INTERNET BROADCASTING

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

By

David Reeves Boggs

January 1982

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Forest Baskett III, principal advisor

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Brian K. Reid, associate advisor

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Fabian Pease, Electrical Engineering

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies & Research

Abstract

Broadcasting should be a standard addressing mode of all packet-switched computer networks. Further, when networks are interconnected to form an internet, a broadcast mechanism is also required.

Broadcasting is the delivery of a packet to all hosts in a network; unicasting is the delivery of a packet to one specific host. They are distinct forms of interprocess communication; functions that are simple to do with one are difficult to do using only the other.

A broadcast is used when you don't know whom specifically to address. There are two situations where this occurs: 1) when you are searching for some information but you don't know who to ask (for example, standing up in a theatre and saying "Is there a doctor in the house?"), and 2) when you posess some information of use to others but you don't know specifically who (for example, standing up in a theatre and yelling "Fire!").

A network should give its best efforts to deliver a copy of a broadcast packet to each host, but perfectly reliable delivery is not required. It is sufficient that most hosts receive a broadcast and that the same hosts not miss retransmissions. Just as with unicasting, higher-level protocols can be used to improve the reliability of the basic broadcast delivery mechanism, if required. Such an "unreliable" broadcast mechanism is straightforward to implement in all types of packet-switched networks.

In an internet composed of possibly thousands of networks and millions of hosts, a full internet-wide broadcast, the obvious internet analog to broadcasting in a single network, is seldom the right choice. A *directed broadcast*, delivery of a packet to all hosts on any single network in an internet, is simpler to implement, closer to what most users need, and sufficient to construct many forms of broadcast-based interprocess communication, including an internet-wide broadcast.

Table of Contents

List of Figures

Chapter 1

Introduction

> **Broadcast** (bro-dkast), *a., adv., sb.* [f. BROAD *adv.* + CAST *pa. pple.*] A. *adj.*
>
> 1. Of seed, etc.: Scattered abroad over the whole surface, instead of being sown in drills or rows.
>
> 2. *fig.* a. Scattered widely abroad, widely disseminated.
>
> Concise Oxford English Dictionary, 1971

1.1. What this thesis is about

This thesis argues that broadcasting should be available as a standard addressing mode of all packet-switched computer networks. Further, when networks are interconnected to form an internet, a broadcast mechanism is also required.

To illustrate that broadcasting is valuable, I will describe several broadcast-based applications implemented in the Xerox Pup internet [Boggs *et. al.*, 1980]. To demonstate that broadcasting is feasible, I will survey current networks, and show how to provide a broadcast facility in each type. Finally, I will describe *directed broadcasting*, the internet broadcast mechanism provided by Pup.

1.2. Some definitions

1.2.1. Packet switching and protocols

Communication switching divides into two fundamental models: *circuit switching* and *packet switching*. The most familiar example of circuit switching is a telephone system. A series of circuits (wires, radio channels, or whatever) are concatenated at the time the call is set up and they remain dedicated until the call is terminated.

The parties in a telephone conversation follow a set of conventions or *protocols* for conducting the call in an orderly manner. The called party usually says "Hello?" to indicate that he is listening, and then the calling party identifies himself "Hello, this is David Boggs calling." Each party speaks in turn, using cues such as voice inflection or certain phrases to avoid speaking at the same time. If a party misses something (*e.g.* because of a momentary

failure of the circuit or because of a loud noise in his room), he asks the other end to repeat. The call is cleanly terminated when both parties say "goodbye" and hang up.

The best example of packet switching is a postal system. Correspondents carry on a dialog by exchanging letters. Agents of the postal system examine the address on a letter at various points to decide in which mail pouch, truck or airplane it should next be carried to get it closer to its destination. No resources are exclusively dedicated to the letter throughout its trip.

Correspondents also use protocols to control the conversation and recover from failures of the communication channel. A letter carries an address to which it can be returned if it can not be delivered. Correspondents usually date their letters so that out-of-order delivery can be detected. Letters are usually exchanged alternately, and a careful sender will keep a copy of each letter he writes. If he receives no answer within a reasonable time, or if the other party sends a letter inquiring about an overdue letter, another copy can be sent.

This thesis will focus on packet-switched computer networks and their protocols.

1.2.2. Broadcasting, multicasting, and unicasting

Continuing our analogy, the postal system illustrates a spectrum of addressing modes:

Broadcasting. A letter can be *broadcast, i.e.* addressed to all recipients in an area—for example, a Congressman's periodic reports to his constituents, which are addressed to "postal patron, 24th Congressional district." These letters (packets) do not contain a fully specified address but rather an area address, and the postal system undertakes to deliver one copy to each valid address in the area. Delivery is more complicated when the area served by a post office does not follow the boundary of a Congressional district.

Multicasting. A letter can be *multicast, i.e.* addressed to some but not all recipients in an area—for example, an amateur radio equipment dealer mailing its catalog to all radio amateurs in California. The postal system will not determine the postal patrons in California who are radio amateurs, so the dealer must get an address list from the Federal Communications Commision and individually address each catalog. The dealer can get a lower rate by presorting the catalogs into bundles by zip code (usually equivalent to a local post office).

Unicasting. A letter can be *unicast, i.e.* addressed to one recipient. If my mail is any indication, most letters carried by the postal system are either broadcast or multicast, with unicast letters being a minority.

Broadcasting and unicasting are special cases of multicasting. Multicasting addresses a subset of all possible addresses. For a broadcast the subset is all elements; for a unicast the subset is one element. Broadcasting and multicasting are not common in circuit-switched

systems. A conference call is a form of multicasting; telephone networks do not support broadcasting.

This thesis will focus on broadcast packets: why they are useful, how to implement them, and protocols for using them.

1.2.3. Networks and internets

It is difficult to define a *network* without going up one level and describing an *internet.* In our postal system example, a national postal system is a network, and the planetary postal system is a network of networks, an *internet.* A letter with an address in English will be delivered anywhere in the United States by the U.S. postal system. To send a letter from the U.S. to Germany, the address should be in German, but with "Federal Republic of Germany" prominently in English.

The client of a network sees changes when he must cross a boundary into another network. In our example, the language of the address changes, and a new field, the "destination country," is added. There are more subtle differences too—for example, an envelope with two transparent windows is not acceptable to the international mail system, but is acceptable to the U.S. mail system.

An internet attempts to smooth out the discontinuities at the boundaries of component networks by defining a set of capabilities that all component networks can support. This makes an internet into just a bigger network; there is no clearcut distinction [Shoch *et al.*, 1980]. Internets tend to be large, and sheer size is often an important factor. The Pup internet, described in the next chapter, can have 256 networks with 256 hosts per network; this should be viewed as a small system.

This thesis will focus on internets.

1.2.4. Disclaimers

The analogy between a postal system and computer packet switching breaks down in several important ways. Users of a postal system must provide it with all of the copies that it delivers. Users of a computer network need not, because computers are very good at copying.

