

Larry L. Peterson and Bruce S. Davie

Fifth Edition

Computer Networks

a systems approach

MK
MORGAN KAUFMANN

www.EBooksWorld.ir

Acquiring Editor: Rick Adams
Development Editor: Nate McFadden
Project Manager: Paul Gottehrer
Designer: Dennis Schaefer

Morgan Kaufmann is an imprint of Elsevier
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA

© 2012 Elsevier, Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: www.elsevier.com/permissions.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods or professional practices, may become necessary. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information or methods described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data

Peterson, Larry L.

Computer networks : a systems approach / Larry L. Peterson and Bruce S. Davie. – 5th ed.

p. cm. – (The Morgan Kaufmann series in networking)

Includes bibliographical references.

ISBN 978-0-12-385059-1 (hardback)

1. Computer networks. I. Davie, Bruce S. II. Title.

TK5105.5.P479 2011

004.6–dc22

2011000786

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

ISBN: 978-0-12-385059-1

For information on all Morgan Kaufmann publications
visit our website at www.mkp.com

Typeset by: diacriTech, India

Printed in the United States of America

11 12 13 14 15 16 10 9 8 7 6 5 4 3 2

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER

BOOK AID
International

Sabre Foundation

www.EBooksWorld.ir

Internetworking 3

Nature seems ... to reach many of her ends by long circuitous routes.

—Rudolph Lotze

In the previous chapter, we saw how to connect one node to another or to an existing network. Many technologies can be used to build “last-mile” links or to connect a modest number of nodes together, but how do we build networks of global scale? A single Ethernet can interconnect no more than 1024 hosts; a point-to-point link connects only two. Wireless networks are limited by the ranges of their radios. To build a global network, we need a way to interconnect these different types of links and networks. The concept of interconnecting different types of networks to

PROBLEM: NOT ALL NETWORKS ARE DIRECTLY CONNECTED

build a large, global network is the core idea of the Internet and is often referred to as *internetworking*.

We can divide the internetworking problem up into a few subproblems. First of all, we need a way to interconnect links. Devices that interconnect links of the same type are often called *switches*, and these devices are the first topic of this chapter. A particularly important class of switches today are those used to interconnect Ethernet segments; these switches are also sometimes called *bridges*. The core job of a switch is to take packets that arrive on an input and *forward*

Computer Networks: A Systems Approach. DOI: 10.1016/B978-0-12-385059-1.00003-X
Copyright © 2012 Elsevier, Inc. All rights reserved.

www.EBooksWorld.ir

(or *switch*) them to the right output so that they will reach their appropriate destination. There are a variety of ways that the switch can determine the “right” output for a packet, which can be broadly categorized as connectionless and connection-oriented approaches. These two approaches have both found important application areas over the years.

Given the enormous diversity of network types, we also need a way to interconnect disparate networks and links (i.e., deal with *heterogeneity*). Devices that perform this task, once called *gateways*, are now mostly known as *routers*. The protocol that was invented to deal with interconnection of disparate network types, the Internet Protocol (IP), is the topic of our second section.

Once we interconnect a whole lot of links and networks with switches and routers, there are likely to be many different possible ways to get from one point to another. Finding a suitable path or *route* through a network is one of the fundamental problems of networking. Such paths should be efficient (e.g., no longer than necessary), loop free, and able to respond to the fact that networks are not static—nodes may fail or reboot, links may break, and new nodes or links may be added. Our third section looks at some of the algorithms and protocols that have been developed to address these issues.

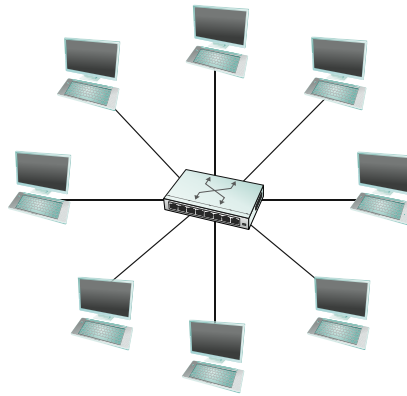
Once we understand the problems of switching and routing, we need some devices to perform those functions. This chapter concludes with some discussion of the ways switches and routers are built. While many packet switches and routers are quite similar to a general-purpose computer, there are many situations where more specialized designs are used. This is particularly the case at the high end, where there seems to be a never-ending need for bigger and faster routers to handle the ever-increasing traffic load in the Internet’s core.



3.1 SWITCHING AND BRIDGING

In the simplest terms, a switch is a mechanism that allows us to interconnect links to form a larger network. A switch is a multi-input, multi-output device that transfers packets from an input to one or more outputs. Thus, a switch adds the star topology (see [Figure 3.1](#)) to the point-to-point link, bus (Ethernet), and ring topologies established in the last chapter. A star topology has several attractive properties:

- Even though a switch has a fixed number of inputs and outputs, which limits the number of hosts that can be connected to a single switch, large networks can be built by interconnecting a number of switches.



