discart Policy, • Routing algorithm, • Packet-lifetime management.
Data link	• Retransmission Policy • Out-of-order caching Policy • Acknowledgement Policy • Flow control Policy.

fig ②:- Policies that affect Congestion.

Qp → Clearly Explained Each & Every Policy in Previous topic, Please do refer.

(3) Congestion Control in Virtual-Circuit Subnets:-

→ Congestion Control methods described by Open & closed loop to Prevent or avoid Congestion.

→ Congestion Control in dynamic Circuits (i.e in Virtual-Circuit Subnet) is Controlled by using Admission Control Policy.

→ One technique that is widely used to Keep Congestion that has already started from getting worse is admission Control.

→ The idea is Simple, Once algorithm Congestion has been Signaled, no more Virtual Circuits are setup Until the Problem has gone away, Thus attempts to set up new transport layer Connections fails.

→ In telephone System, when a Switch gets Overloaded, it also Pratices admission Control by not giving dial tones.

→ An alternative technique / approach is to allow new Virtual circuit but Carefully route all new Virtual Circuit around Problem areas.

→ for Example, Consider the Subnet of fig (a) in which two routers are Congested, as indicated.

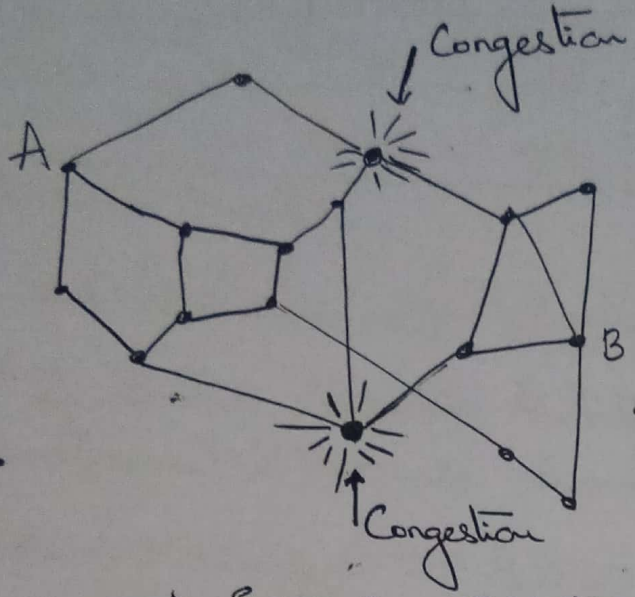


fig (a) A Congested Subnet.

→ Suppose, assume a host attached to router A. wants to setup a Connection to a host attached to router B. Normally, this Connection would Pass through One the Congested routers. To avoid this Situation, we can redraw the subnet as shown in fig (b).

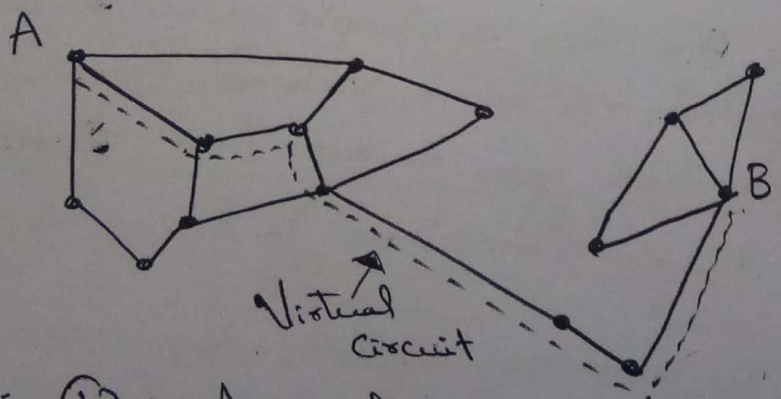


fig (b) : A re-drawn subnet that Eliminates Congestion.

→

Congestion Control in Datagram Subnets:-

→ Each router can easily monitor the utilization of its output lines & other resources.

Eg:- It can associate with each line a real variable u, whose values b/w 0.0 & 1.0 reflects the recent utilization of that line. To maintain good estimate of u, a sample of instantaneous line utilization, f (either 0 or 1), can be made periodically & u updated according to

$$u_{new} = a u_{old} + (1-a) f$$

where:-
a → Constant

→ Whenever u moves above the threshhold, the output line enters a "warning" state. Each newly-arriving packet is checked to see if its output line is in warning state. If it is

(a) Warning Bit:-

→ The old DECNET architecture signaled the warning state by setting a special bit in the packet header.

→ So does frame relay. When the packet arrived at its destination, the transport entity copied the bit into next acknowledgement sent back to the source. Then source then cut back on traffic.

→ As long as the router was in the warning state, it continued to set the warning bit, which meant that the source continued to get acknowledgements with it set.

→ The source monitored the fraction of acks with the bit set & adjusted its transmission accordingly.

→ As long as the warning bits continued to flow in, the source continued to decrease its transmission rate.

→ When they slowed to a trickle, it increased its transmission rate.

(b) Choke Packets :-

→ Here, the router sends a choke packet back to the source host, giving it the destination found in the packet.

→ The original packet is tagged. So, that will not generate any more choke packets farther along the path & is then forwarded in the usual way.

→ When the source host gets the choke packet, it is required to reduce the traffic sent to the specified destination by X percent.

→ Since other packets aimed at the same destination are probably already underway & will generate yet more choke packets, the host should ignore choke packets referring to that destination for a fixed time interval.

→ After that period has expired, the host listens for more choke packets for another interval.

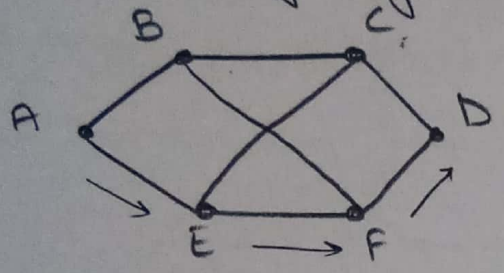
→ If one arrives, the line is still congested, so the host reduces the flow still more & begins ignoring choke packets again.

→ If no choke packets arrive during the listening period, the host may increase the flow again.

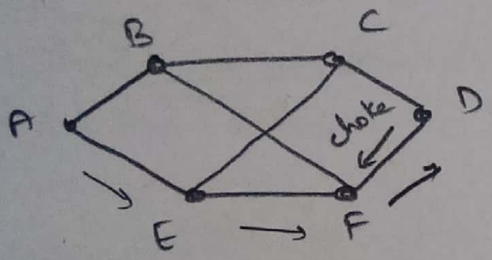
Hop-by-Hop Count:-

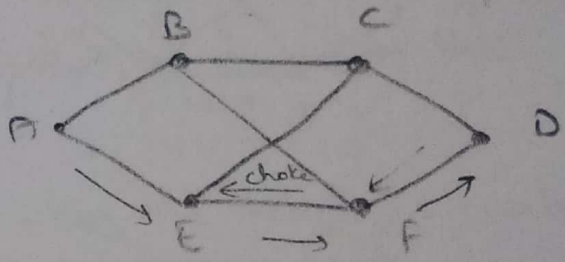
→ At high-speeds or over long distances, sending a choke packet to the source host doesn't work well because the reaction is so slow.

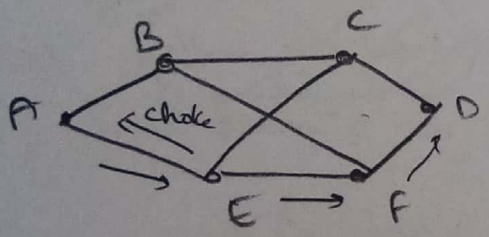
→ for Eg:- Consider fig (a).

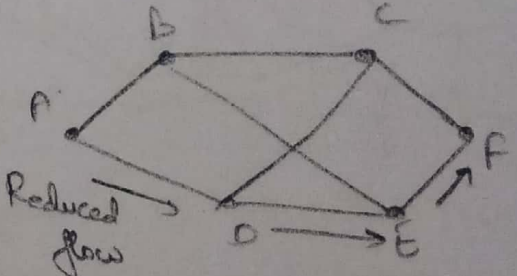


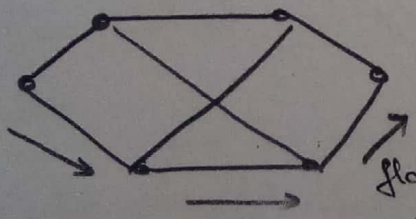
A → D i.e 155 Mbps.
time taken → 30 Sec.
→ 4.6 mega bits



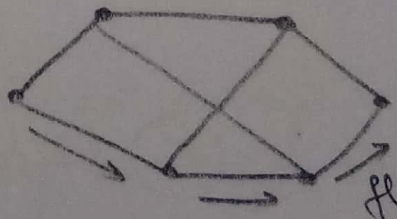








flow is still at maximum rate



flow is reduced

fig (a):- A choke packet that affects only the source.

→ Alternative approach is to have the choke packet that effects at every hop it passes through, as shown in sequence of fig (b)

→ Here, as soon as the choke packet reaches F, F is required to reduce the flow to D. Next to previous node until source. So, finally 'A' genuinely slows down.

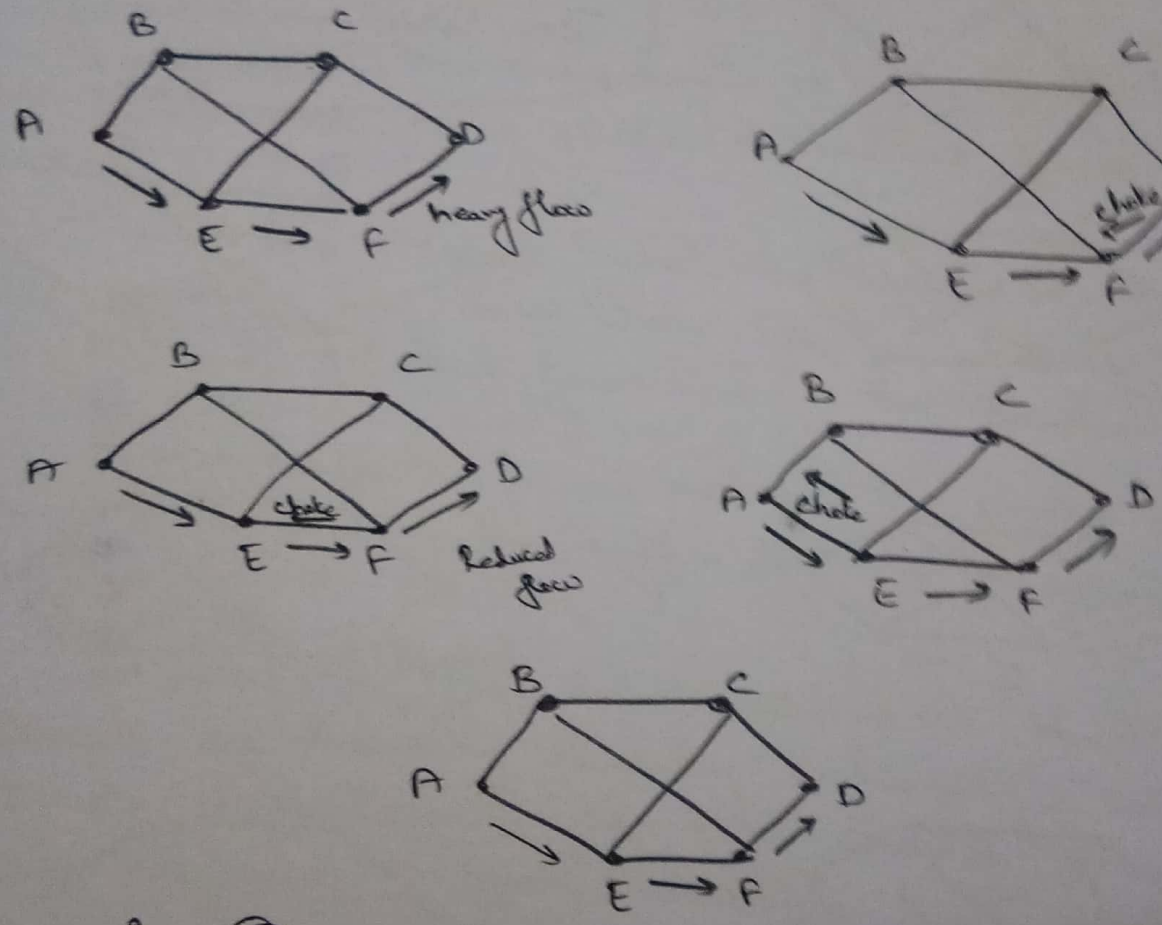


fig (b) A choke packet that effects each hop it passes through.