A letter can contain a large number of words, but packets are quite short. A written communication seldom gets so large that it must be segmented and mailed in separate letters, but this is often the case with computer conversations using packets. This introduces a layer of complication that seldom arises in postal systems.

1.3. Related work

1.3.1. Reverse-path forwarding

Yogen Dalal investigated broadcasting, concentrating on store-and-forward networks [1977]. He considered and analyzed six ways to implement a broadcast capability: separately-addressed packets, multicast addressing, hot-potato forwarding, source-based forwarding, reverse-path forwarding, and forwarding along a spanning tree. Of the six techniques, reverse-path forwarding is by far the best. He also did a paper design of a distributed file system to illustrate places where a broadcast capability would be valuable.

1.3.2 Center-based forwarding

David Wall investigated mechanisms for broadcast and multicast in store-and-forward networks [1980]. He extended Dalal's broadcast mechanisms to the more general problem of multicasting. He also investigated broadcasting and multicasting along a new class of "centered" trees. By picking a node that is in some sense "in the center" of a network, he can control the delay of a broadcast rather than its cost.

1.3.3. The Ethernet local network

Since every packet transmitted on an Ethernet local network passes by every host interface, it is intrinsically a broadcast network [Metcalfe & Boggs, 1976]. The address in each packet is checked by the attached computer against a preset address. If it matches, the packet is copied into memory. This is unicasting. In addition, a packet with a special destination address is recognized by all listening computers and copied into memory. This is broadcasting. The computer can also instruct the interface to copy all packets into memory, where arbitrarily complex forms of address filtering can be done. This permits multicasting.

When Ethernets were interconnected to form the beginnings of the Pup internet [Boggs *et al.*, 1980], one of the first applications for broadcasts was the routing algorithm. As other types of networks were added to the internet, ways were found to implement broadcasts in them too, so that the simple broadcast-based routing algorithm could be used. Soon, application programs using the internet encountered problems similar to those which the routing system solved so cleanly and simply by broadcasting, and so the broadcast facilites were generalized and made available to internet clients.

1.3.4. Distributed Computing System at U.C. Irvine

The Distributed Computing System ring network [Farber *et al.*, 1973], developed by David Farber at the University of California at Irvine contains a very flexible addressing scheme. Each network switch (in this case a ring repeater and its associated computer interface), contains an associative table in which the destination address of each packet is looked up. If the address matches, a copy is made for the attached computer. Broadcasting is achieved by entering some address (the broadcast address) in all tables. Multicasting is accomplished by entering an address in a subset of the tables. Larry Rowe described a distributed operating system implemented on top of this hardware [1975]. Paul Mockapetris has proposed extensions to the associative table which would permit matching on subfields of an address [1978].

An improved version of the hardware, called the *Arpa Local Net Interface*, is currently being developed [Clark *et al.*, 1978]. The first version of this new design, which is operating at several sites including UCLA, MIT and Berkeley, retains the name table. The second version eliminates the name table in favor of the address filtering scheme used by the Ethernet.

1.3.5. Emerging uses of broadcasts

The development of the Arpa internet [DoD, 1980] has paralleled that of the Pup internet in many ways, but without a broadcast capability. An Arpa Internet service for converting host names into addresses (much like the Pup name server described in chapter 2) has been proposed by Pickens *et al.* [1979], but the design glosses over the means by which a host finds the name server itself—a perfect application for a broadcast.

Bill Plummer has considered ways to implement a *news wire* service in the Arpa internet [1977]. He remarked that a fundamental assumption of the internet design is that no "special requirements" may be placed on the component networks. Some networks, such as the Arpa Packet Satellite network, support broadcasting, and it would be nice to take some advantage of this capability. But other component networks, such as the Arpanet, do not have a broadcast capability, and so it can not be used.

The Arpanet routing algorithm was recently changed to use a broadcast to disseminate routing information [Rosen, 1979]. A specialized reliable broadcast facility is implemented by the routing processes in Imps. Unfortunately this mechanism is not general purpose and can not be used by regular Arpanet hosts.

Some distributed system designers have not waited for network designers to agree that broadcasts are an essential feature, but rather have plunged ahead assuming them. Peacock *et al.* investigated the use of broadcasts for synchronizing multiple processors cooperating in a simulation [1979].

1.4. Synopsis

The thesis is that broadcasting should be a standard addressing mode of all packet-switched computer networks and internets. The argument is divided into four parts: description of an example internet, case studies of the value of broadcasting, ways to broadcast in various networks, and ways to broadcast in an internet.

Chapter 2: The Pup internet. This is a description of a real, functioning internet. From time to time, topics in the following chapters are exemplified by considering them in the context of this system.

The chapter begins with the major issues which must be confronted when designing an internet. The second section is a high-level description of the architecture. The third section sketches an implementation for an ideal machine. The last section describes a particularly relevant broadcast-based application: internet routing.

Chapter 3: Why broadcast? This chapter demonstrates the utility of broadcasting by presenting more broadcast-based applications that are in widespread use in the Pup internet.

The protocol for converting between names and addresses illustrates the need for and use of a database with multiple copies. The protocol for bootstrap loading machines through a network shows the power of broadcasting when simplicity is essential. The protocol for obtaining the date and time is even simpler. The protocols used by machine diagnostics illustrate some specialized addressing modes similar to multicasting. The chapter concludes with some general comments on broadcast-based applications.

Chapter 4: How to broadcast. The purpose of this chapter is to show that it is feasible to implement broadcasts in all types of networks. Few existing networks (including some intrinsically broadcast ones) implement any form of broadcast; of these few even fewer make the capability available to clients.

The chapter opens with some general comments on broadcasting applicable to all types of networks. The next section examines the addressing modes available in a number of current networks. Broadcasting in a store-and-forward network seems hard to do; the last section shows that a suitably-defined broadcast facility in such a network is not hard to provide.

Chapter 5: Directed broadcasting. Having demonstated *why* broadcasting is useful, and *how* to broadcast in single networks, this chapter examines the details of broadcasting in an internet.

After discussing some design issues peculiar to broadcasting in an internet, broadcast mechanisms suitable for embedding in an internet are discussed. A simple *directed broadcast* has proven to be sufficient in the Pup internet. The last section discusses techniques to search for and distribute information using directed broadcasts.

The research reported in this dissertation is that of the author. Previous work is referenced where appropriate and where collaborative work has been involved, specific acknowledgement or reference is made to the other team members.

Chapter 2

The Pup internet

> But let your communication be, Yea, yea; Nay, nay: for whatsoever is more than these cometh of evil.
>
> Matthew 5:37

This chapter describes Pup [Boggs *et al.*, 1980], a working internet that is used as a concrete example throughout the rest of the thesis.

The name *Pup* originally referred to the abstract design of a standard internetwork packet (the Parc[1] Universal Packet), but has expanded in usage to include a whole hierarchy of internetwork protocols as well as a general style for internetwork communication.