■ FIGURE 3.1 A switch provides a star topology.

- We can connect switches to each other and to hosts using point-to-point links, which typically means that we can build networks of large geographic scope.
- Adding a new host to the network by connecting it to a switch does not necessarily reduce the performance of the network for other hosts already connected.

This last claim cannot be made for the shared-media networks discussed in the last chapter. For example, it is impossible for two hosts on the same 10-Mbps Ethernet segment to transmit continuously at 10 Mbps because they share the same transmission medium. Every host on a switched network has its own link to the switch, so it may be entirely possible for many hosts to transmit at the full link speed (bandwidth), provided that the switch is designed with enough aggregate capacity. Providing high aggregate throughput is one of the design goals for a switch; we return to this topic later. In general, switched networks are considered more *scalable* (i.e., more capable of growing to large numbers of nodes) than shared-media networks because of this ability to support many hosts at full speed.

A switch is connected to a set of links and, for each of these links, runs the appropriate data link protocol to communicate with the node at the other end of the link. A switch's primary job is to receive incoming packets on one of its links and to transmit them on some other link. This function is sometimes referred to as either *switching* or *forwarding*, and in

terms of the Open Systems Interconnection (OSI) architecture, it is the main function of the network layer.

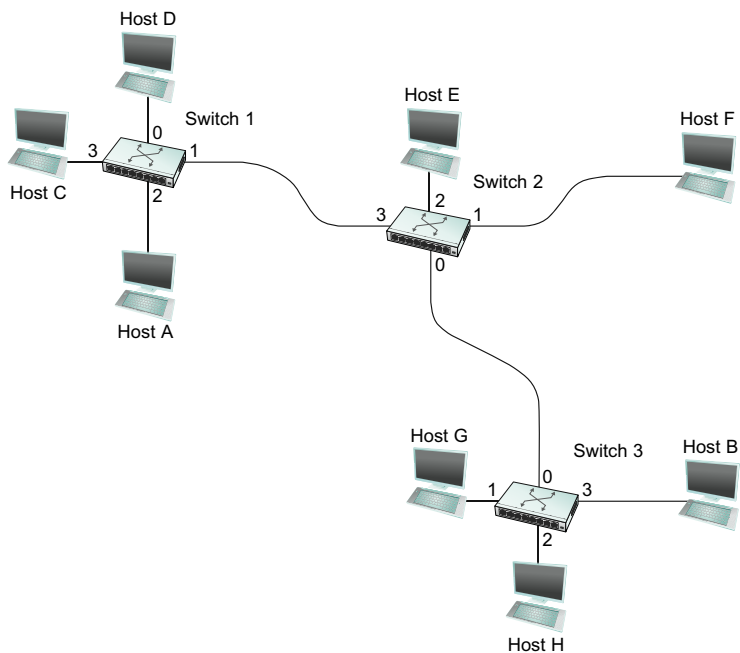
The question, then, is how does the switch decide which output link to place each packet on? The general answer is that it looks at the header of the packet for an identifier that it uses to make the decision. The details of how it uses this identifier vary, but there are two common approaches. The first is the *datagram* or *connectionless* approach. The second is the *virtual circuit* or *connection-oriented* approach. A third approach, *source routing*, is less common than these other two, but it does have some useful applications.

One thing that is common to all networks is that we need to have a way to identify the end nodes. Such identifiers are usually called *addresses*. We have already seen examples of addresses in the previous chapter, such as the 48-bit address used for Ethernet. The only requirement for Ethernet addresses is that no two nodes on a network have the same address. This is accomplished by making sure that all Ethernet cards are assigned a *globally unique* identifier. For the following discussions, we assume that each host has a globally unique address. Later on, we consider other useful properties that an address might have, but global uniqueness is adequate to get us started.

Another assumption that we need to make is that there is some way to identify the input and output ports of each switch. There are at least two sensible ways to identify ports: One is to number each port, and the other is to identify the port by the name of the node (switch or host) to which it leads. For now, we use numbering of the ports.

3.1.1 Datagrams

The idea behind datagrams is incredibly simple: You just include in every packet enough information to enable any switch to decide how to get it to its destination. That is, every packet contains the complete destination address. Consider the example network illustrated in Figure 3.2, in which the hosts have addresses A, B, C, and so on. To decide how to forward a packet, a switch consults a *forwarding table* (sometimes called a *routing table*), an example of which is depicted in Table 3.1. This particular table shows the forwarding information that switch 2 needs to forward datagrams in the example network. It is pretty easy to figure out such a table when you have a complete map of a simple network like that depicted here; we could imagine a network operator configuring



■ FIGURE 3.2 Datagram forwarding: an example network.