→ The net effect of this approach is to provide quick relief at the point of congestion at the price of using more buffers upstream.

→ In this way, congestion can be nipped in the bud without loosing any packets.

Load Shedding :-

→ When none of the above methods make congestion disappear, routers can bring out the heavy artillery called Load Shedding.

→ Load Shedding is a fancy way of saying that when routers are being inundated by Packets that they can handle, they just throw them away.

→ A router drowning in Packets can just Pick Packets at random to drop, but usually it can do better than that. Which Packet to discard may depend on the applications running.

→ For file transfer, an old Packet is worth more than a new one because dropping Packet 6 & keeping Packet 7 through 10 will cause a gap at the receiver that may force Packets 6 through 10 to be retransmitted.

→ The former Policy (old is better than new) is often called Wine & the latter (new is better than old) is often called milk.

→ To implement an intelligent discard Policy, applications must mark their Packets in Priority classes to indicate how important they are.

→ If they do this, then when Packets have to be discarded, routers can first drop Packets from the lowest class, then the next lowest class & so on.

→ There is some significant incentive to mark Packets as anything other than VERY IMPORTANT — NEVER, EVER DISCARD etc.

→ Alternatively, Senders might be allowed to send high priority packets under conditions of light load, but if the load increased they would be discarded.

⑥ Jitter Control:-

→ Applications such as audio, & video streaming, it doesn't matter much if the Packet take 20msec or 30 msec to be delivered, as long as the transit time is Constant.

→ The variation in the Packet arrival times is called jitter.

→ High jitter, having some Packets taking 20msec & other taking 30msec to arrive will give an Uneven quality to the sound or noise as shown in fig ⓐ below.

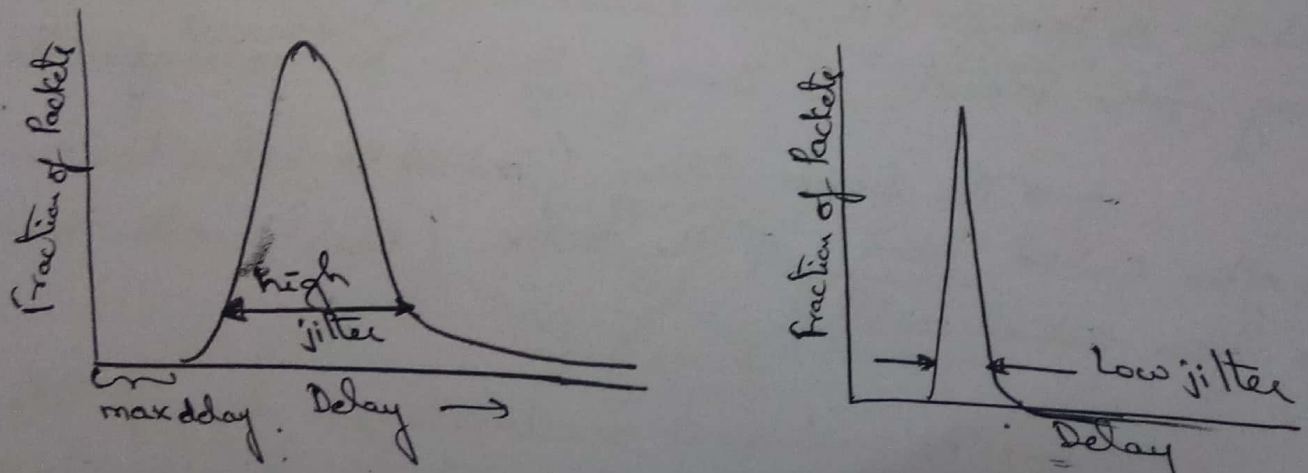


fig ⓐ High jitter ⓑ Low jitter.

→ The jitter can be bounded by Computing the Expected transist time for each hop along with the path.

→ When a Packet arrives at the router, the router checks to see how much the Packet is behind or ahead of its schedule.

→ This information is stored in the Packet & updated at each hop.

→ If the Packet is ahead of schedule, it is held just long enough to get it back scheduled.

→ If it is behind schedule, the router tries to get it out the door quickly.

(6) Traffic Shaping:-

→ Traffic Shaping is about regulating the average rate of data transmission.

→ When Connection is set-up, the User and the subnet agree on certain traffic Patterns for that Circuits.

→ Sometimes this is called a Service level agreement.

→ Traffic shaping reduces Congestion & thus helps the Carrier live up to the Promise.

Eg:- audio, Video

→ Monitoring a traffic flow is called traffic Policing.

→ Agreeing to a traffic shape & Policing it afterward are easier with Virtual Circuit subnets than with datagram subnets.

⑤ Leaky Bucket Algorithm:-

→ Imagine a bucket with a small hole in bottom, as shown in fig ④. No matter the rate water enters the bucket, the outflow is at Constant P, when there is any water in the bucket & Zero when the bucket is Empty.

→ Also, Once bucket is full, any additional water Entering it spills over the sides & is lost.

→ Now, the same idea can be applied to Packets as shown in fig ⑥.

→ If a Packet arrives at the queued; the new Packet is Unceremoniously discarded.

→ This arrangement can be built into the hardware interface or Simulated by the host operating System.

→ It is first Proposed by Turner (1986) & it is called Leaky bucket Algorithm.

→ Leaky bucket Algorithm is similar to Single-Server queuing System with Constant Service time.

→ When the Packets are all the same size (Eg., ATM Cells), this algorithm is used, if Packets are of Variable-sized Packets are being used, it is often better to allow a fixed no. of bytes Per tick, rather than just one Packet.

128/214

(8)

→ The leaky bucket Consists of a finite queues, otherwise it is discarded.

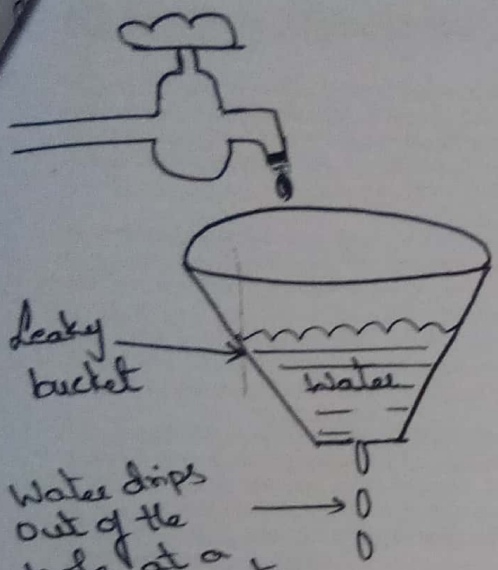


fig (a) A leaky bucket with water

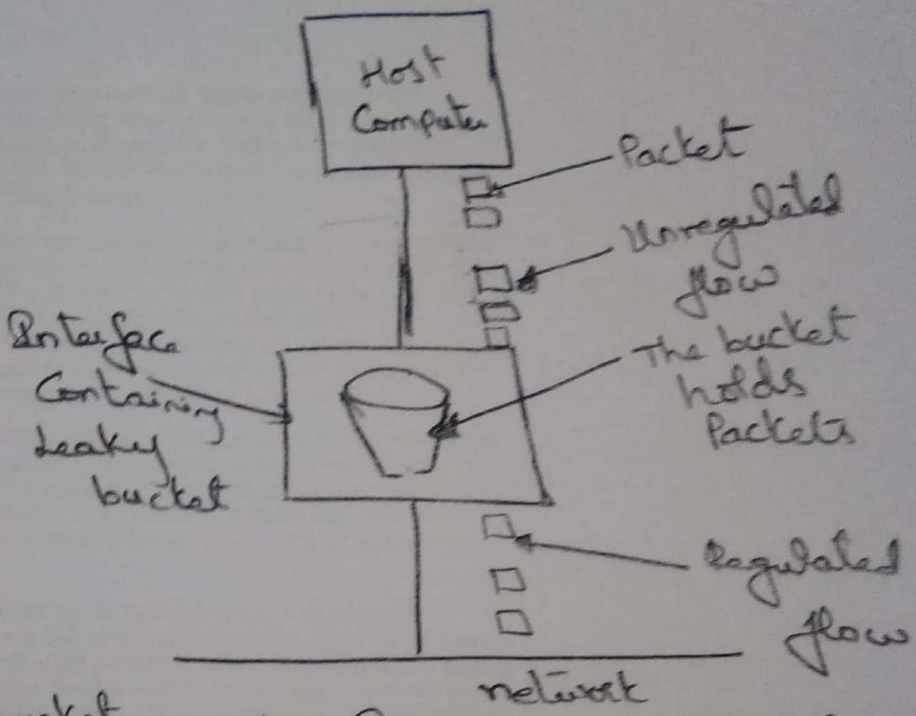


fig (b) a leaky bucket with Packets.

5.2 Routing Algorithms

The main function of the network layer is routing packets from the source machine to the destination machine. In most subnets, packets will require multiple hops to make the journey. The only notable exception is for broadcast networks, but even here routing is an issue if the source and destination are not on the same network. The algorithms that choose the routes and the data structures that they use are a major area of network layer design.

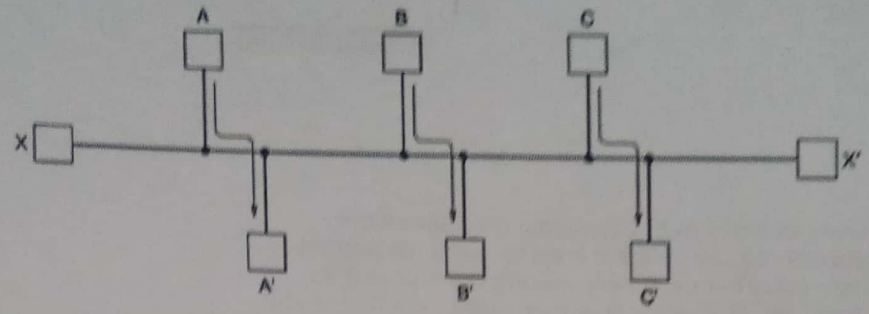
The **routing algorithm** is that part of the network layer software responsible for deciding which output line an incoming packet should be transmitted on. If the subnet uses datagrams internally, this decision must be made anew for every arriving data packet since the best route may have changed since last time. If the subnet uses virtual circuits internally, routing decisions are made only when a new virtual circuit is being set up. Thereafter, data packets just follow the previously-established route. The latter case is sometimes called **session routing** because a route remains in force for an entire user session (e.g., a login session at a terminal or a file transfer).

It is sometimes useful to make a distinction between routing, which is making the decision which routes to use, and forwarding, which is what happens when a packet arrives. One can think of a router as having two processes inside it. One of them handles each packet as it arrives, looking up the outgoing line to use for it in the routing tables. This process is **forwarding**. The other process is responsible for filling in and updating the routing tables. That is where the routing algorithm comes into play.

Regardless of whether routes are chosen independently for each packet or only when new connections are established, certain properties are desirable in a routing algorithm: correctness, simplicity, robustness, stability, fairness, and optimality. Correctness and simplicity hardly require comment, but the need for robustness may be less obvious at first. Once a major network comes on the air, it may be expected to run continuously for years without systemwide failures. During that period there will be hardware and software failures of all kinds. Hosts, routers, and lines will fail repeatedly, and the topology will change many times. The routing algorithm should be able to cope with changes in the topology and traffic without requiring all jobs in all hosts to be aborted and the network to be rebooted every time some router crashes.