The computational environment in which Pup evolved includes a large number of "Alto" minicomputers [Thacker *et al.*, 1980] and other personal computers capable of high-quality interaction with human users. Supporting these are various specialized server systems that are shared among many clients and provide access to expensive peripherals such as large disks, magnetic tapes, and laser printers. Additionally, there are several general-purpose time sharing systems providing conventional services for terminal users (see figure 2.1).

The communication environment includes several different network designs. The dominant design is the experimental Ethernet network, a local-area broadcast network with a bandwidth of 3 megabits per second [Metcalfe & Boggs, 1976]. Longer-haul communication facilities include the Arpanet [McQuillan & Walden, 1977], the Arpa Packet Radio Net [Kahn, 1977], and a collection of leased lines implementing a store-and-forward network. These various facilities have distinct native protocols and exhibit as much as three orders of magnitude difference in performance.

The applications supported include a wide range of activites: electronic mail, file transfer, access to time sharing services, access to specialized data bases, document transmission, software maintenance, and packet telephony, to name just a few.

Section 2.1 discusses some of the architectural issues which form the foundation of Pup. Section 2.2 describes the Pup architecture in moderate detail. Section 2.3 sketches an implementation of the lower levels for an ideal machine. Section 2.4 describes Pup's broadcast-based routing algorithm.

---

1. *Parc* is an acronym for the Xerox Palo Alto Research Center.

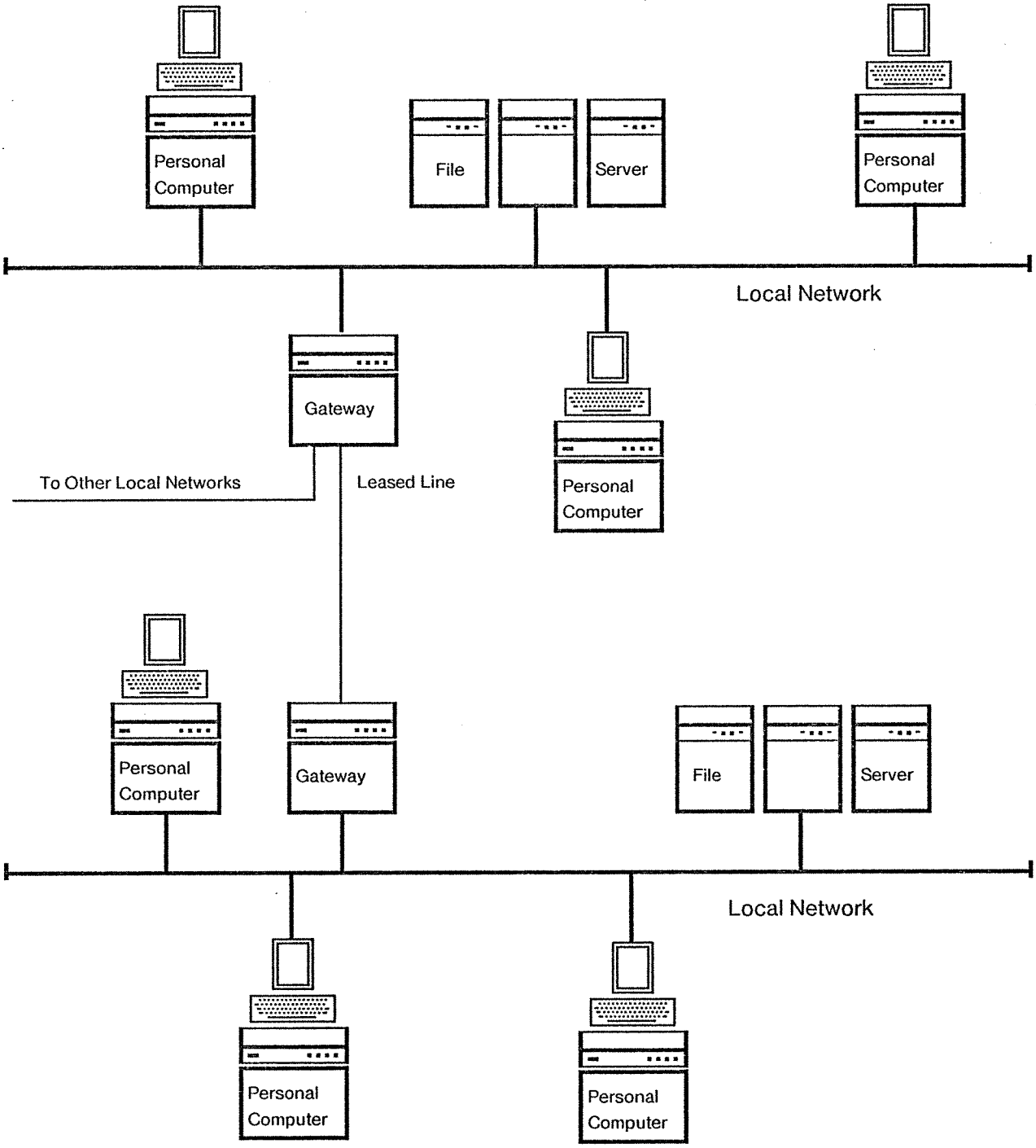

Figure 2.1. A typical Pup internet.

2.1. Architectural issues

This section enumerates some of the major design issues confronted in the development of a network architecture and describes, in general terms, the choices made in the development of Pup. From this discussion the broad outlines of Pup will emerge; later sections provide details.

2.1.1. The internet

The internet is the collection of host software, networks, and gateways that transports packets (also called *datagrams*). At a minimum, it should provide a way for any host to send a packet to any other host.

The basic function provided by the Pup internet is the transport of packets. The internet does not guarantee reliable delivery of packets (also called "Pups"); it simply gives its "best efforts" to deliver each one and allows the end processes to build protocols that provide reliable communication of the quality they themselves desire [Metcalfe, 1973]. The internet has no notion of a *connection.* It transports each Pup independently, and leaves construction of a connection—if that is the appropriate interprocess communication model—to the end processes. Keeping fragile end-to-end state information out of the packet transport system contributes to its reliability and simplicity.

Individual networks give their best efforts to deliver internet packets, but they do not guarantee reliable delivery. Packets may be lost, duplicated, delivered out of order, after a great delay, or with concealed damage. A network can have any combination of bandwidth, delay, error characteristics, topology, and economics; the routing algorithm should attempt to take these characteristics into consideration.

A *gateway* is a host connected to two or more networks and running a program that forwards packets among them. In order for hosts (including other gateways) to know of its existence, a gateway must also give out routing information.

2.1.2. A layered hierarchy of protocols

Layering of protocols is an effective means for structuring a network design: each level uses the functions of a lower level, and adds some functionality of its own for possible use by higher levels. Pup protocols are organized into a hierarchy, as shown in figure 2.2. A level represents an abstraction that can be realized in different ways in different hosts.