Table 3.1 Forwarding Table for Switch 2	
Destination	Port
A	3
B	0
C	3
D	3
E	2
F	1
G	0
H	0

the tables statically. It is a lot harder to create the forwarding tables in large, complex networks with dynamically changing topologies and multiple paths between destinations. That harder problem is known as *routing* and is the topic of Section 3.3. We can think of routing as a process that takes place in the background so that, when a data packet turns up, we will have the right information in the forwarding table to be able to forward, or switch, the packet.

Datagram networks have the following characteristics:

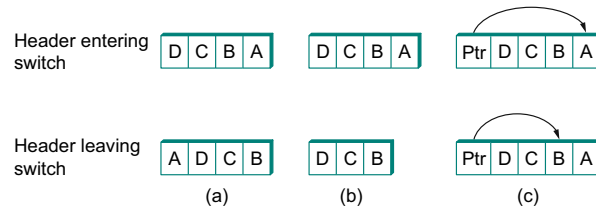
- A host can send a packet anywhere at any time, since any packet that turns up at a switch can be immediately forwarded (assuming a correctly populated forwarding table). For this reason, datagram networks are often called *connectionless*; this contrasts with the *connection-oriented* networks described below, in which some *connection state* needs to be established before the first data packet is sent.
- When a host sends a packet, it has no way of knowing if the network is capable of delivering it or if the destination host is even up and running.
- Each packet is forwarded independently of previous packets that might have been sent to the same destination. Thus, two successive packets from host A to host B may follow completely different paths (perhaps because of a change in the forwarding table at some switch in the network).
- A switch or link failure might not have any serious effect on communication if it is possible to find an alternate route around the failure and to update the forwarding table accordingly.

This last fact is particularly important to the history of datagram networks. One of the important design goals of the Internet is robustness to failures, and history has shown it to be quite effective at meeting this goal.¹

3.1.2 Virtual Circuit Switching

A second technique for packet switching, which differs significantly from the datagram model, uses the concept of a *virtual circuit* (VC).

¹The oft-repeated claim that the ARPANET was built to withstand nuclear attack does not appear to be substantiated by those who actually worked on its design, but robustness to failure of individual components was certainly a goal.



■ **FIGURE 3.8** Three ways to handle headers for source routing: (a) rotation; (b) stripping; (c) pointer. The labels are read right to left.

Figure 3.8. In each case, the entry that this switch needs to read is A, and the entry that the next switch needs to read is B.

Source routing can be used in both datagram networks and virtual circuit networks. For example, the Internet Protocol, which is a datagram protocol, includes a source route option that allows selected packets to be source routed, while the majority are switched as conventional datagrams. Source routing is also used in some virtual circuit networks as the means to get the initial setup request along the path from source to destination.

Source routes are sometimes categorized as “strict” or “loose.” In a strict source route, every node along the path must be specified, whereas a loose source route only specifies a set of nodes to be traversed, without saying exactly how to get from one node to the next. A loose source route can be thought of as a set of waypoints rather than a completely specified route. The loose option can be helpful to limit the amount of information that a source must obtain to create a source route. In any reasonably large network, it is likely to be hard for a host to get the complete path information it needs to construct a correct strict source route to any destination. But both types of source routes do find application in certain scenarios, one of which is described in [Section 4.3](#).

3.1.4 Bridges and LAN Switches

Having discussed some of the basic ideas behind switching, we now focus more closely on some specific switching technologies. We begin by considering a class of switch that is used to forward packets between LANs (local area networks) such as Ethernets. Such switches are sometimes known by the obvious name of LAN switches; historically, they have also



LAB 04:
VLAN

been referred to as *bridges*, and they are very widely used in campus and enterprise networks.

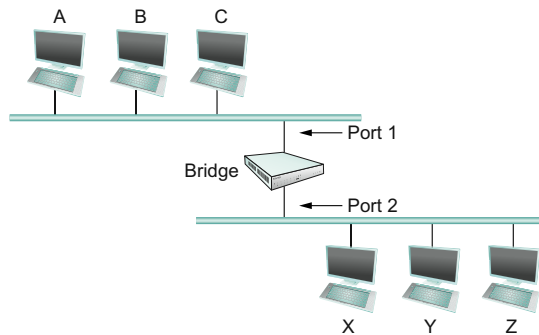
Suppose you have a pair of Ethernets that you want to interconnect. One approach you might try is to put a repeater between them, as described in [Chapter 2](#). This would not be a workable solution, however, if doing so exceeded the physical limitations of the Ethernet. (Recall that no more than two repeaters between any pair of hosts and no more than a total of 2500 m in length are allowed.) An alternative would be to put a node with a pair of Ethernet adaptors between the two Ethernets and have the node forward frames from one Ethernet to the other. This node would differ from a repeater, which operates on bits, not frames, and just blindly copies the bits received on one interface to another. Instead, this node would fully implement the Ethernet's collision detection and media access protocols on each interface. Hence, the length and number-of-host restrictions of the Ethernet, which are all about managing collisions, would not apply to the combined pair of Ethernets connected in this way. This device operates in promiscuous mode, accepting all frames transmitted on either of the Ethernets, and forwarding them to the other.