Stability is also an important goal for the routing algorithm. There exist routing algorithms that never converge to equilibrium, no matter how long they run. A stable algorithm reaches equilibrium and stays there. Fairness and optimality may sound obvious—surely no reasonable person would oppose them—but as it turns out, they are often contradictory goals. As a simple example of this conflict, look at Fig. 5-5. Suppose that there is enough traffic between *A* and *A'*, between *B* and *B'*, and between *C* and *C'* to saturate the horizontal links. To maximize the total flow, the *X* to *X'* traffic should be shut off altogether. Unfortunately, *X* and *X'* may not see it that way. Evidently, some compromise between global efficiency and fairness to individual connections is needed.

Figure 5-5. Conflict between fairness and optimality.



Before we can even attempt to find trade-offs between fairness and optimality, we must decide what it is we seek to optimize. Minimizing mean packet delay is an obvious candidate, but so is maximizing total network throughput. Furthermore, these two goals are also in conflict, since operating any queueing system near capacity implies a long queueing delay. As a compromise, many networks attempt to minimize the number of hops a packet must make, because reducing the number of hops tends to improve the delay and also reduce the amount of bandwidth consumed, which tends to improve the throughput as well.

Routing algorithms can be grouped into two major classes: nonadaptive and adaptive. **Nonadaptive algorithms** do not base their routing decisions on measurements or estimates of the current traffic and topology. Instead, the choice of the route to use to get from I to J (for all I and J) is computed in advance, off-line, and downloaded to the routers when the network is booted. This procedure is sometimes called **static routing**.

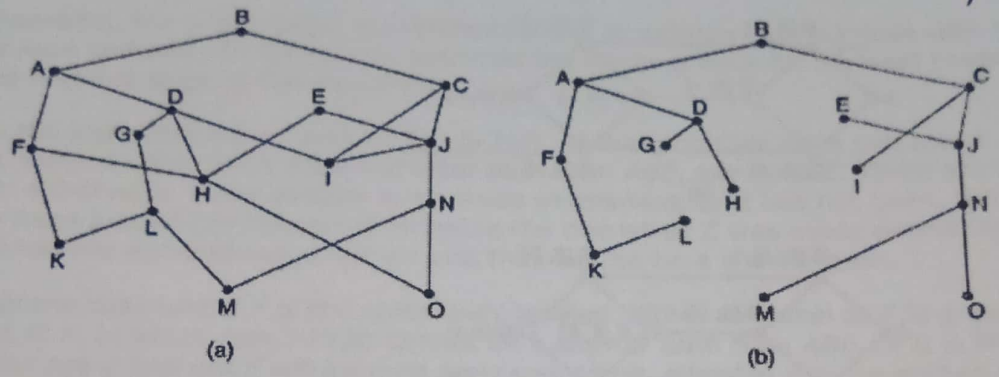
Adaptive algorithms, in contrast, change their routing decisions to reflect changes in the topology, and usually the traffic as well. Adaptive algorithms differ in where they get their information (e.g., locally, from adjacent routers, or from all routers), when they change the routes (e.g., every ΔT sec, when the load changes or when the topology changes), and what metric is used for optimization (e.g., distance, number of hops, or estimated transit time). In the following sections we will discuss a variety of routing algorithms, both static and dynamic.

5.2.1 The Optimality Principle

Before we get into specific algorithms, it may be helpful to note that one can make a general statement about optimal routes without regard to network topology or traffic. This statement is known as the **optimality principle**. It states that if router J is on the optimal path from router I to router K, then the optimal path from J to K also falls along the same route. To see this, call the part of the route from I to J r_1 and the rest of the route r_2. If a route better than r_2 existed from J to K, it could be concatenated with r_1 to improve the route from I to K, contradicting our statement that r_1r_2 is optimal.

As a direct consequence of the optimality principle, we can see that the set of optimal routes from all sources to a given destination form a tree rooted at the destination. Such a tree is called a **sink tree** and is illustrated in Fig. 5-6, where the distance metric is the number of hops. Note that a sink tree is not necessarily unique; other trees with the same path lengths may exist. The goal of all routing algorithms is to discover and use the sink trees for all routers.

Figure 5-6. (a) A subnet. (b) A sink tree for router *B*.



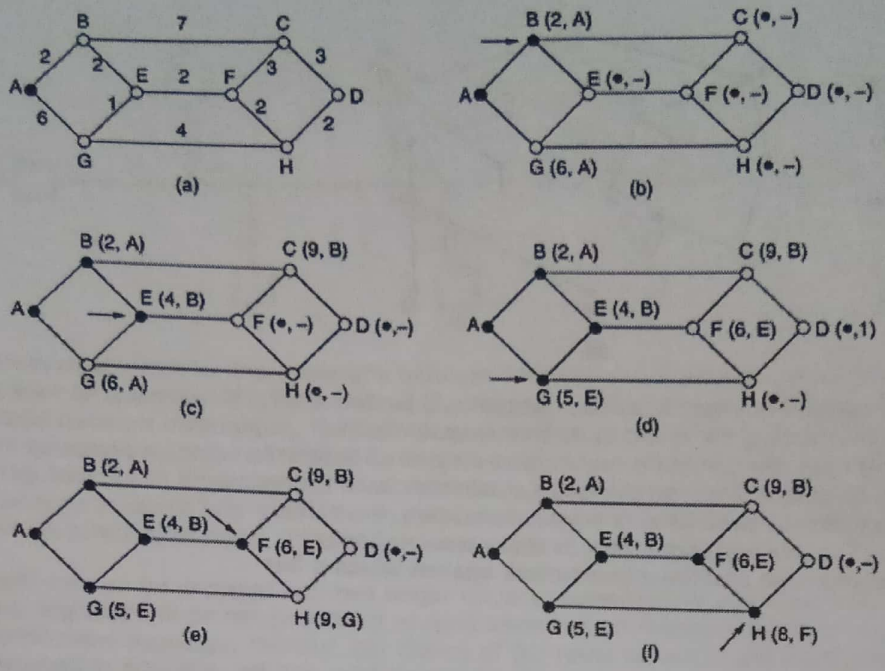
Since a sink tree is indeed a tree, it does not contain any loops, so each packet will be delivered within a finite and bounded number of hops. In practice, life is not quite this easy. Links and routers can go down and come back up during operation, so different routers may have different ideas about the current topology. Also, we have quietly finessed the issue of whether each router has to individually acquire the information on which to base its sink tree computation or whether this information is collected by some other means. We will come back to these issues shortly. Nevertheless, the optimality principle and the sink tree provide a benchmark against which other routing algorithms can be measured.

5.2.2 Shortest Path Routing

Let us begin our study of feasible routing algorithms with a technique that is widely used in many forms because it is simple and easy to understand. The idea is to build a graph of the subnet, with each node of the graph representing a router and each arc of the graph representing a communication line (often called a link). To choose a route between a given pair of routers, the algorithm just finds the shortest path between them on the graph.

The concept of a **shortest path** deserves some explanation. One way of measuring path length is the number of hops. Using this metric, the paths *ABC* and *ABE* in Fig. 5-7 are equally long. Another metric is the geographic distance in kilometers, in which case *ABC* is clearly much longer than *ABE* (assuming the figure is drawn to scale).

Figure 5-7. The first five steps used in computing the shortest path from *A* to *D*. The arrows indicate the working node.



However, many other metrics besides hops and physical distance are also possible. For example, each arc could be labeled with the mean queueing and transmission delay for some standard test packet as determined by hourly test runs. With this graph labeling, the shortest path is the fastest path rather than the path with the fewest arcs or kilometers.

In the general case, the labels on the arcs could be computed as a function of the distance, bandwidth, average traffic, communication cost, mean queue length, measured delay, and other factors. By changing the weighting function, the algorithm would then compute the "shortest" path measured according to any one of a number of criteria or to a combination of criteria.

Several algorithms for computing the shortest path between two nodes of a graph are known. This one is due to Dijkstra (1959). Each node is labeled (in parentheses) with its distance from the source node along the best known path. Initially, no paths are known, so all nodes are labeled with infinity. As the algorithm proceeds and paths are found, the labels may change, reflecting better paths. A label may be either tentative or permanent. Initially, all labels are tentative. When it is discovered that a label represents the shortest possible path from the source to that node, it is made permanent and never changed thereafter.

To illustrate how the labeling algorithm works, look at the weighted, undirected graph of Fig. 5-7(a), where the weights represent, for example, distance. We want to find the shortest path from *A* to *D*. We start out by marking node *A* as permanent, indicated by a filled-in circle. Then we examine, in turn, each of the nodes adjacent to *A* (the working node), relabeling each one with the distance to *A*. Whenever a node is relabeled, we also label it with the node from which the probe was made so that we can reconstruct the final path later. Having examined each of the nodes adjacent to *A*, we examine all the tentatively labeled nodes in the whole graph and make the one with the smallest label permanent, as shown in Fig. 5-7(b). This one becomes the new working node.

We now start at *B* and examine all nodes adjacent to it. If the sum of the label on *B* and the distance from *B* to the node being considered is less than the label on that node, we have a shorter path, so the node is relabeled.

After all the nodes adjacent to the working node have been inspected and the tentative labels

changed if possible, the entire graph is searched for the tentatively-labeled node with the smallest value. This node is made permanent and becomes the working node for the next round. Figure 5-7 shows the first five steps of the algorithm.

To see why the algorithm works, look at Fig. 5-7(c). At that point we have just made *E* permanent. Suppose that there were a shorter path than *ABE*, say *AXYZE*. There are two possibilities: either node *Z* has already been made permanent, or it has not been. If it has, then *E* has already been probed (on the round following the one when *Z* was made permanent), so the *AXYZE* path has not escaped our attention and thus cannot be a shorter path.

Now consider the case where *Z* is still tentatively labeled. Either the label at *Z* is greater than or equal to that at *E*, in which case *AXYZE* cannot be a shorter path than *ABE*, or it is less than that of *E*, in which case *Z* and not *E* will become permanent first, allowing *E* to be probed from *Z*.

This algorithm is given in Fig. 5-8. The global variables *n* and *dist* describe the graph and are initialized before *shortest_path* is called. The only difference between the program and the algorithm described above is that in Fig. 5-8, we compute the shortest path starting at the terminal node, *t*, rather than at the source node, *s*. Since the shortest path from *t* to *s* in an undirected graph is the same as the shortest path from *s* to *t*, it does not matter at which end we begin (unless there are several shortest paths, in which case reversing the search might discover a different one). The reason for searching backward is that each node is labeled with its predecessor rather than its successor. When the final path is copied into the output variable, *path*, the path is thus reversed. By reversing the search, the two effects cancel, and the answer is produced in the correct order.

Figure 5-8. Dijkstra's algorithm to compute the shortest path through a graph.

```
#define MAX_NODES 1024                  /* maximum number of nodes */
#define INFINITY 1000000000             /* a number larger than every maximum path */
int n, dist[MAX_NODES][MAX_NODES];/* dist[i][j] is the distance from i to j */

void shortest_path(int s, int t, int path[])
{ struct state {                        /* the path being worked on */
      int predecessor;                  /* previous node */
      int length;                       /* length from source to this node */
      enum {permanent, tentative} label; /* label state */
  } state[MAX_NODES];

  int i, k, min;
  struct state *p;

  for (p = &state[0]; p < &state[n]; p++) { /* initialize state */
      p->predecessor = -1;
      p->length = INFINITY;
      p->label = tentative;
  }
  state[t].length = 0;  state[t].label = permanent;
  k = t;                                /* k is the initial working node */
  do {                                  /* Is there a better path from k? */
      for (i = 0; i < n; i++)           /* this graph has n nodes */
          if (dist[k][i] != 0 && state[i].label == tentative) {
              if (state[k].length + dist[k][i] < state[i].length) {
                  state[i].predecessor = k;
                  state[i].length = state[k].length + dist[k][i];
              }
          }

      /* Find the tentatively labeled node with the smallest label. */
      k = 0; min = INFINITY;
      for (i = 0; i < n; i++)
          if (state[i].label == tentative && state[i].length < min) {
              min = state[i].length;
              k = i;
          }
      state[k].label = permanent;
  } while (k != s);

  /* Copy the path into the output array. */
  i = 0;  k = s;
  do {path[i++] = k; k = state[k].predecessor; } while (k >= 0);
}
```

5.2.3 Flooding

Another static algorithm is **flooding**, in which every incoming packet is sent out on every outgoing line except the one it arrived on. Flooding obviously generates vast numbers of duplicate packets, in fact, an infinite number unless some measures are taken to damp the process. One such measure is to have a hop counter contained in the header of each packet, which is decremented at each hop, with the packet being discarded when the counter reaches zero. Ideally, the hop counter should be initialized to the length of the path from source to destination. If the sender does not know how long the path is, it can initialize the counter to the worst case, namely, the full diameter of the subnet.