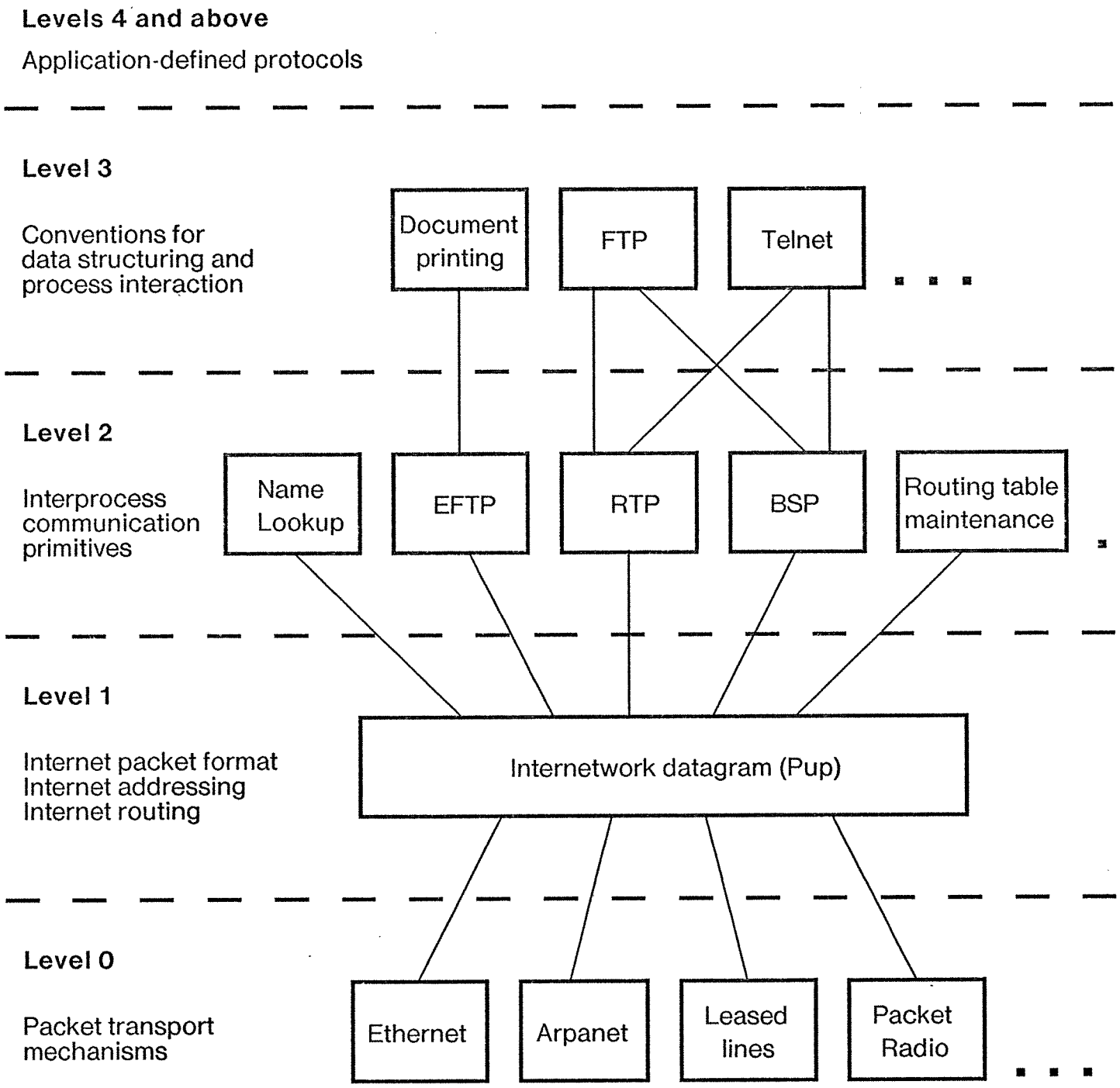

Figure 2.2. The Pup protocol hierarchy.

The level-0 abstraction is a packet transport mechanism. There are many realizations of packet transport, including an Ethernet, the Arpanet, the Arpa packet radio network, and leased phone lines. Level-0 protocols include specifications such as hardware interfaces, electrical and timing characteristics, bit encodings, line control procedures, and network-dependent packet formatting conventions.

The level-1 abstraction is an internet packet. The realization of this abstraction consists of the format of a Pup, a hierarchical addressing scheme, and an internetwork routing algorithm. There is only one box at level 1: the internet packet protocol. It is this layer of commonality that unifies all of the different networks that might be used at level 0, and which makes available a uniform interface to all of the layers above. The purpose of this level is to provide medium independence while maintaining the common properties of the underlying packet networks. My thesis is that broadcasting should be one of the common properties of all networks and should be available to clients of an internet.

The level-2 abstraction is an interprocess communication mechanism: a way to move bits without saying much about their form or content. Various level-2 protocols provide many combinations of reliability, throughput, delay and complexity. These protocols can be divided into two classes according to the amount and lifetime of state kept by the communicating end processes. Connectionless protocols support short-lived interactions; the end processes maintain little state, and usually only during the exchange of a few Pups—no more than a few seconds. Connection-based protocols support sustained interactions, generally requiring substantial state to be maintained at both ends, and for longer periods—minutes to hours.

Level 3 adds structure to the data moved at level 2, as well as conventions for how processes interact. For example, the File Transfer Protocol (FTP) consists of a set of conventions for talking about files and a format for sending them through a level-2 BSP connection. Level-3 protocols are sometimes referred to as function-oriented protocols [Crocker *et al.*, 1972].

Above level 3 the dividing lines become blurred, and individual applications evolve with their own natural decomposition into additional layers. This thesis is primarily concerned with levels 0, 1, and connectionless level-2 protocols. With respect to layering of protocols, Pup is similar in many ways to the Arpa internet and TCP design [Postel, Sunshine & Cohen, 1981; Cerf & Kahn, 1974] and the Open Systems Architecture [Zimmermann, 1979; desJardins, 1981].

2.1.3. Reliability

Defining packets to be less than perfectly reliable is realistic since it reflects the characteristics of many existing packet transport mechanisms. Probabilistic transmission is basic to the theory of operation of certain network designs such as Ethernet. Even in networks nominally designed to deliver correctly sequenced, error-free packets, occasional anomalies can

result from certain hardware or software failures: an Arpanet Imp may crash while holding the only copy of a packet, or an X.25 virtual circuit may be reset.

The Pup internet *always* has the option of discarding packets, though the system should be engineered so that this is an infrequent event. This point is of considerable practical importance when one considers the complicated measures required to avoid deadlocks in the Arpanet—conditions that are a direct consequence of attempting to provide reliable delivery of every packet in a store-and-forward network [Metcalfe, 1973]. Packet management strategies that attempt to guarantee perfect reliability must be designed to operate correctly under *worst-case* conditions, whereas strategies that have the option of discarding packets when necessary need operate correctly only under *most* conditions. The idea is to sacrifice the guarantee of reliable delivery of individual packets and to capitalize on the resulting simplicity to produce higher overall reliability and performance.