The node we have just described is typically called a *bridge*, and a collection of LANs connected by one or more bridges is usually said to form an *extended LAN*. In their simplest variants, bridges simply accept LAN frames on their inputs and forward them out on all other outputs. This simple strategy was used by early bridges but has some pretty serious limitations as we'll see below. A number of refinements have been added over the years to make bridges an effective mechanism for interconnecting a set of LANs. The rest of this section fills in the more interesting details.

Note that a bridge meets our definition of a switch from the previous section: a multi-input, multi-output device, which transfers packets from an input to one or more outputs. And recall that this provides a way to increase the total bandwidth of a network. For example, while a single Ethernet segment might carry only 100 Mbps of total traffic, an Ethernet bridge can carry as much as $100n$ Mbps, where n is the number of ports (inputs and outputs) on the bridge.

Learning Bridges

The first optimization we can make to a bridge is to observe that it need not forward all frames that it receives. Consider the bridge in [Figure 3.9](#). Whenever a frame from host A that is addressed to host B arrives on port 1,



■ FIGURE 3.9 Illustration of a learning bridge.

Table 3.4 Forwarding Table Maintained by a Bridge

Host	Port
A	1
B	1
C	1
X	2
Y	2
Z	2

there is no need for the bridge to forward the frame out over port 2. The question, then, is how does a bridge come to learn on which port the various hosts reside?

One option would be to have a human download a table into the bridge similar to the one given in Table 3.4. Then, whenever the bridge receives a frame on port 1 that is addressed to host A, it would not forward the frame out on port 2; there would be no need because host A would have already directly received the frame on the LAN connected to port 1. Anytime a frame addressed to host A was received on port 2, the bridge would forward the frame out on port 1.

No one actually builds bridges in which the table is configured by hand. Having a human maintain this table is too burdensome, and there is a

simple trick by which a bridge can learn this information for itself. The idea is for each bridge to inspect the *source* address in all the frames it receives. Thus, when host A sends a frame to a host on either side of the bridge, the bridge receives this frame and records the fact that a frame from host A was just received on port 1. In this way, the bridge can build a table just like Table 3.4.

Note that a bridge using such a table implements a version of the datagram (or connectionless) model of forwarding described in Section 3.1.1. Each packet carries a global address, and the bridge decides which output to send a packet on by looking up that address in a table.

When a bridge first boots, this table is empty; entries are added over time. Also, a timeout is associated with each entry, and the bridge discards the entry after a specified period of time. This is to protect against the situation in which a host—and, as a consequence, its LAN address—is moved from one network to another. Thus, this table is not necessarily complete. Should the bridge receive a frame that is addressed to a host not currently in the table, it goes ahead and forwards the frame out on all the other ports. In other words, this table is simply an optimization that filters out some frames; it is not required for correctness.

Implementation

The code that implements the learning bridge algorithm is quite simple, and we sketch it here. Structure `BridgeEntry` defines a single entry in the bridge's forwarding table; these are stored in a `Map` structure (which supports `mapCreate`, `mapBind`, and `mapResolve` operations) to enable entries to be efficiently located when packets arrive from sources already in the table. The constant `MAX_TTL` specifies how long an entry is kept in the table before it is discarded.

```
#define BRIDGE_TAB_SIZE    1024    /* max. size of bridging
                                     table */
#define MAX_TTL             120    /* time (in seconds) before
                                     an entry is flushed */

typedef struct {
    MacAddr    destination;    /* MAC address of a node */
    int        ifnumber;       /* interface to reach it */
    u_short    TTL;            /* time to live */
    Binding    binding;        /* binding in the Map */
}
```

```

} BridgeEntry;

int      numEntries = 0;
Map      bridgeMap = mapCreate(BRIDGE_TAB_SIZE,
                               sizeof(BridgeEntry));

```

The routine that updates the forwarding table when a new packet arrives is given by `updateTable`. The arguments passed are the source media access control (MAC) address contained in the packet and the interface number on which it was received. Another routine, not shown here, is invoked at regular intervals, scans the entries in the forwarding table, and decrements the TTL (time to live) field of each entry, discarding any entries whose TTL has reached 0. Note that the TTL is reset to `MAX_TTL` every time a packet arrives to refresh an existing table entry and that the interface on which the destination can be reached is updated to reflect the most recently received packet.

```

void
updateTable (MacAddr src, int inif)
{
    BridgeEntry      *b;

    if (mapResolve(bridgeMap, &src, (void **)&b) == FALSE )
    {
        /* this address is not in the table, so try to add it */
        if (numEntries < BRIDGE_TAB_SIZE)
        {
            b = NEW(BridgeEntry);
            b->binding = mapBind( bridgeMap, &src, b);
            /* use source address of packet as dest. address in table */
            b->destination = src;
            numEntries++;
        }
        else
        {
            /* can't fit this address in the table now, so give up */
            return;
        }
    }
}