An alternative technique for damming the flood is to keep track of which packets have been flooded, to avoid sending them out a second time. achieve this goal is to have the source router put a sequence number in each packet it receives from its hosts. Each router then needs a list per source router telling which sequence numbers originating at that source have already been seen.

if an incoming packet is on the list, it is not flooded.

To prevent the list from growing without bound, each list should be augmented by a counter, k, meaning that all sequence numbers through k have been seen. When a packet comes in, it is easy to check if the packet is a duplicate; if so, it is discarded. Furthermore, the full list below k is not needed, since k effectively summarizes it.

A variation of flooding that is slightly more practical is **selective flooding**.In this algorithm the routers do not send every incoming packet out on every line, only on those lines that are going approximately in the right direction. There is usually little point in sending a westbound packet on an eastbound line unless the topology is extremely peculiar and the router is sure of this fact.

Flooding is not practical in most applications, but it does have some uses. For example, in military applications, where large numbers of routers may be blown to bits at any instant, the tremendous robustness of flooding is highly desirable. In distributed database applications, it is sometimes necessary to update all the databases concurrently, in which case flooding can be useful. In wireless networks, all messages transmitted by a station can be received by all other stations within its radio range, which is, in fact, flooding, and some algorithms utilize this property. A fourth possible use of flooding is as a metric against which other routing algorithms can be compared. Flooding always chooses the shortest path because it chooses every possible path in parallel. Consequently, no other algorithm can produce a shorter delay (if we ignore the overhead generated by the flooding process itself).

5.2.4 Distance Vector Routing

Modern computer networks generally use dynamic routing algorithms rather than the static ones described above because static algorithms do not take the current network load into account. Two dynamic algorithms in particular, distance vector routing and link state routing, are the most popular. In this section we will look at the former algorithm. In the following section we will study the latter algorithm.

Distance vector routing algorithms operate by having each router maintain a table (i.e, a vector) giving the best known distance to each destination and which line to use to get there. These tables are updated by exchanging information with the neighbors.

The distance vector routing algorithm is sometimes called by other names, most commonly the distributed **Bellman-Ford** routing algorithm and the **Ford-Fulkerson** algorithm, after the researchers who developed it (Bellman, 1957; and Ford and Fulkerson, 1962). It was the original ARPANET routing algorithm and was also used in the Internet under the name RIP.

In distance vector routing, each router maintains a routing table indexed by, and containing one entry for, each router in the subnet. This entry contains two parts: the preferred outgoing line to use for that destination and an estimate of the time or distance to that destination. The metric used might be number of hops, time delay in milliseconds, total number of packets queued along the path, or something similar.

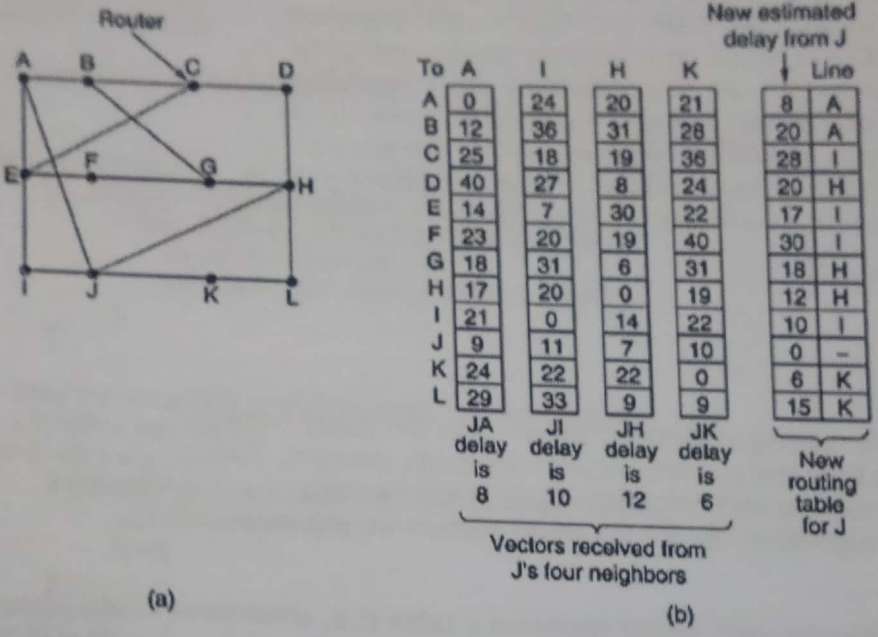
The router is assumed to know the "distance" to each of its neighbors. If the metric is hops, the distance is just one hop. If the metric is queue length, the router simply examines each queue. If the metric is delay, the router can measure it directly with special ECHO packets that the receiver just timestamps and sends back as fast as it can.

As an example, assume that delay is used as a metric and that the router knows the delay to each of its neighbors. Once every T msec each router sends to each neighbor a list of its estimated delays to each destination. It also receives a similar list from each neighbor. Imagine that one of these tables has just come in from neighbor X, with X_i being X's estimate of how long

it takes to get to router *i*. If the router knows that the delay to *X* is *m* msec, it also knows that it can reach router *i* via *X* in $X_i + m$ msec. By performing this calculation for each neighbor, a router can find out which estimate seems the best and use that estimate and the corresponding line in its new routing table. Note that the old routing table is not used in the calculation.

This updating process is illustrated in Fig. 5-9. Part (a) shows a subnet. The first four columns of part (b) show the delay vectors received from the neighbors of router *J*. *A* claims to have a 12-msec delay to *B*, a 25-msec delay to *C*, a 40-msec delay to *D*, etc. Suppose that *J* has measured or estimated its delay to its neighbors, *A*, *I*, *H*, and *K* as 8, 10, 12, and 6 msec, respectively.

Figure 5-9. (a) A subnet. (b) Input from *A*, *I*, *H*, *K*, and the new routing table for *J*.



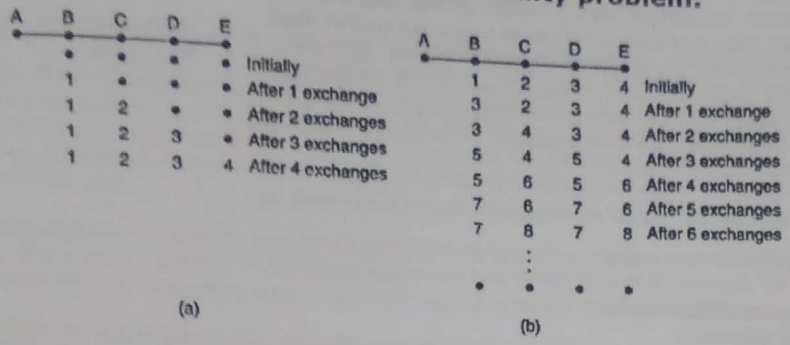
Consider how *J* computes its new route to router *G*. It knows that it can get to *A* in 8 msec, and *A* claims to be able to get to *G* in 18 msec, so *J* knows it can count on a delay of 26 msec to *G* if it forwards packets bound for *G* to *A*. Similarly, it computes the delay to *G* via *I*, *H*, and *K* as 41 (31 + 10), 18 (6 + 12), and 37 (31 + 6) msec, respectively. The best of these values is 18, so it makes an entry in its routing table that the delay to *G* is 18 msec and that the route to use is via *H*. The same calculation is performed for all the other destinations, with the new routing table shown in the last column of the figure.

The Count-to-Infinity Problem

Distance vector routing works in theory but has a serious drawback in practice: although it converges to the correct answer, it may do so slowly. In particular, it reacts rapidly to good news, but leisurely to bad news. Consider a router whose best route to destination *X* is large. If on the next exchange neighbor *A* suddenly reports a short delay to *X*, the router just switches over to using the line to *A* to send traffic to *X*. In one vector exchange, the good news is processed.

To see how fast good news propagates, consider the five-node (linear) subnet of Fig. 5-10, where the delay metric is the number of hops. Suppose *A* is down initially and all the other routers know this. In other words, they have all recorded the delay to *A* as infinity.

Figure 5-10. The count-to-infinity problem.



When *A* comes up, the other routers learn about it via the vector exchanges. For simplicity we will assume that there is a gigantic gong somewhere that is struck periodically to initiate a vector exchange at all routers simultaneously. At the time of the first exchange, *B* learns that its left neighbor has zero delay to *A*. *B* now makes an entry in its routing table that *A* is one hop away to the left. All the other routers still think that *A* is down. At this point, the routing table entries for *A* are as shown in the second row of Fig. 5-10(a). On the next exchange, *C* learns that *B* has a path of length 1 to *A*, so it updates its routing table to indicate a path of length 2, but *D* and *E* do not hear the good news until later. Clearly, the good news is spreading at the rate of one hop per exchange. In a subnet whose longest path is of length *N* hops, within *N* exchanges everyone will know about newly-revived lines and routers.

Now let us consider the situation of Fig. 5-10(b), in which all the lines and routers are initially up. Routers *B*, *C*, *D*, and *E* have distances to *A* of 1, 2, 3, and 4, respectively. Suddenly *A* goes down, or alternatively, the line between *A* and *B* is cut, which is effectively the same thing from *B*'s point of view.

At the first packet exchange, *B* does not hear anything from *A*. Fortunately, *C* says: Do not worry; I have a path to *A* of length 2. Little does *B* know that *C*'s path runs through *B* itself. For all *B* knows, *C* might have ten lines all with separate paths to *A* of length 2. As a result, *B* thinks it can reach *A* via *C*, with a path length of 3. *D* and *E* do not update their entries for *A* on the first exchange.

On the second exchange, *C* notices that each of its neighbors claims to have a path to *A* of length 3. It picks one of the them at random and makes its new distance to *A* 4, as shown in the third row of Fig. 5-10(b). Subsequent exchanges produce the history shown in the rest of Fig. 5-10(b).

From this figure, it should be clear why bad news travels slowly: no router ever has a value more than one higher than the minimum of all its neighbors. Gradually, all routers work their way up to infinity, but the number of exchanges required depends on the numerical value used for infinity. For this reason, it is wise to set infinity to the longest path plus 1. If the metric is time delay, there is no well-defined upper bound, so a high value is needed to prevent a path with a long delay from being treated as down. Not entirely surprisingly, this problem is known as the **count-to-infinity** problem. There have been a few attempts to solve it (such as split horizon with poisoned reverse in RFC 1058), but none of these work well in general. The core of the problem is that when *X* tells *Y* that it has a path somewhere, *Y* has no way of knowing whether it itself is on the path.

5.2.5 Link State Routing

Distance vector routing was used in the ARPANET until 1979, when it was replaced by link state

routing. Two primary problems caused its demise. First, since the delay metric was queue length, it did not take line bandwidth into account when choosing routes. Initially, all the lines were 56 kbps, so line bandwidth was not an issue, but after some lines had been upgraded to 230 kbps and others to 1.544 Mbps, not taking bandwidth into account was a major problem. Of course, it would have been possible to change the delay metric to factor in line bandwidth, but a second problem also existed, namely, the algorithm often took too long to converge (the count-to-infinity problem). For these reasons, it was replaced by an entirely new algorithm, now called **link state routing**. Variants of link state routing are now widely used.

The idea behind link state routing is simple and can be stated as five parts. Each router must do the following:

1. Discover its neighbors and learn their network addresses.
2. Measure the delay or cost to each of its neighbors.
3. Construct a packet telling all it has just learned.
4. Send this packet to all other routers.
5. Compute the shortest path to every other router.