For some applications, perfectly reliable transport is unnecessary and possibly even undesirable, especially if it is obtained at the cost of increased delay. For example, in real-time speech applications, loss of an occasional packet is of little consequence, but even short delays (or worse, highly variable ones) can cause significant degradation [Cohen, 1977, Sproull & Cohen, 1978].

On the other hand, many applications such as file transfer and terminal connection *do* depend upon fully reliable transmission. In these cases, it is perfectly reasonable to build a reliable end-to-end protocol on top of the internet packets. Ultimately, reliability (by some definition) is always required; the issue is the protocol level at which it should be provided. The design of Pup is based on the principle that it is the responsibility of the end processes to define and implement whatever form of reliable transport is appropriate to the situation.

Reliable delivery requires the maintenance of state information at the source and destination. The actions of a large class of simple servers, such as giving out routing tables or converting names into addresses, are idempotent (*i.e.* may be repeated without incremental effects), and a client of that service can simply retransmit a request if no response arrives. These protocols reduce to a simple exchange of Pups, with an occasional retransmission by the client, but with no state retained by the server.

2.1.4. Naming, addressing, and routing

Names, addresses, and routes are three important and distinct entities in an internet [Shoch, 1978]:

> The *name* of a resource is *what* one seeks,
> an *address* indicates *where* it is, and
> a *route* is *how to get there.*

A *name* is a symbol, such as a human-readable text string, identifying some resource (process, device, service, *etc.*). An *address* is a data structure whose format is understood by level 1 of the internet, which is used to specify the source and destination of a Pup. A *route* is the information needed to forward a Pup to its specified address. Each of these represents a tighter binding of information: names are mapped into addresses, and addresses are mapped into routes. Error recovery should successively fall back to find an alternate route, then an alternate address, then an alternate name.

The mapping from names to addresses is necessarily application-specific, since the syntax and semantics of names depend entirely on the types of entities that are being named and the use that is being made of them. This issue is dealt with at the appropriate higher levels of protocol. Section 3.1 describes a broadcast-based level-2 protocol for converting between host names and addresses.

An address field, as contained in a Pup, is one of the important elements of commonality in the internet design. An end process sends and receives Pups through a *port* identifed by a hierarchical address consisting of three parts: a *network number*, a *host number*, and a *socket number*. The internet is decoupled from hosts' process structures by the use of ports: one process may own several ports and one port may be shared by several processes.

The actual process of routing a packet through the Pup internet uses a distributed adaptive routing procedure [McQuillan, 1974]. The source process specifies only the destination address and not the path from source to destination. The internetwork gateways route Pups to the proper network, a network then routes Pups to the proper host, and a host routes Pups to the proper socket.

This routing process is associated with level 1 in the protocol hierarchy, the level at which packet formats and internet addresses are standardized. The software implementing level 1 is sometimes referred to a *router*, since it routes outgoing packets to networks and incoming packets to sockets. Thus, the routing table itself is kept at level 1; a very simple host (or gateway) would need only levels 0 and 1 in order to route Pups. But the routing table also requires periodic updating, as gateways exchange and distribute current routing information. This *routing table maintenance* protocol, described in section 2.4, is based on broadcast packets and is found logically at level 2 of the hierarchy.

Hosts are expected to know or be able to discover their addresses (host numbers) on directly-connected networks; gateways are the source of network numbers. Gateways provide internet routing tables to individual hosts as well as to each other. Hosts use this routing information to decide where to send outgoing packets destined other than to directly-connected networks.

2.2. The Pup architecture

The preceding section outlined some of the internetworking issues Pup addresses. This section is a bottom-up description of the Pup architecture diagrammed in figure 2.2.

2.2.1. Level 0: packet transport

An individual network moves network-specific packets among hosts; the addressing schemes, error characteristics, maximum packet sizes, and other attributes of networks vary greatly. An internetwork *packet transport mechanism*, on the other hand, moves Pups between hosts. The level-0 code which transforms a network into an internet packet transport mechanism is called a *network driver.*

A host has a driver for each directly-connected network. Of course, most hosts are connected to only one network and have only one driver, but hosts with multiple network connections (which includes gateways) have one driver for each network. Only the driver knows about the peculiarities of a network's hardware interface and low-level protocol (*e.g.*, collisions on an Ethernet, RFNMs in the Arpanet).

A network driver may also be asked to broadcast a packet to all other hosts on that network. If the bare network does not provide a broadcast capability then the driver must implement one. This is the principal topic of chapter 4.

Pup transport mechanisms do not have to be perfectly reliable, but they should successfully deliver a packet most of the time—a packet loss rate of less than .1 percent is acceptable. If a network's inherent error characteristics are unfavorable, the driver should take steps to improve its performance, perhaps by incorporating a network-specific low-level acknowledgment and retransmission protocol.

Many types of networks have been integrated into the Pup architecture, each with a different level 0 driver. Since there are many different host computers in the Pup internet, most drivers have been implemented in several languages and under several operating systems.

Ethernet. An Ethernet was one of the first networks to be used as a packet transport mechanism [Metcalfe & Boggs, 1976]. A packet can be broadcast to all other Ethernet hosts by setting the destination address to zero. An Ethernet has good reliability: typical packet loss rates range from one in a thousand to one in a million, though under extreme overload it can drop to one in ten [Shoch, 1979].

Arpanet. Pups have been transported through the Arpanet. Arpanet Pup transport is based on Host-Imp protocol messages, not Host-Host protocol streams. The Arpanet provides high reliability through the use of its own internal acknowledgment and retransmission protocols—far more reliability than Pups need. The Arpanet does not presently support broadcast packets; rather than sending packets to all possible Arpanet hosts, the network driver

does not implement broadcasts at all. This limits the utility of the Arpanet for Pup.

Leased line store-and-forward network. Different local networks are interconnected over long distances through the use of a private store-and-forward network constructed using telephone circuits. Similar in spirit to the Arpanet, this system is used to connect internetwork gateways. Unlike the Arpanet, the system does not use separate packet switches (Imps), but instead switches packets through the gateway hosts themselves. The system is fairly reliable and does not require low-level acknowledgments. When asked to broadcast a Pup, the driver sends copies down each outgoing phone line.

Arpa Packet Radio network. The Arpa Packet Radio network [Kahn *et al.*, 1978] is used to carry traffic among local networks in the San Francisco Bay area [Shoch & Stewart, 1979]. Although radio is a broadcast medium, the packet radio network has a store-and-forward structure, and as we shall see in chapter 4, an efficient broadcast addressing mode for a store-and-forward network requires some mechanism in the network's switches. The Arpa packet radio design does not include this mechanism, so the Pup driver must build its own broadcast. The driver periodically identifies Packet Radio hosts that might be running Pup software and when asked to broadcast a Pup, the driver sends copies to all such hosts.