```

```

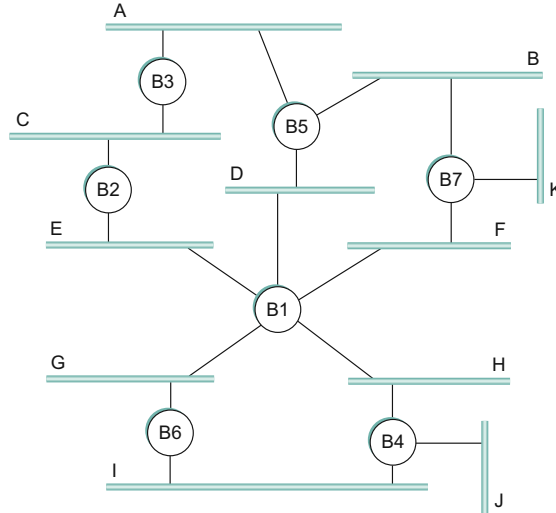
/* reset TTL and use most recent input interface */
b->TTL = MAX_TTL;
b->ifnumber = inif;
}

```

Note that this implementation adopts a simple strategy in the case where the bridge table has become full to capacity—it simply fails to add the new address. Recall that completeness of the bridge table is not necessary for correct forwarding; it just optimizes performance. If there is some entry in the table that is not currently being used, it will eventually time out and be removed, creating space for a new entry. An alternative approach would be to invoke some sort of cache replacement algorithm on finding the table full; for example, we might locate and remove the entry with the smallest TTL to accommodate the new entry.

Spanning Tree Algorithm

The preceding strategy works just fine until the extended LAN has a loop in it, in which case it fails in a horrible way—frames potentially loop through the extended LAN forever. This is easy to see in the example depicted in Figure 3.10, where, for example, bridges B1, B4, and B6 form

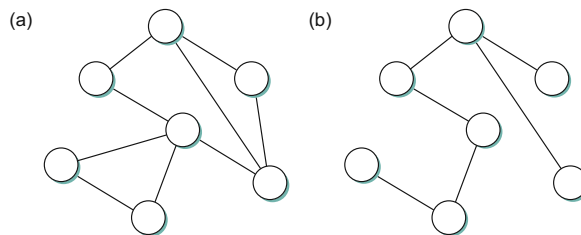


■ FIGURE 3.10 Extended LAN with loops.

a loop. Suppose that a packet enters bridge B4 from Ethernet J and that the destination address is one not yet in any bridge's forwarding table: B4 sends a copy of the packet out to Ethernets H and I. Now bridge B6 forwards the packet to Ethernet G, where B1 would see it and forward it back to Ethernet H; B4 still doesn't have this destination in its table, so it forwards the packet back to Ethernets I and J. There is nothing to stop this cycle from repeating endlessly, with packets looping in both directions among B1, B4, and B6.

Why would an extended LAN come to have a loop in it? One possibility is that the network is managed by more than one administrator, for example, because it spans multiple departments in an organization. In such a setting, it is possible that no single person knows the entire configuration of the network, meaning that a bridge that closes a loop might be added without anyone knowing. A second, more likely scenario is that loops are built into the network on purpose—to provide redundancy in case of failure. After all, a network with no loops needs only one link failure to become split into two separate partitions.

Whatever the cause, bridges must be able to correctly handle loops. This problem is addressed by having the bridges run a distributed *spanning tree* algorithm. If you think of the extended LAN as being represented by a graph that possibly has loops (cycles), then a spanning tree is a subgraph of this graph that covers (spans) all the vertices but contains no cycles. That is, a spanning tree keeps all of the vertices of the original graph but throws out some of the edges. For example, [Figure 3.11](#) shows a cyclic graph on the left and one of possibly many spanning trees on the right.



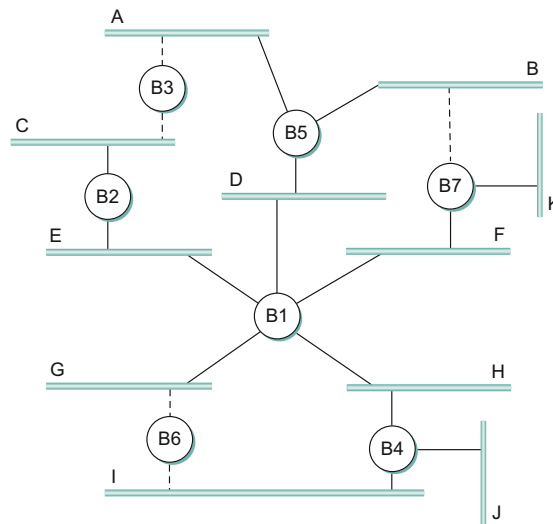
■ **FIGURE 3.11** Example of (a) a cyclic graph; (b) a corresponding spanning tree.