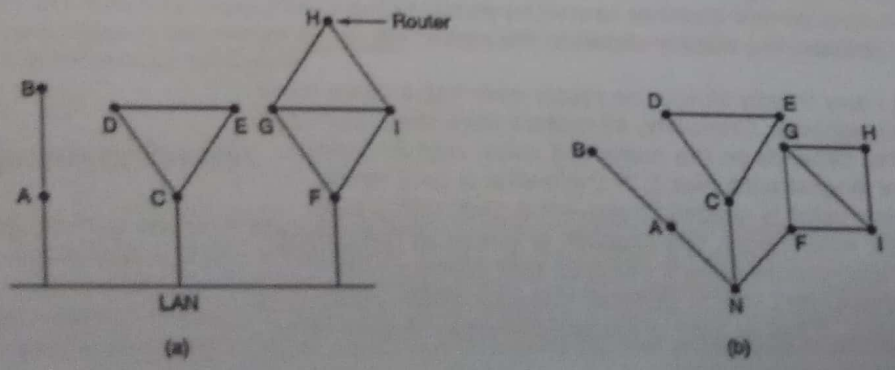
In effect, the complete topology and all delays are experimentally measured and distributed to every router. Then Dijkstra's algorithm can be run to find the shortest path to every other router. Below we will consider each of these five steps in more detail.

Learning about the Neighbors

When a router is booted, its first task is to learn who its neighbors are. It accomplishes this goal by sending a special HELLO packet on each point-to-point line. The router on the other end is expected to send back a reply telling who it is. These names must be globally unique because when a distant router later hears that three routers are all connected to *F*, it is essential that it can determine whether all three mean the same *F*.

When two or more routers are connected by a LAN, the situation is slightly more complicated. Fig. 5-11(a) illustrates a LAN to which three routers, *A*, *C*, and *F*, are directly connected. Each of these routers is connected to one or more additional routers, as shown.

Figure 5-11. (a) Nine routers and a LAN. (b) A graph model of (a).



One way to model the LAN is to consider it as a node itself, as shown in Fig. 5-11(b). Here we have introduced a new, artificial node, *N*, to which *A*, *C*, and *F* are connected. The fact that it is possible to go from *A* to *C* on the LAN is represented by the path *ANC* here.

140/214

Measuring Line Cost

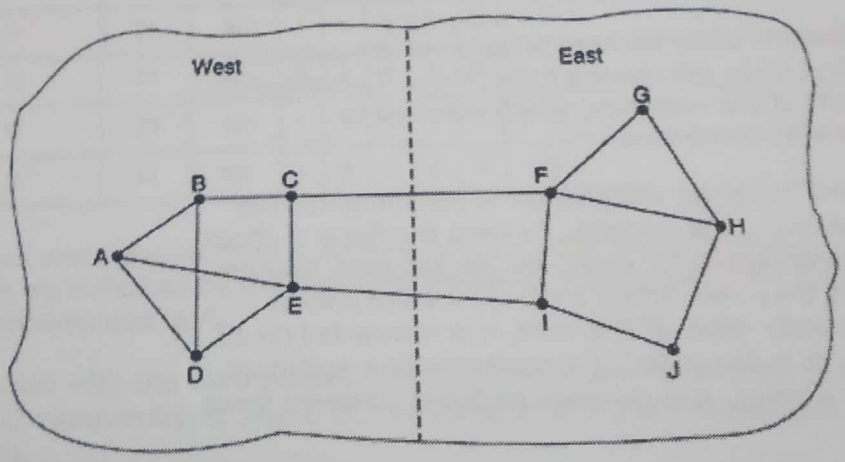
The link state routing algorithm requires each router to know, or at least have a reasonable estimate of, the delay to each of its neighbors. The most direct way to determine this delay is to send over the line a special ECHO packet that the other side is required to send back immediately. By measuring the round-trip time and dividing it by two, the sending router can get a reasonable estimate of the delay. For even better results, the test can be conducted several times, and the average used. Of course, this method implicitly assumes the delays are symmetric, which may not always be the case.

An interesting issue is whether to take the load into account when measuring the delay. To factor the load in, the round-trip timer must be started when the ECHO packet is queued. To ignore the load, the timer should be started when the ECHO packet reaches the front of the queue.

Arguments can be made both ways. Including traffic-induced delays in the measurements means that when a router has a choice between two lines with the same bandwidth, one of which is heavily loaded all the time and one of which is not, the router will regard the route over the unloaded line as a shorter path. This choice will result in better performance.

Unfortunately, there is also an argument against including the load in the delay calculation. Consider the subnet of Fig. 5-12, which is divided into two parts, East and West, connected by two lines, *CF* and *EI*.

Figure 5-12. A subnet in which the East and West parts are connected by two lines.

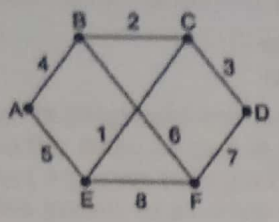


Suppose that most of the traffic between East and West is using line *CF*, and as a result, this line is heavily loaded with long delays. Including queueing delay in the shortest path calculation will make *EI* more attractive. After the new routing tables have been installed, most of the East-West traffic will now go over *EI*, overloading this line. Consequently, in the next update, *CF* will appear to be the shortest path. As a result, the routing tables may oscillate wildly, leading to erratic routing and many potential problems. If load is ignored and only bandwidth is considered, this problem does not occur. Alternatively, the load can be spread over both lines, but this solution does not fully utilize the best path. Nevertheless, to avoid oscillations in the choice of best path, it may be wise to distribute the load over multiple lines, with some known fraction going over each line.

Building Link State Packets

Once the information needed for the exchange has been collected, the next step is for each router to build a packet containing all the data. The packet starts with the identity of the sender, followed by a sequence number and age (to be described later), and a list of neighbors. For each neighbor, the delay to that neighbor is given. An example subnet is given in Fig. 5-13(a) with delays shown as labels on the lines. The corresponding link state packets for all six routers are shown in Fig. 5-13(b).

Figure 5-13. (a) A subnet. (b) The link state packets for this subnet.



(a)

Link State Packets

A	B	C	D	E	F
Seq.	Seq.	Seq.	Seq.	Seq.	Seq.
Age	Age	Age	Age	Age	Age
B 4	A 4	B 2	C 3	A 5	B 6
E 5	C 2	D 3	F 7	C 1	D 7
	F 6	E 1		F 8	E 8

(b)

Building the link state packets is easy. The hard part is determining when to build them. One possibility is to build them periodically, that is, at regular intervals. Another possibility is to build them when some significant event occurs, such as a line or neighbor going down or coming back up again or changing its properties appreciably.

Distributing the Link State Packets

The trickiest part of the algorithm is distributing the link state packets reliably. As the packets are distributed and installed, the routers getting the first ones will change their routes. Consequently, the different routers may be using different versions of the topology, which can lead to inconsistencies, loops, unreachable machines, and other problems.

First we will describe the basic distribution algorithm. Later we will give some refinements. The fundamental idea is to use flooding to distribute the link state packets. To keep the flood in check, each packet contains a sequence number that is incremented for each new packet sent. Routers keep track of all the (source router, sequence) pairs they see. When a new link state packet comes in, it is checked against the list of packets already seen. If it is new, it is forwarded on all lines except the one it arrived on. If it is a duplicate, it is discarded. If a packet with a sequence number lower than the highest one seen so far ever arrives, it is rejected as being obsolete since the router has more recent data.

This algorithm has a few problems, but they are manageable. First, if the sequence numbers wrap around, confusion will reign. The solution here is to use a 32-bit sequence number. With one link state packet per second, it would take 137 years to wrap around, so this possibility can be ignored.

Second, if a router ever crashes, it will lose track of its sequence number. If it starts again at 0, the next packet will be rejected as a duplicate.

Third, if a sequence number is ever corrupted and 65,540 is received instead of 4 (a 1-bit error), packets 5 through 65,540 will be rejected as obsolete, since the current sequence number is thought to be 65,540.

The solution to all these problems is to include the age of each packet after the sequence number and decrement it once per second. When the age hits zero, the information from that router is discarded. Normally, a new packet comes in, say, every 10 sec, so router information only times

when a router is down (or six consecutive packets have been lost, an unlikely event). The *Age* field is also decremented by each router during the initial flooding process, to make sure no packet can get lost and live for an indefinite period of time (a packet whose age is zero is discarded).

Some refinements to this algorithm make it more robust. When a link state packet comes in to a router for flooding, it is not queued for transmission immediately. Instead it is first put in a holding area to wait a short while. If another link state packet from the same source comes in before the first packet is transmitted, their sequence numbers are compared. If they are equal, the duplicate is discarded. If they are different, the older one is thrown out. To guard against errors on the router-router lines, all link state packets are acknowledged. When a line goes idle, the holding area is scanned in round-robin order to select a packet or acknowledgement to send.

The data structure used by router *B* for the subnet shown in Fig. 5-13(a) is depicted in Fig. 5-14. Each row here corresponds to a recently-arrived, but as yet not fully-processed, link state packet. The table records where the packet originated, its sequence number and age, and the data. In addition, there are send and acknowledgement flags for each of *B*'s three lines (to *A*, *C*, and *F*, respectively). The send flags mean that the packet must be sent on the indicated line. The acknowledgement flags mean that it must be acknowledged there.

Figure 5-14. The packet buffer for router *B* in Fig. 5-13.

			Send flags			ACK flags			
Source	Seq.	Age	A	C	F	A	C	F	Data
A	21	60	0	1	1	1	0	0	
F	21	60	1	1	0	0	0	1	
E	21	59	0	1	0	1	0	1	
C	20	60	1	0	1	0	1	0	
D	21	59	1	0	0	0	1	1	

In Fig. 5-14, the link state packet from *A* arrives directly, so it must be sent to *C* and *F* and acknowledged to *A*, as indicated by the flag bits. Similarly, the packet from *F* has to be forwarded to *A* and *C* and acknowledged to *F*.

However, the situation with the third packet, from *E*, is different. It arrived twice, once via *EAB* and once via *EFB*. Consequently, it has to be sent only to *C* but acknowledged to both *A* and *F*, as indicated by the bits.

If a duplicate arrives while the original is still in the buffer, bits have to be changed. For example, if a copy of *C*'s state arrives from *F* before the fourth entry in the table has been forwarded, the six bits will be changed to 100011 to indicate that the packet must be acknowledged to *F* but not sent there.

Computing the New Routes

Once a router has accumulated a full set of link state packets, it can construct the entire subnet graph because every link is represented. Every link is, in fact, represented twice, once for each direction. The two values can be averaged or used separately.

Now Dijkstra's algorithm can be run locally to construct the shortest path to all possible destinations. The results of this algorithm can be installed in the routing tables, and normal

operation resumed.

For a subnet with n routers, each of which has k neighbors, the memory required to store the input data is proportional to kn. For large subnets, this can be a problem. Also, the computation time can be an issue. Nevertheless, in many practical situations, link state routing works well.

However, problems with the hardware or software can wreak havoc with this algorithm (also with other ones). For example, if a router claims to have a line it does not have or forgets a line it does have, the subnet graph will be incorrect. If a router fails to forward packets or corrupts them while forwarding them, trouble will arise. Finally, if it runs out of memory or does the routing calculation wrong, bad things will happen. As the subnet grows into the range of tens or hundreds of thousands of nodes, the probability of some router failing occasionally becomes nonnegligible. The trick is to try to arrange to limit the damage when the inevitable happens. Perlman (1988) discusses these problems and their solutions in detail.

Link state routing is widely used in actual networks, so a few words about some example protocols using it are in order. The OSPF protocol, which is widely used in the Internet, uses a link state algorithm. We will describe OSPF in Sec. 5.6.4.

Another link state protocol is **IS-IS** (**Intermediate System-Intermediate System**), which was designed for DECnet and later adopted by ISO for use with its connectionless network layer protocol, CLNP. Since then it has been modified to handle other protocols as well, most notably, IP. IS-IS is used in some Internet backbones (including the old NSFNET backbone) and in some digital cellular systems such as CDPD. Novell NetWare uses a minor variant of IS-IS (NLSP) for routing IPX packets.