2.2.2. Level 1: internetwork packets

This is the level at which addresses, packet formats, and the routing algorithm are defined. It is the lowest level of interprocess communication.

Addresses. Pups contain source and destination addresses. These hierarchical addresses include an 8-bit network number, an 8-bit host number, and a 32-bit socket number. By convention, hosts are expected to know their own host numbers, to discover their network numbers by locating gateways and asking them for this information, and to assign socket numbers in some systematic way not legislated by the internet.

There are some important conventions associated with the use of network addresses. A distinguished value of the network number field refers to "this network" without identifying it. Such a capability is necessary for host initialization (since most hosts have no permanent local storage and consequently no *a priori* knowledge of the connected network number), and to permit communication to take place within a vestigial internet consisting of an unidentified local network with no gateways. A distinguished value of the destination host number is used to request a broadcast. Certain values of the socket number field refer, by convention, to "well-known sockets" associated with standard, widely-used services, as is done in the Arpanet.

Pup format. The format of a Pup packet is shown in figure 2.3. The following paragraphs highlight the sorts of information required at the internet packet level.

The *Pup length* field is the number of 8-bit bytes in the Pup, including the internetwork header (20 bytes), contents, and checksum (2 bytes).

The *transport control* field is used for two purposes: as a scratch area for use by agents that handle the Pup ("office use only"), and as a way for source processes to advise the internet about how to handle the Pup ("time value", "bulk rate", *etc.*). Other network designs call this the *facilities* or *options* field.

The *Pup type* field is assigned by the source process for interpretation by the destination process and defines the format of the Pup contents. Interpretation of the type field is strictly a matter of agreement between the source and destination processes—gateways do no type-dependent processing of packets.

The *Pup identifier* field is used by most protocols to hold a sequence number. Its presence in the internetwork header is to permit a response generated from within the internet (*e.g.* error or trace information) to identify the Pup that triggered it in a manner that does not depend on knowledge of the higher-level protocols used by the end processes.

The *destination* and *source addresses* are self-explanatory.

The *data* field contains up to 532 data bytes. Much of the internet traffic consists of packets containing individual "pages" of 512 bytes each, reflecting the quantization of memory in most of the computers in the Pup internet. Just carrying the data is often not enough, since the packet should accommodate higher-level protocol overhead and some identifying information as well. Allowing 20 additional bytes for such purposes, we arrive at 532 bytes as the maximum size of the data field.

The optional *software checksum* is used for complete end-to-end coverage—it is computed as close to the source of the data and checked as close to the ultimate destination as is possible. This checksum protects a Pup when it is not covered by some network-specific technique, such as when it is sitting in a gateway's memory or passing through a parallel I/O path. Most networks employ some sort of error checking on the serial parts of the channel, but parallel data paths in the interface and the I/O system often are not checked.

Routing. Accompanying the addressing conventions and packet format defined at level 1 are the procedures for internetwork routing. Each host, whether or not it is a gateway, executes a routing procedure on every outgoing Pup. The difference between a gateway and any other host is that a gateway accepts packets for routing from hosts on its directly-connected networks as well as from processes inside itself.

The routing procedure decides, as a function of the Pup destination network number, upon which directly-connected network the Pup is to be transmitted (if there is more than one choice), and it picks an *immediate destination host* that is the address on that network of either the ultimate destination or some gateway believed to be closer to the destination. Each routing step employs the same algorithm based on local routing information, and each Pup is routed independently.

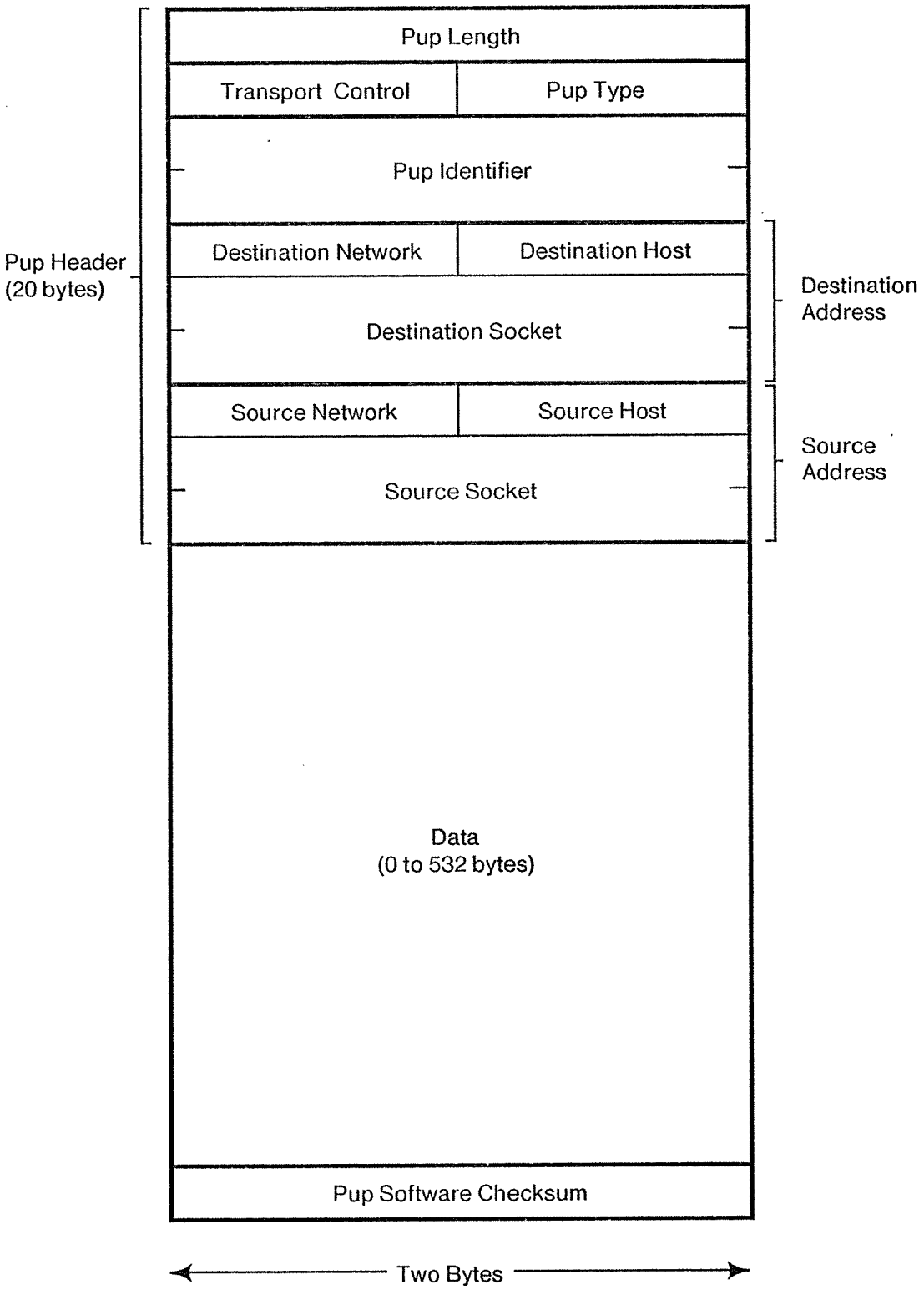

Figure 2.3. Format of a Pup packet.