The idea of a spanning tree is simple enough: It's a subset of the actual network topology that has no loops and that reaches all the LANs in the extended LAN. The hard part is how all of the bridges coordinate their decisions to arrive at a single view of the spanning tree. After all, one topology is typically able to be covered by multiple spanning trees. The answer lies in the spanning tree protocol, which we'll describe now.

The spanning tree algorithm, which was developed by Radia Perlman at the Digital Equipment Corporation, is a protocol used by a set of bridges to agree upon a spanning tree for a particular extended LAN. (The IEEE 802.1 specification for LAN bridges is based on this algorithm.) In practice, this means that each bridge decides the ports over which it is and is not willing to forward frames. In a sense, it is by removing ports from the topology that the extended LAN is reduced to an acyclic tree.⁴ It is even possible that an entire bridge will not participate in forwarding frames, which seems kind of strange at first glance. The algorithm is dynamic, however, meaning that the bridges are always prepared to reconfigure themselves into a new spanning tree should some bridge fail, and so those unused ports and bridges provide the redundant capacity needed to recover from failures.

The main idea of the spanning tree is for the bridges to select the ports over which they will forward frames. The algorithm selects ports as follows. Each bridge has a unique identifier; for our purposes, we use the labels B1, B2, B3, and so on. The algorithm first elects the bridge with the smallest ID as the root of the spanning tree; exactly how this election takes place is described below. The root bridge always forwards frames out over all of its ports. Next, each bridge computes the shortest path to the root and notes which of its ports is on this path. This port is also selected as the bridge's preferred path to the root. Finally, all the bridges connected to a given LAN elect a single *designated* bridge that will be responsible for forwarding frames toward the root bridge. Each LAN's designated bridge is the one that is closest to the root. If two or more bridges are equally close to the root, then the bridges' identifiers are used to break ties, and

⁴Representing an extended LAN as an abstract graph is a bit awkward. Basically, you let both the bridges and the LANs correspond to the vertices of the graph and the ports correspond to the graph's edges. However, the spanning tree we are going to compute for this graph needs to span only those nodes that correspond to networks. It is possible that nodes corresponding to bridges will be disconnected from the rest of the graph. This corresponds to a situation in which all the ports connecting a bridge to various networks get removed by the algorithm.



■ **FIGURE 3.12** Spanning tree with some ports not selected.

the smallest ID wins. Of course, each bridge is connected to more than one LAN, so it participates in the election of a designated bridge for each LAN it is connected to. In effect, this means that each bridge decides if it is the designated bridge relative to each of its ports. The bridge forwards frames over those ports for which it is the designated bridge.

Figure 3.12 shows the spanning tree that corresponds to the extended LAN shown in Figure 3.10. In this example, B1 is the root bridge, since it has the smallest ID. Notice that both B3 and B5 are connected to LAN A, but B5 is the designated bridge since it is closer to the root. Similarly, both B5 and B7 are connected to LAN B, but in this case B5 is the designated bridge since it has the smaller ID; both are an equal distance from B1.

While it is possible for a human to look at the extended LAN given in Figure 3.10 and to compute the spanning tree given in Figure 3.12 according to the rules given above, the bridges in an extended LAN do not have the luxury of being able to see the topology of the entire network, let alone peek inside other bridges to see their ID. Instead, the bridges have to exchange configuration messages with each other and then decide

whether or not they are the root or a designated bridge based on these messages.

Specifically, the configuration messages contain three pieces of information:

1. The ID for the bridge that is sending the message
2. The ID for what the sending bridge believes to be the root bridge
3. The distance, measured in hops, from the sending bridge to the root bridge

Each bridge records the current *best* configuration message it has seen on each of its ports (“best” is defined below), including both messages it has received from other bridges and messages that it has itself transmitted.

Initially, each bridge thinks it is the root, and so it sends a configuration message out on each of its ports identifying itself as the root and giving a distance to the root of 0. Upon receiving a configuration message over a particular port, the bridge checks to see if that new message is better than the current best configuration message recorded for that port. The new configuration message is considered *better* than the currently recorded information if any of the following is true:

- It identifies a root with a smaller ID.
- It identifies a root with an equal ID but with a shorter distance.
- The root ID and distance are equal, but the sending bridge has a smaller ID

If the new message is better than the currently recorded information, the bridge discards the old information and saves the new information. However, it first adds 1 to the distance-to-root field since the bridge is one hop farther away from the root than the bridge that sent the message.