Basically IS-IS distributes a picture of the router topology, from which the shortest paths are computed. Each router announces, in its link state information, which network layer addresses it can reach directly. These addresses can be IP, IPX, AppleTalk, or any other addresses. IS-IS can even support multiple network layer protocols at the same time.

Many of the innovations designed for IS-IS were adopted by OSPF (OSPF was designed several years after IS-IS). These include a self-stabilizing method of flooding link state updates, the concept of a designated router on a LAN, and the method of computing and supporting path splitting and multiple metrics. As a consequence, there is very little difference between IS-IS and OSPF. The most important difference is that IS-IS is encoded in such a way that it is easy and natural to simultaneously carry information about multiple network layer protocols, a feature OSPF does not have. This advantage is especially valuable in large multiprotocol environments.

5.2.6 Hierarchical Routing

As networks grow in size, the router routing tables grow proportionally. Not only is router memory consumed by ever-increasing tables, but more CPU time is needed to scan them and more bandwidth is needed to send status reports about them. At a certain point the network may grow to the point where it is no longer feasible for every router to have an entry for every other router, so the routing will have to be done hierarchically, as it is in the telephone network.

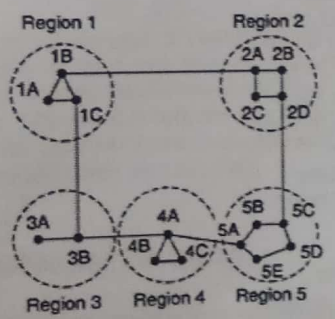
When hierarchical routing is used, the routers are divided into what we will call **regions**, with each router knowing all the details about how to route packets to destinations within its own region, but knowing nothing about the internal structure of other regions. When different networks are interconnected, it is natural to regard each one as a separate region in order to free the routers in one network from having to know the topological structure of the other ones.

For huge networks, a two-level hierarchy may be insufficient; it may be necessary to group the regions into clusters, the clusters into zones, the zones into groups, and so on, until we run out of

...s for aggregations. As an example of a multilevel hierarchy, consider how a packet might be ...ed from Berkeley, California, to Malindi, Kenya. The Berkeley router would know the detailed ...ology within California but would send all out-of-state traffic to the Los Angeles router. The Los ...geles router would be able to route traffic to other domestic routers but would send foreign ...ffic to New York. The New York router would be programmed to direct all traffic to the router in ...e destination country responsible for handling foreign traffic, say, in Nairobi. Finally, the packet ...ould work its way down the tree in Kenya until it got to Malindi.

Figure 5-15 gives a quantitative example of routing in a two-level hierarchy with five regions. The full routing table for router 1*A* has 17 entries, as shown in Fig. 5-15(b). When routing is done hierarchically, as in Fig. 5-15(c), there are entries for all the local routers as before, but all other regions have been condensed into a single router, so all traffic for region 2 goes via the 1*B* -2*A* line, but the rest of the remote traffic goes via the 1*C* -3*B* line. Hierarchical routing has reduced the table from 17 to 7 entries. As the ratio of the number of regions to the number of routers per region grows, the savings in table space increase.

Figure 5-15. Hierarchical routing.



(a)

Full table for 1A

Dest.	Line	Hops
1A	–	–
1B	1B	1
1C	1C	1
2A	1B	2
2B	1B	3
2C	1B	3
2D	1B	4
3A	1C	3
3B	1C	2
4A	1C	3
4B	1C	4
4C	1C	4
5A	1C	4
5B	1C	5
5C	1B	5
5D	1C	6
5E	1C	5

(b)

Hierarchical table for 1A

Dest.	Line	Hops
1A	–	–
1B	1B	1
1C	1C	1
2	1B	2
3	1C	2
4	1C	3
5	1C	4

(c)

Unfortunately, these gains in space are not free. There is a penalty to be paid, and this penalty is in the form of increased path length. For example, the best route from 1*A* to 5*C* is via region 2, but with hierarchical routing all traffic to region 5 goes via region 3, because that is better for most destinations in region 5.

When a single network becomes very large, an interesting question is: How many levels should the hierarchy have? For example, consider a subnet with 720 routers. If there is no hierarchy, each router needs 720 routing table entries. If the subnet is partitioned into 24 regions of 30 routers each, each router needs 30 local entries plus 23 remote entries for a total of 53 entries. If a three-level hierarchy is chosen, with eight clusters, each containing 9 regions of 10 routers, each router needs 10 entries for local routers, 8 entries for routing to other regions within its own cluster, and 7 entries for distant clusters, for a total of 25 entries. Kamoun and Kleinrock (1979) discovered that the optimal number of levels for an N router subnet is ln N, requiring a total of e ln N entries per router. They have also shown that the increase in effective mean path length caused by hierarchical routing is sufficiently small that it is usually acceptable.

5.2.7 Broadcast Routing

In some applications, hosts need to send messages to many or all other hosts. For example, a service distributing weather reports, stock market updates, or live radio programs might work best by broadcasting to all machines and letting those that are interested read the data. Sending a packet to all destinations simultaneously is called **broadcasting**; various methods have been proposed for doing it.

One broadcasting method that requires no special features from the subnet is for the source to simply send a distinct packet to each destination. Not only is the method wasteful of bandwidth, but it also requires the source to have a complete list of all destinations. In practice this may be the only possibility, but it is the least desirable of the methods.

Flooding is another obvious candidate. Although flooding is ill-suited for ordinary point-to-point communication, for broadcasting it might rate serious consideration, especially if none of the methods described below are applicable. The problem with flooding as a broadcast technique is the same problem it has as a point-to-point routing algorithm: it generates too many packets and consumes too much bandwidth.

A third algorithm is **multidestination routing**. If this method is used, each packet contains either a list of destinations or a bit map indicating the desired destinations. When a packet arrives at a router, the router checks all the destinations to determine the set of output lines that will be needed. (An output line is needed if it is the best route to at least one of the destinations.) The router generates a new copy of the packet for each output line to be used and includes in each packet only those destinations that are to use the line. In effect, the destination set is partitioned among the output lines. After a sufficient number of hops, each packet will carry only one destination and can be treated as a normal packet. Multidestination routing is like separately addressed packets, except that when several packets must follow the same route, one of them pays full fare and the rest ride free.

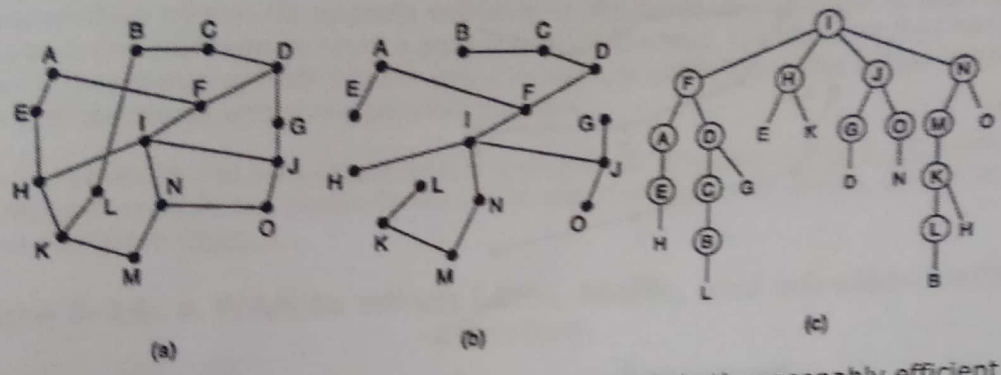
A fourth broadcast algorithm makes explicit use of the sink tree for the router initiating the broadcast—or any other convenient spanning tree for that matter. A **spanning tree** is a subset of the subnet that includes all the routers but contains no loops. If each router knows which of its lines belong to the spanning tree, it can copy an incoming broadcast packet onto all the spanning tree lines except the one it arrived on. This method makes excellent use of bandwidth, generating the absolute minimum number of packets necessary to do the job. The only problem is that each router must have knowledge of some spanning tree for the method to be applicable. Sometimes this information is available (e.g., with link state routing) but sometimes it is not (e.g., with distance vector routing).

Our last broadcast algorithm is an attempt to approximate the behavior of the previous one, even when the routers do not know anything at all about spanning trees. The idea, called **reverse path forwarding**, is remarkably simple once it has been pointed out. When a broadcast packet arrives at a router, the router checks to see if the packet arrived on the line that is normally used for sending packets *to* the source of the broadcast. If so, there is an excellent chance that the broadcast packet itself followed the best route from the router and is therefore the first copy to arrive at the router. This being the case, the router forwards copies of it onto all lines except the one it arrived on. If, however, the broadcast packet arrived on a line other than the preferred one for reaching the source, the packet is discarded as a likely duplicate.

An example of reverse path forwarding is shown in Fig. 5-16. Part (a) shows a subnet, part (b) shows a sink tree for router *I* of that subnet, and part (c) shows how the reverse path algorithm works. On the first hop, *I* sends packets to *F*, *H*, *J*, and *N*, as indicated by the second row of the tree. Each of these packets arrives on the preferred path to *I* (assuming that the preferred path falls along the sink tree) and is so indicated by a circle around the letter. On the second hop, eight packets are generated, two by each of the routers that received a packet on the first hop. As it

...e all eight of these arrive at previously unvisited routers, and five of these arrive along preferred line. Of the six packets generated on the third hop, only three arrive on the preferred path (at *C*, *E*, and *K*); the others are duplicates. After five hops and 24 packets, the broadcasting terminates, compared with four hops and 14 packets had the sink tree been followed directly.

Figure 5-16. Reverse path forwarding. (a) A subnet. (b) A sink tree. (c) The tree built by reverse path forwarding.



The principal advantage of reverse path forwarding is that it is both reasonably efficient and easy to implement. It does not require routers to know about spanning trees, nor does it have the overhead of a destination list or bit map in each broadcast packet as does multidestination addressing. Nor does it require any special mechanism to stop the process, as flooding does (either a hop counter in each packet and a priori knowledge of the subnet diameter, or a list of packets already seen per source).

5.2.8 Multicast Routing

Some applications require that widely-separated processes work together in groups, for example, a group of processes implementing a distributed database system. In these situations, it is frequently necessary for one process to send a message to all the other members of the group. If the group is small, it can just send each other member a point-to-point message. If the group is large, this strategy is expensive. Sometimes broadcasting can be used, but using broadcasting to inform 1000 machines on a million-node network is inefficient because most receivers are not interested in the message (or worse yet, they are definitely interested but are not supposed to see it). Thus, we need a way to send messages to well-defined groups that are numerically large in size but small compared to the network as a whole.

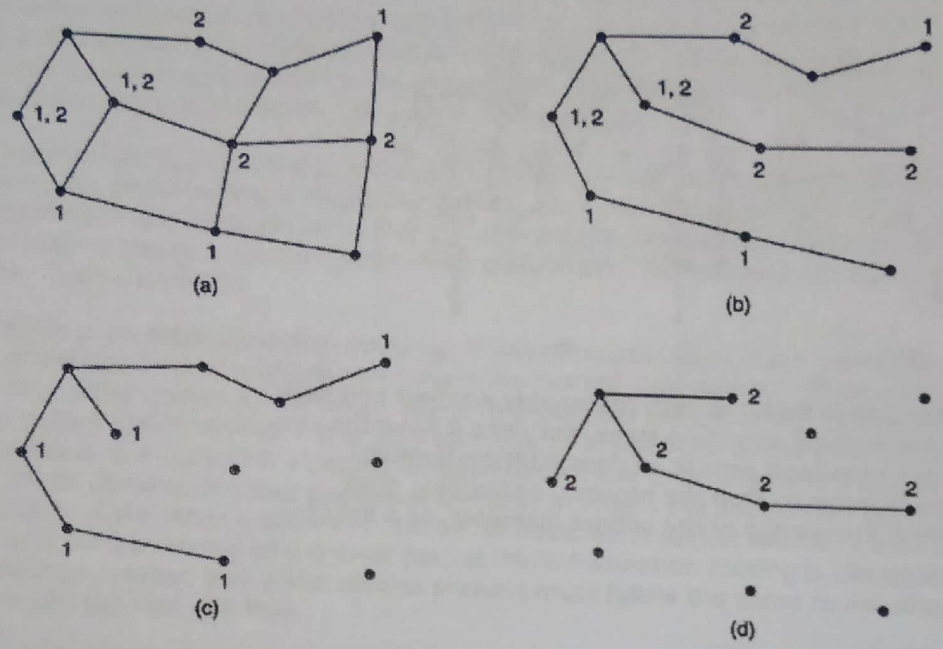
Sending a message to such a group is called **multicasting**, and its routing algorithm is called **multicast routing**. In this section we will describe one way of doing multicast routing. For additional information, see (Chu et al., 2000; Costa et al. 2001; Kasera et al., 2000; Madruga and Garcia-Luna-Aceves, 2001; Zhang and Ryu, 2001).