Routing information is maintained in a manner similar to the "old" Arpanet distributed adaptive procedures [McQuillan, 1974]. The metric used for selecting routes is the *hop count*, namely the number of intermediate gateways between source and destination. The protocol for updating the routing tables involves exchanging Pups with neighboring gateways and rests logically at level 2 of the protocol hierarchy. The routing protocol is an ideal application for broadcast packets, and is described in section 2.4.

A host that is not a gateway still implements a portion of this level-2 routing-update protocol: it initially obtains an internetwork routing table from a gateway on its directly-connected network, and it obtains updated information periodically. If there is more than one gateway providing connections to other networks, a host can merge their routing tables and send packets to the gateway offering the best route to each network.

2.2.3. Level 2: interprocess communication

Given the raw packet facility provided at level 1, we can begin to build data-transport protocols tailored to provide appropriate levels of reliability or functionality for real applications.

These protocols generally fall into two categories: those in which a connection is established for a sustained exchange of packets, and those in which individual packets are exchanged on a connectionless basis. Connection-style protocols usually transport data very reliably and transparently. Several are now described.

EFTP: Easy File Transfer Protocol. This is a very simple protocol for sending files. Each data Pup gives rise to an immediate acknowledgment, and there is at most one Pup outstanding at a time. This protocol is a descendant of the one described by Metcalfe and Boggs [1976]. Its simplicity makes this piece of communication mechanism easy to include under conditions of very limited resources. For example, a complete EFTP receiver has been implemented in 256 words of minicomputer assembly language, for use in the network-based bootstrap and down-line loading process described in section 3.2.

RTP: Rendezvous/Termination Protocol. This protocol is a general means to initiate, manage, and terminate connections in a reliable fashion [Sunshine & Dalal, 1978]. In normal use, an RTP user initiates a connection by communicating with a well-known socket at some server. That server will spawn a new socket to actually provide the service, and the RTP will establish contact with this socket. The result of a rendezvous is that a pair of sockets agree on a 32-bit *connection identifier*. This number is included in packets to distinguish among multiple instantiations of the same connection and to cope with delayed packets without making assumptions about maximum packet lifetimes.

BSP: Byte Stream Protocol. This is a relatively sophisticated protocol for supporting reliable, sequenced streams of data. It provides for multiple outstanding packets from the source, and uses a moving-window flow control procedure. User processes can place *mark bytes* in the stream to identify logical boundaries and can send out-of-band *interrupt* signals. RTP and BSP combined perform a function similar to that of the Transmission Control Protocol (TCP) [Cerf & Kahn, 1974; Postel, 1979], with which they share a certain amount of common ancestry.

We now turn to some connectionless protocols. Connectionless protocols do not attempt to maintain any long-term state; they usually do not guarantee reliability, but leave it up to the designer to construct the most suitable system. Their simplicity and ease of implementation make them extremely useful.

Echo. A very simple protocol can be used to send test Pups to an *echo server* process that will check them and send them back. Such echo servers are usually embedded in gateways and other server hosts to aid in network monitoring and maintenance. The server is trivial to implement on top of the level-1 facilities. It is connectionless because the server does not remember anything about a packet that it has echoed: two or more clients can simultaneously echo packets off the same server.

Name lookup. Another server (described in section 3.1) provides the mapping between the names of resources and their addresses. This service is often addressed with a broadcast Pup, since it is used as the first step in locating a resource. The name lookup service itself must be located at a well-known socket.

Routing table maintenance. The internetwork routing tables are maintained by Pups exchanged periodically among internetwork gateways and broadcast for use by other hosts. This protocol is described in section 2.4.

Date and time. When a time server receives a time request Pup, it responds with a Pup containing the date and time. The request is usually broadcast; the time protocol is described in section 3.3.

Other connectionless protocols are used to access a *boot server*, an *authentication server*, and a *mail check server* integrated with an on-line message system. These protocols are designed to be as cheap as possible to implement (*i.e.* without connection overhead) so that such servers may be replicated extensively and accessed routinely without consuming excessive resources. For example, instances of some of these servers are present in all gateway hosts so as to maximize their availability.

2.2.4. Level 3: application protocols

Armed with a reasonable collection of data transport protocols at level 2, one can begin to evolve specific applications at level 3. These are supported by various function-oriented protocols.

Telnet. Terminal access to remote hosts is provided with an internetwork Telnet protocol, which makes use of the combination of the Rendezvous/Termination Protocol (RTP) and the Byte Stream Protocol (BSP) at level 2. Using the notion of a *virtual terminal*, Telnet implementations map characteristics of actual terminals into a network-independent representation: a mark byte in the stream and an out-of-band interrupt, for example, are used to signal an "attention" [Davidson *et al.*, 1977; Feinler & Postel, 1978].

File Transfer Protocol. The RTP and BSP are again combined as the foundation for an internetwork File Transfer Protocol (FTP), supporting stream-oriented access to files. The underlying byte stream provides reliable communication. The major task of FTP is to communicate commands and responses and to sort out different representations of data in different file systems. FTP is one of the principal protocols used to access dedicated, high-capacity network file servers.

Printing. Among the important shared resources in the internet are high-quality printing servers such as the one that printed this thesis. Rather than using the fully developed BSP and FTP, the specialized task of sending unnamed, standard-format document files to a printer makes use of the more restricted but much simpler EFTP.

CopyDisk. Given high-performance networks and simple gateways that can forward Pups among them efficiently, it is perfectly reasonable to copy entire disk packs through the internet. The CopyDisk protocol negotiates between the participating machines to ensure that the disks are compatible, and handles error recovery should something go wrong.

Additional applications have included cooperative editing of common documents from multiple machines, packet telephony, and many others. As users create new applications, these systems tend to develop their own natural layering of function. Some may require new protocol designs in the existing hierarchy; the Pup architecture permits this degree of flexibility down to the level of the simple internetwork packet. As new systems are built, common pieces begin to emerge that might be of more general use; they eventually find their way into an appropriate place in this hierarchy of communications protocols.

2.3. Implementation

This section sketches an implementation of levels 0 and 1 of the Pup architecture. These layers are the Internet proper, namely the layers that move Pups between client processes.

Figure 2.4 is a block diagram of the internet software in an ideal host machine. The boundaries between levels are shown as dashed lines. Client processes at level 2 are the ultimate producers and consumers of Pups. Level 1 routes outgoing Pups to network drivers and incoming Pups to client sockets. The level-0 drivers transmit and receive Pups through various types of networks.