When a bridge receives a configuration message indicating that it is not the root bridge—that is, a message from a bridge with a smaller ID—the bridge stops generating configuration messages on its own and instead only forwards configuration messages from other bridges, after first adding 1 to the distance field. Likewise, when a bridge receives a configuration message that indicates it is not the designated bridge for that port—that is, a message from a bridge that is closer to the root or equally far from the root but with a smaller ID—the bridge stops sending

configuration messages over that port. Thus, when the system stabilizes, only the root bridge is still generating configuration messages, and the other bridges are forwarding these messages only over ports for which they are the designated bridge. At this point, a spanning tree has been built, and all the bridges are in agreement on which ports are in use for the spanning tree. Only those ports may be used for forwarding data packets in the extended LAN.

Let's see how this works with an example. Consider what would happen in Figure 3.12 if the power had just been restored to the building housing this network, so that all the bridges boot at about the same time. All the bridges would start off by claiming to be the root. We denote a configuration message from node X in which it claims to be distance d from root node Y as (Y, d, X) . Focusing on the activity at node B3, a sequence of events would unfold as follows:

1. B3 receives $(B2, 0, B2)$.
2. Since $2 < 3$, B3 accepts B2 as root.
3. B3 adds one to the distance advertised by B2 (0) and thus sends $(B2, 1, B3)$ toward B5.
4. Meanwhile, B2 accepts B1 as root because it has the lower ID, and it sends $(B1, 1, B2)$ toward B3.
5. B5 accepts B1 as root and sends $(B1, 1, B5)$ toward B3.
6. B3 accepts B1 as root, and it notes that both B2 and B5 are closer to the root than it is; thus, B3 stops forwarding messages on both its interfaces.

This leaves B3 with both ports not selected, as shown in Figure 3.12.

Even after the system has stabilized, the root bridge continues to send configuration messages periodically, and the other bridges continue to forward these messages as described in the previous paragraph. Should a particular bridge fail, the downstream bridges will not receive these configuration messages, and after waiting a specified period of time they will once again claim to be the root, and the algorithm just described will kick in again to elect a new root and new designated bridges.

One important thing to notice is that although the algorithm is able to reconfigure the spanning tree whenever a bridge fails, it is not able to forward frames over alternative paths for the sake of routing around a congested bridge.

Broadcast and Multicast

The preceding discussion has focused on how bridges forward unicast frames from one LAN to another. Since the goal of a bridge is to transparently extend a LAN across multiple networks, and since most LANs support both broadcast and multicast, then bridges must also support these two features. Broadcast is simple—each bridge forwards a frame with a destination broadcast address out on each active (selected) port other than the one on which the frame was received.

Multicast can be implemented in exactly the same way, with each host deciding for itself whether or not to accept the message. This is exactly what is done in practice. Notice, however, that since not all the LANs in an extended LAN necessarily have a host that is a member of a particular multicast group, it is possible to do better. Specifically, the spanning tree algorithm can be extended to prune networks over which multicast frames need not be forwarded. Consider a frame sent to group M by a host on LAN A in Figure 3.12. If there is no host on LAN J that belongs to group M, then there is no need for bridge B4 to forward the frames over that network. On the other hand, not having a host on LAN H that belongs to group M does not necessarily mean that bridge B1 can avoid forwarding multicast frames onto LAN H. It all depends on whether or not there are members of group M on LANs I and J.

How does a given bridge learn whether it should forward a multicast frame over a given port? It learns exactly the same way that a bridge learns whether it should forward a unicast frame over a particular port—by observing the *source* addresses that it receives over that port. Of course, groups are not typically the source of frames, so we have to cheat a little. In particular, each host that is a member of group M must periodically send a frame with the address for group M in the source field of the frame header. This frame would have as its destination address the multicast address for the bridges.

Note that, although the multicast extension just described has been proposed, it is not widely adopted. Instead, multicast is implemented in exactly the same way as broadcast on today's extended LANs.

Limitations of Bridges

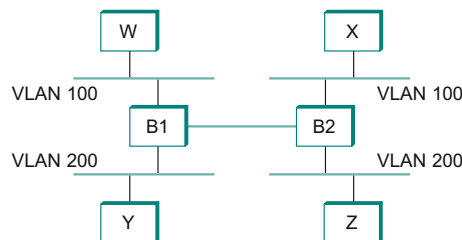
The bridge-based solution just described is meant to be used in only a fairly limited setting—to connect a handful of similar LANs. The main

limitations of bridges become apparent when we consider the issues of scale and heterogeneity.

On the issue of scale, it is not realistic to connect more than a few LANs by means of bridges, where in practice *few* typically means “tens of.” One reason for this is that the spanning tree algorithm scales linearly; that is, there is no provision for imposing a hierarchy on the extended LAN. A second reason is that bridges forward all broadcast frames. While it is reasonable for all hosts within a limited setting (say, a department) to see each other’s broadcast messages, it is unlikely that all the hosts in a larger environment (say, a large company or university) would want to have to be bothered by each other’s broadcast messages. Said another way, broadcast does not scale, and as a consequence extended LANs do not scale.