Multicasting requires group management. Some way is needed to create and destroy groups, and to allow processes to join and leave groups. How these tasks are accomplished is not of concern to the routing algorithm. What is of concern is that when a process joins a group, it informs its host of this fact. It is important that routers know which of their hosts belong to which groups. Either hosts must inform their routers about changes in group membership, or routers must query their hosts periodically. Either way, routers learn about which of their hosts are in which groups. Routers tell their neighbors, so the information propagates through the subnet.

To do multicast routing, each router computes a spanning tree covering all other routers. For

example, in Fig. 5-17(a) we have two groups, 1 and 2. Some routers are attached to hosts that belong to one or both of these groups, as indicated in the figure. A spanning tree for the leftmost router is shown in Fig. 5-17(b).

Figure 5-17. (a) A network. (b) A spanning tree for the leftmost router. (c) A multicast tree for group 1. (d) A multicast tree for group 2.



When a process sends a multicast packet to a group, the first router examines its spanning tree and prunes it, removing all lines that do not lead to hosts that are members of the group. In our example, Fig. 5-17(c) shows the pruned spanning tree for group 1. Similarly, Fig. 5-17(d) shows the pruned spanning tree for group 2. Multicast packets are forwarded only along the appropriate spanning tree.

Various ways of pruning the spanning tree are possible. The simplest one can be used if link state routing is used and each router is aware of the complete topology, including which hosts belong to which groups. Then the spanning tree can be pruned, starting at the end of each path, working toward the root, and removing all routers that do not belong to the group in question.

With distance vector routing, a different pruning strategy can be followed. The basic algorithm is reverse path forwarding. However, whenever a router with no hosts interested in a particular group and no connections to other routers receives a multicast message for that group, it responds with a PRUNE message, telling the sender not to send it any more multicasts for that group. When a router with no group members among its own hosts has received such messages on all its lines, it, too, can respond with a PRUNE message. In this way, the subnet is recursively pruned.

One potential disadvantage of this algorithm is that it scales poorly to large networks. Suppose that a network has n groups, each with an average of m members. For each group, m pruned spanning trees must be stored, for a total of mn trees. When many large groups exist, considerable storage is needed to store all the trees.

An alternative design uses **core-based trees** (Ballardie et al., 1993). Here, a single spanning tree per group is computed, with the root (the core) near the middle of the group. To send a

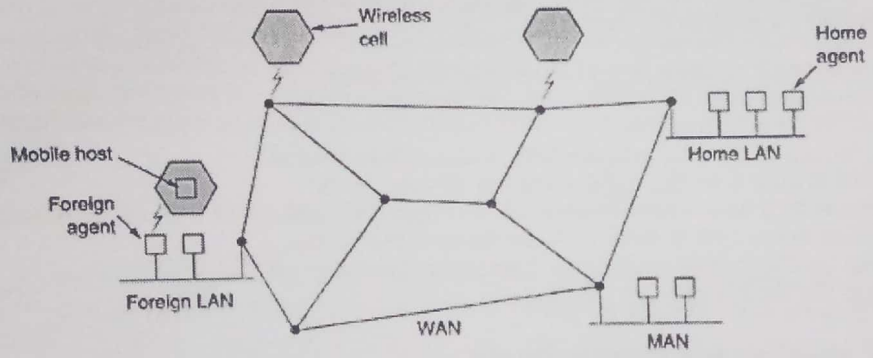
...st message, a host sends it to the core, which then does the multicast along the spanning ...Although this tree will not be optimal for all sources, the reduction in storage costs from *m* ...s to one tree per group is a major saving.

5.2.9 Routing for Mobile Hosts

Millions of people have portable computers nowadays, and they generally want to read their e-mail and access their normal file systems wherever in the world they may be. These mobile hosts introduce a new complication: to route a packet to a mobile host, the network first has to find it. The subject of incorporating mobile hosts into a network is very young, but in this section we will sketch some of the issues and give a possible solution.

The model of the world that network designers typically use is shown in Fig. 5-18. Here we have a WAN consisting of routers and hosts. Connected to the WAN are LANs, MANs, and wireless cells of the type we studied in Chap. 2.

Figure 5-18. A WAN to which LANs, MANs, and wireless cells are attached.



Hosts that never move are said to be stationary. They are connected to the network by copper wires or fiber optics. In contrast, we can distinguish two other kinds of hosts. Migratory hosts are basically stationary hosts who move from one fixed site to another from time to time but use the network only when they are physically connected to it. Roaming hosts actually compute on the run and want to maintain their connections as they move around. We will use the term **mobile hosts** to mean either of the latter two categories, that is, all hosts that are away from home and still want to be connected.

All hosts are assumed to have a permanent **home location** that never changes. Hosts also have a permanent home address that can be used to determine their home locations, analogous to the way the telephone number 1-212-5551212 indicates the United States (country code 1) and Manhattan (212). The routing goal in systems with mobile hosts is to make it possible to send packets to mobile hosts using their home addresses and have the packets efficiently reach them wherever they may be. The trick, of course, is to find them.

In the model of Fig. 5-18, the world is divided up (geographically) into small units. Let us call them areas, where an area is typically a LAN or wireless cell. Each area has one or more **foreign agents**, which are processes that keep track of all mobile hosts visiting the area. In addition, each area has a **home agent**, which keeps track of hosts whose home is in the area, but who are currently visiting another area.

When a new host enters an area, either by connecting to it (e.g., plugging into the LAN) or just

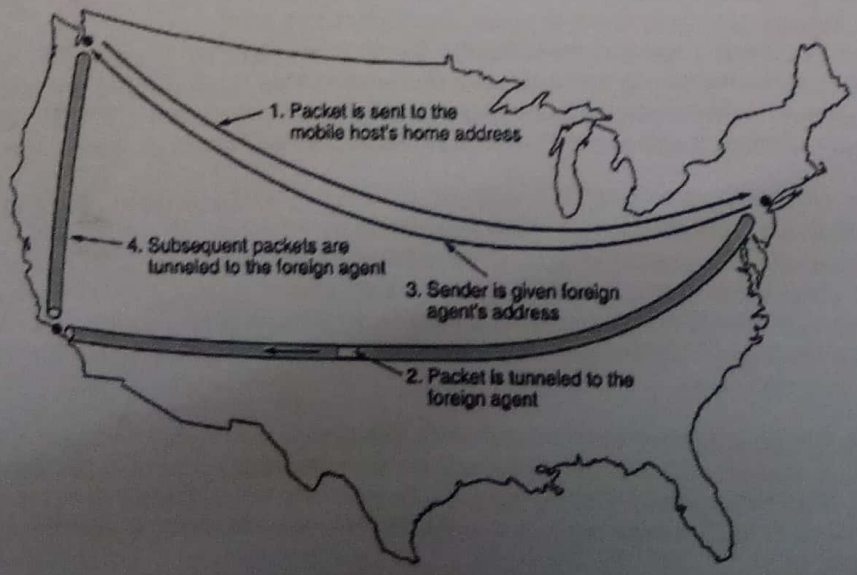
wandering into the cell, his computer must register itself with the foreign agent there. The registration procedure typically works like this:

1. Periodically, each foreign agent broadcasts a packet announcing its existence and address. A newly-arrived mobile host may wait for one of these messages, but if none arrives quickly enough, the mobile host can broadcast a packet saying: Are there any foreign agents around?
2. The mobile host registers with the foreign agent, giving its home address, current data link layer address, and some security information.
3. The foreign agent contacts the mobile host's home agent and says: One of your hosts is over here. The message from the foreign agent to the home agent contains the foreign agent's network address. It also includes the security information to convince the home agent that the mobile host is really there.
4. The home agent examines the security information, which contains a timestamp, to prove that it was generated within the past few seconds. If it is happy, it tells the foreign agent to proceed.
5. When the foreign agent gets the acknowledgement from the home agent, it makes an entry in its tables and informs the mobile host that it is now registered.

Ideally, when a host leaves an area, that, too, should be announced to allow deregistration, but many users abruptly turn off their computers when done.

When a packet is sent to a mobile host, it is routed to the host's home LAN because that is what the address says should be done, as illustrated in step 1 of Fig. 5-19. Here the sender, in the northwest city of Seattle, wants to send a packet to a host normally across the United States in New York. Packets sent to the mobile host on its home LAN in New York are intercepted by the home agent there. The home agent then looks up the mobile host's new (temporary) location and finds the address of the foreign agent handling the mobile host, in Los Angeles.

Figure 5-19. Packet routing for mobile hosts.



The home agent then does two things. First, it encapsulates the packet in the payload field of an

...r packet and sends the latter to the foreign agent (step 2 in Fig. 5-19). This mechanism is ...ed tunneling; we will look at it in more detail later. After getting the encapsulated packet, the ...reign agent removes the original packet from the payload field and sends it to the mobile host ...a data link frame.

Second, the home agent tells the sender to henceforth send packets to the mobile host by encapsulating them in the payload of packets explicitly addressed to the foreign agent instead of just sending them to the mobile host's home address (step 3). Subsequent packets can now be routed directly to the host via the foreign agent (step 4), bypassing the home location entirely.

The various schemes that have been proposed differ in several ways. First, there is the issue of how much of this protocol is carried out by the routers and how much by the hosts, and in the latter case, by which layer in the hosts. Second, in a few schemes, routers along the way record mapped addresses so they can intercept and redirect traffic even before it gets to the home location. Third, in some schemes each visitor is given a unique temporary address; in others, the temporary address refers to an agent that handles traffic for all visitors.

Fourth, the schemes differ in how they actually manage to arrange for packets that are addressed to one destination to be delivered to a different one. One choice is changing the destination address and just retransmitting the modified packet. Alternatively, the whole packet, home address and all, can be encapsulated inside the payload of another packet sent to the temporary address. Finally, the schemes differ in their security aspects. In general, when a host or router gets a message of the form "Starting right now, please send all of Stephany's mail to me," it might have a couple of questions about whom it was talking to and whether this is a good idea. Several mobile host protocols are discussed and compared in (Hac and Guo, 2000; Perkins, 1998a; Snoeren and Balakrishnan, 2000; Solomon, 1998; and Wang and Chen, 2001).

5.2.10 Routing in Ad Hoc Networks

We have now seen how to do routing when the hosts are mobile but the routers are fixed. An even more extreme case is one in which the routers themselves are mobile. Among the possibilities are:

1. Military vehicles on a battlefield with no existing infrastructure.
2. A fleet of ships at sea.
3. Emergency workers at an earthquake that destroyed the infrastructure.
4. A gathering of people with notebook computers in an area lacking 802.11.

In all these cases, and others, each node consists of a router and a host, usually on the same computer. Networks of nodes that just happen to be near each other are called **ad hoc networks** or **MANETs** (**Mobile Ad hoc NETworks**). Let us now examine them briefly. More information can be found in (Perkins, 2001).

What makes ad hoc networks different from wired networks is that all the usual rules about fixed topologies, fixed and known neighbors, fixed relationship between IP address and location, and more are suddenly tossed out the window. Routers can come and go or appear in new places at the drop of a bit. With a wired network, if a router has a valid path to some destination, that path continues to be valid indefinitely (barring a failure somewhere in the system). With an ad hoc network, the topology may be changing all the time, so desirability and even validity of paths can change spontaneously, without warning. Needless to say, these circumstances make routing in ad hoc networks quite different from routing in their fixed counterparts.