One approach to increasing the scalability of extended LANs is the *virtual LAN* (VLAN). VLANs allow a single extended LAN to be partitioned into several seemingly separate LANs. Each virtual LAN is assigned an identifier (sometimes called a *color*), and packets can only travel from one segment to another if both segments have the same identifier. This has the effect of limiting the number of segments in an extended LAN that will receive any given broadcast packet.

We can see how VLANs work with an example. Figure 3.13 shows four hosts on four different LAN segments. In the absence of VLANs, any broadcast packet from any host will reach all the other hosts. Now let’s suppose that we define the segments connected to hosts W and X as being in one VLAN, which we’ll call VLAN 100. We also define the segments that connect to hosts Y and Z as being in VLAN 200. To do this, we need to



■ FIGURE 3.13 Two virtual LANs share a common backbone.

configure a VLAN ID on each port of bridges B1 and B2. The link between B1 and B2 is considered to be in both VLANs.

When a packet sent by host X arrives at bridge B2, the bridge observes that it came in a port that was configured as being in VLAN 100. It inserts a VLAN header between the Ethernet header and its payload. The interesting part of the VLAN header is the VLAN ID; in this case, that ID is set to 100. The bridge now applies its normal rules for forwarding to the packet, with the extra restriction that the packet may not be sent out an interface that is not part of VLAN 100. Thus, under no circumstances will the packet—even a broadcast packet—be sent out the interface to host Z, which is in VLAN 200. The packet is, however, forwarded on to bridge B1, which follows the same rules and thus may forward the packet to host W but not to host Y.

An attractive feature of VLANs is that it is possible to change the logical topology without moving any wires or changing any addresses. For example, if we wanted to make the segment that connects to host Z be part of VLAN 100 and thus enable X, W, and Z to be on the same virtual LAN, then we would just need to change one piece of configuration on bridge B2.

On the issue of heterogeneity, bridges are fairly limited in the kinds of networks they can interconnect. In particular, bridges make use of the network's frame header and so can support only networks that have exactly the same format for addresses. Thus, bridges can be used to connect Ethernets to Ethernets, token rings to token rings, and one 802.11 network to another. It's also possible to put a bridge between, say, an Ethernet and an 802.11 network, since both networks support the same 48-bit address format. However, bridges do not readily generalize to other kinds of networks with different addressing formats, such as ATM.⁵

Despite their limitations, bridges are a very important part of the complete networking picture. Their main advantage is that they allow multiple LANs to be transparently connected; that is, the networks can be connected without the end hosts having to run any additional protocols (or even be aware, for that matter). The one potential exception is when the hosts are expected to announce their membership in a multicast group, as described in [Section 3.1.4](#).

⁵Ultimately there was quite a lot of work done to make ATM networks behave more like Ethernets, called *LAN Emulation*, to get around this limitation, but this is rarely seen today.

Notice, however, that this transparency can be dangerous. If a host or, more precisely, the application and transport protocol running on that host is programmed under the assumption that it is running on a single LAN, then inserting bridges between the source and destination hosts can have unexpected consequences. For example, if a bridge becomes congested, it may have to drop frames; in contrast, it is rare that a single Ethernet ever drops a frame. As another example, the latency between any pair of hosts on an extended LAN becomes both larger and more highly variable; in contrast, the physical limitations of a single Ethernet make the latency both small and predictable. As a final example, it is possible (although unlikely) that frames will be reordered in an extended LAN; in contrast, frame order is never shuffled on a single Ethernet. The bottom line is that it is never safe to design network software under the assumption that it will run over a single Ethernet segment. Bridges happen.



3.2 BASIC INTERNETWORKING (IP)

In the previous section, we saw that it was possible to build reasonably large LANs using bridges and LAN switches, but that such approaches were limited in their ability to scale and to handle heterogeneity. In this section, we explore some ways to go beyond the limitations of bridged networks, enabling us to build large, highly heterogeneous networks with reasonably efficient routing. We refer to such networks as *internetworks*. We'll continue the discussion of how to build a truly global internetwork in the next chapter, but for now we'll explore the basics. We start by considering more carefully what the word *internetwork* means.

3.2.1 What Is an Internetwork?

We use the term *internetwork*, or sometimes just *internet* with a lowercase *i*, to refer to an arbitrary collection of networks interconnected to provide some sort of host-to-host packet delivery service. For example, a corporation with many sites might construct a private internetwork by interconnecting the LANs at their different sites with point-to-point links leased from the phone company. When we are talking about the widely used global internetwork to which a large percentage of networks are now connected, we call it the *Internet* with a capital *I*. In keeping with the first-principles approach of this book, we mainly want you to learn about the