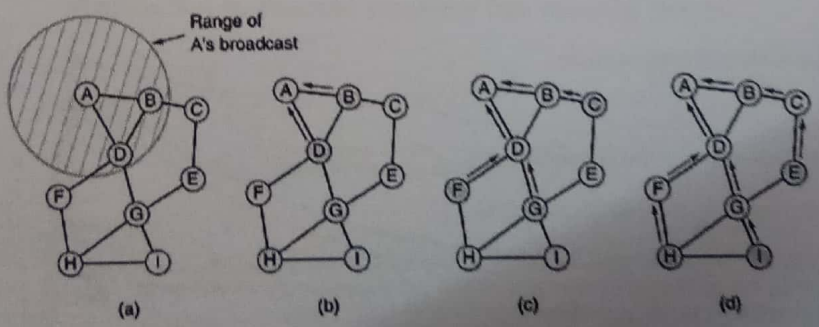
A variety of routing algorithms for ad hoc networks have been proposed. One of the more interesting ones is the **AODV** (**Ad hoc On-demand Distance Vector**) routing algorithm (Perkins and Royer, 1999). It is a distant relative of the Bellman-Ford distance vector algorithm but adapted to work in a mobile environment and takes into account the limited bandwidth and low battery life found in this environment. Another unusual characteristic is that it is an on-demand algorithm, that is, it determines a route to some destination only when somebody wants to send a packet to that destination. Let us now see what that means.

Route Discovery

At any instant of time, an ad hoc network can be described by a graph of the nodes (routers + hosts). Two nodes are connected (i.e., have an arc between them in the graph) if they can communicate directly using their radios. Since one of the two may have a more powerful transmitter than the other, it is possible that *A* is connected to *B* but *B* is not connected to *A*. However, for simplicity, we will assume all connections are symmetric. It should also be noted that the mere fact that two nodes are within radio range of each other does not mean that they are connected. There may be buildings, hills, or other obstacles that block their communication.

To describe the algorithm, consider the ad hoc network of Fig. 5-20, in which a process at node *A* wants to send a packet to node *I*. The AODV algorithm maintains a table at each node, keyed by destination, giving information about that destination, including which neighbor to send packets to in order to reach the destination. Suppose that *A* looks in its table and does not find an entry for *I*. It now has to discover a route to *I*. This property of discovering routes only when they are needed is what makes this algorithm "on demand."

Figure 5-20. (a) Range of *A*'s broadcast. (b) After *B* and *D* have received *A*'s broadcast. (c) After *C*, *F*, and *G* have received *A*'s broadcast. (d) After *E*, *H*, and *I* have received *A*'s broadcast. The shaded nodes are new recipients. The arrows show the possible reverse routes.



To locate *I*, *A* constructs a special ROUTE REQUEST packet and broadcasts it. The packet reaches *B* and *D*, as illustrated in Fig. 5-20(a). In fact, the reason *B* and *D* are connected to *A* in the graph is that they can receive communication from *A*. *F*, for example, is not shown with an arc to *A* because it cannot receive *A*'s radio signal. Thus, *F* is not connected to *A*.

The format of the ROUTE REQUEST packet is shown in Fig. 5-21. It contains the source and destination addresses, typically their IP addresses, which identify who is looking for whom. It also contains a *Request ID*, which is a local counter maintained separately by each node and incremented each time a ROUTE REQUEST is broadcast. Together, the *Source address* and *Request ID* fields uniquely identify the ROUTE REQUEST packet to allow nodes to discard any

...icates they may receive.

Figure 5-21. Format of a ROUTE REQUEST packet.

Source address	Request ID	Destination address	Source sequence #	Dest. sequence #	Hop count

In addition to the *Request ID* counter, each node also maintains a second sequence counter incremented whenever a ROUTE REQUEST is sent (or a reply to someone else's ROUTE REQUEST). It functions a little bit like a clock and is used to tell new routes from old routes. The fourth field of Fig. 5-21 is *A*'s sequence counter; the fifth field is the most recent value of *I*'s sequence number that *A* has seen (0 if it has never seen it). The use of these fields will become clear shortly. The final field, *Hop count*, will keep track of how many hops the packet has made. It is initialized to 0.

When a ROUTE REQUEST packet arrives at a node (*B* and *D* in this case), it is processed in the following steps.

1. The (*Source address*, *Request ID*) pair is looked up in a local history table to see if this request has already been seen and processed. If it is a duplicate, it is discarded and processing stops. If it is not a duplicate, the pair is entered into the history table so future duplicates can be rejected, and processing continues.
2. The receiver looks up the destination in its route table. If a fresh route to the destination is known, a ROUTE REPLY packet is sent back to the source telling it how to get to the destination (basically: Use me). Fresh means that the *Destination sequence number* stored in the routing table is greater than or equal to the *Destination sequence number* in the ROUTE REQUEST packet. If it is less, the stored route is older than the previous route the source had for the destination, so step 3 is executed.
3. Since the receiver does not know a fresh route to the destination, it increments the *Hop count* field and rebroadcasts the ROUTE REQUEST packet. It also extracts the data from the packet and stores it as a new entry in its reverse route table. This information will be used to construct the reverse route so that the reply can get back to the source later. The arrows in Fig. 5-20 are used for building the reverse route. A timer is also started for the newly-made reverse route entry. If it expires, the entry is deleted.

Neither *B* nor *D* knows where *I* is, so each of them creates a reverse route entry pointing back to *A*, as shown by the arrows in Fig. 5-20, and broadcasts the packet with *Hop count* set to 1. The broadcast from *B* reaches *C* and *D*. *C* makes an entry for it in its reverse route table and rebroadcasts it. In contrast, *D* rejects it as a duplicate. Similarly, *D*'s broadcast is rejected by *B*. However, *D*'s broadcast is accepted by *F* and *G* and stored, as shown in Fig. 5-20(c). After *E*, *H*, and *I* receive the broadcast, the ROUTE REQUEST finally reaches a destination that knows where *I* is, namely, *I* itself, as illustrated in Fig. 5-20(d). Note that although we have shown the broadcasts in three discrete steps here, the broadcasts from different nodes are not coordinated in any way.

In response to the incoming request, *I* builds a ROUTE REPLY packet, as shown in Fig. 5-22. The *Source address*, *Destination address*, and *Hop count* are copied from the incoming request, but the *Destination sequence number* taken from its counter in memory. The *Hop count* field is set to 0. The *Lifetime* field controls how long the route is valid. This packet is unicast to the node that the ROUTE REQUEST packet came from, in this case, *G*. It then follows the reverse path to *D* and finally to *A*. At each node, *Hop count* is incremented so the node can see how far from the destination (*I*) it is.

Figure 5-22. Format of a ROUTE REPLY packet.

Source address	Destination address	Destination sequence #	Hop count	Lifetime

At each intermediate node on the way back, the packet is inspected. It is entered into the local routing table as a route to *I* if one or more of the following three conditions are met:

1. No route to *I* is known.
2. The sequence number for *I* in the ROUTE REPLY packet is greater than the value in the routing table.
3. The sequence numbers are equal but the new route is shorter.

In this way, all the nodes on the reverse route learn the route to *I* for free, as a byproduct of *A*'s route discovery. Nodes that got the original REQUEST ROUTE packet but were not on the reverse path (*B*, *C*, *E*, *F*, and *H* in this example) discard the reverse route table entry when the associated timer expires.

In a large network, the algorithm generates many broadcasts, even for destinations that are close by. The number of broadcasts can be reduced as follows. The IP packet's *Time to live* is initialized by the sender to the expected diameter of the network and decremented on each hop. If it hits 0, the packet is discarded instead of being broadcast.

The discovery process is then modified as follows. To locate a destination, the sender broadcasts a ROUTE REQUEST packet with *Time to live* set to 1. If no response comes back within a reasonable time, another one is sent, this time with *Time to live* set to 2. Subsequent attempts use 3, 4, 5, etc. In this way, the search is first attempted locally, then in increasingly wider rings.

Route Maintenance

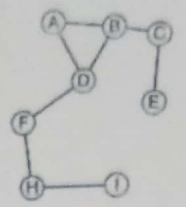
Because nodes can move or be switched off, the topology can change spontaneously. For example, in Fig. 5-20, if *G* is switched off, *A* will not realize that the route it was using to *I* (*ADGI*) is no longer valid. The algorithm needs to be able to deal with this. Periodically, each node broadcasts a *Hello* message. Each of its neighbors is expected to respond to it. If no response is forthcoming, the broadcaster knows that that neighbor has moved out of range and is no longer connected to it. Similarly, if it tries to send a packet to a neighbor that does not respond, it learns that the neighbor is no longer available.

This information is used to purge routes that no longer work. For each possible destination, each node, *N*, keeps track of its neighbors that have fed it a packet for that destination during the last ΔT seconds. These are called *N*'s **active neighbors** for that destination. *N* does this by having a routing table keyed by destination and containing the outgoing node to use to reach the destination, the hop count to the destination, the most recent destination sequence number, and the list of active neighbors for that destination. A possible routing table for node *D* in our example topology is shown in Fig. 5-23(a).

Figure 5-23. (a) *D*'s routing table before *G* goes down. (b) The graph after *G* has gone down.

Dest.	Next hop	Distance	Active neighbors	Other fields
A	A	1	F, G	
B	B	1	F, G	
C	B	2	F	
E	G	2		
F	F	1	A, B	
G	G	1	A, B	
H	F	2	A, B	
I	G	2	A, B	

(a)



(b)

When any of *N*'s neighbors becomes unreachable, it checks its routing table to see which destinations have routes using the now-gone neighbor. For each of these routes, the active neighbors are informed that their route via *N* is now invalid and must be purged from their routing tables. The active neighbors then tell their active neighbors, and so on, recursively, until all routes depending on the now-gone node are purged from all routing tables.

As an example of route maintenance, consider our previous example, but now with *G* suddenly switched off. The changed topology is illustrated in Fig. 5-23(b). When *D* discovers that *G* is gone, it looks at its routing table and sees that *G* was used on routes to *E*, *G*, and *I*. The union of the active neighbors for these destinations is the set {*A*, *B*}. In other words, *A* and *B* depend on *G* for some of their routes, so they have to be informed that these routes no longer work. *D* tells them by sending them packets that cause them to update their own routing tables accordingly. *D* also purges the entries for *E*, *G*, and *I* from its routing table.

It may not have been obvious from our description, but a critical difference between AODV and Bellman-Ford is that nodes do not send out periodic broadcasts containing their entire routing table. This difference saves both bandwidth and battery life.

AODV is also capable of doing broadcast and multicast routing. For details, consult (Perkins and Royer, 2001). Ad hoc routing is a red-hot research area. A great deal has been published on the topic. A few of the papers include (Chen et al., 2002; Hu and Johnson, 2001; Li et al., 2001; Raju and Garcia-Luna-Aceves, 2001; Ramanathan and Redi, 2002; Royer and Toh, 1999; Spohn and Garcia-Luna-Aceves, 2001; Tseng et al., 2001; and Zadeh et al., 2002).

5.2.11 Node Lookup in Peer-to-Peer Networks

A relatively new phenomenon is peer-to-peer networks, in which a large number of people, usually with permanent wired connections to the Internet, are in contact to share resources. The first widespread application of peer-to-peer technology was for mass crime: 50 million Napster users were exchanging copyrighted songs without the copyright owners' permission until Napster was shut down by the courts amid great controversy. Nevertheless, peer-to-peer technology has many interesting and legal uses. It also has something similar to a routing problem, although it is not quite the same as the ones we have studied so far. Nevertheless, it is worth a quick look.

What makes peer-to-peer systems interesting is that they are totally distributed. All nodes are symmetric and there is no central control or hierarchy. In a typical peer-to-peer system the users each have some information that may be of interest to other users. This information may be free software, (public domain) music, photographs, and so on. If there are large numbers of users, they will not know each other and will not know where to find what they are looking for. One solution is a big central database, but this may not be feasible for some reason (e.g., nobody is willing to host and maintain it). Thus, the problem comes down to how a user finds a node that contains what he is looking for in the absence of a centralized database or even a centralized

* Internetworking
* N/w layer in internet
* Token bucket