2.3.1. Level 1: internet packets

Level 1 is principally concerned with routing. The output process routes a Pup to a network based on the destination network field of its address. The input process routes a Pup to a socket based on the destination socket field of its address.

The routing table is shown inside level 1 because its primary client is the output process. The table is maintained by a level-2 process using the protocol described in section 2.4. The information in the routing table is also useful to clients above level 1, so a public interface (at least to read it) is required.

The interface to level 2 consists of socket queues and the output queue. If the host makes a distinction between the "system" and "users", then this is where the line is drawn. Widely used level-2 protocols such as the BSP can be implemented by the "system" for efficiency, but the functionality at the level 1/2 interface should be such that an ordinary user process could implement BSP if it wanted to.

2.3.1.1. Sockets

A socket is the ultimate destination of a Pup. It is the principal interface between a client and the internet, and generally contains other items besides the socket queue. A socket may be owned by a single process or shared by several processes, but this is of no concern to the internet. The internet's job is done when a Pup is queued on a socket's input queue.

Although sockets are often spoken of as the ultimate source of Pups, this is not strictly true. Sockets do not contain output queues, but rather there is one common output queue shared by all processes. This is analogous to a postal system where each person has a private mailbox for incoming letters, but many people share the public mailbox on the corner for outgoing letters. Just as the post office does not check that the return address on a letter is the sender's, so the internet does not check that the source address on a Pup is that of some local socket. On the other hand, if an answer is expected, it is ordinarily returned to the source address, so a socket is strongly associated with a source of Pups.

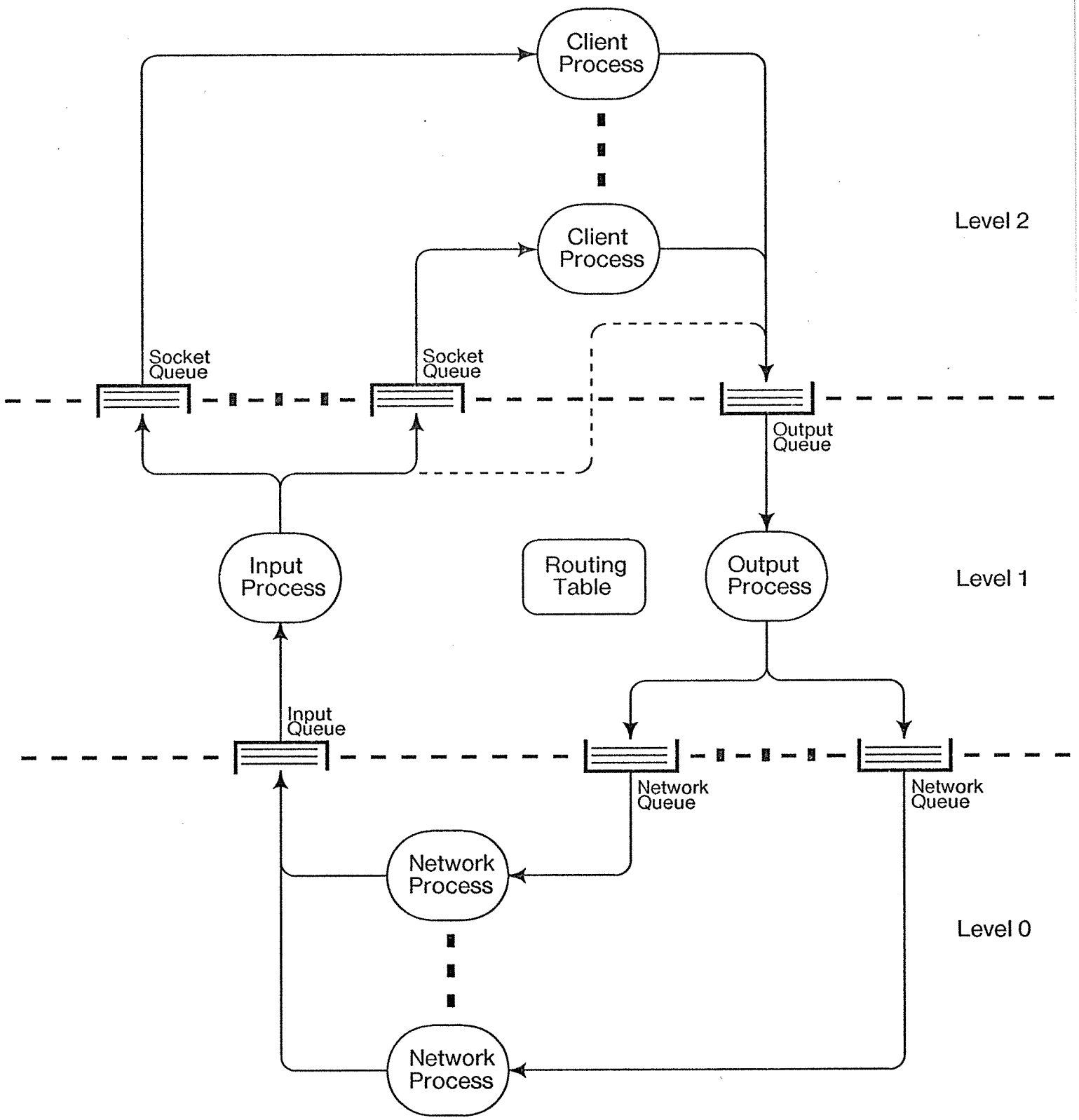

Figure 2.4. Pup in an ideal host.

2.3.1.2. The output process

All Pups generated within a host are put on its output queue when ready for sending. As part of putting a Pup on the output queue, "systems" often default fields from information kept in the data structure associated with the socket. Examples are filling in the source and destination addresses, and computing the software checksum.

The output process routes Pups from the output queue to a network queue. Assume that a Pup's destination address is (network *A*, host *b*). The output process looks up network *A* in the routing table. The routing table entry for network *A* contains a pointer to a network queue, a hop count, and a gateway host number, *c*. If the hop count is zero, then the network queue pointed to is for network *A* itself, and the Pup should be sent to host *b*. If the hop count is nonzero, then the network queue is for some transit network, and the Pup should be sent to the gateway, *c*.

2.3.1.3. The input process

The input process routes Pups from the input queue to a socket queue or the output queue. First it decides whether the Pup is addressed to a socket in the local machine. If so, the Pup is put on the proper socket queue; if no such socket exists, the Pup is discarded.

Pup socket numbers are 32 bits long so that systems assign sockets in such a way that they are rarely reused. If a client is concerned about receiving packets from unexpected sources (*e.g.* delayed duplicate packets from a previous conversation at that socket), it can check the source and destination sockets.

A Pup not addressed to a local socket should only arrive at a gateway host; an ordinary host that receives such a Pup should discard it. A gateway's input process checks the Pup's checksum and discards it if bad, otherwise it puts the Pup on the output queue for the output process to route closer to its destination. A gateway host often accumulates statistics on Pups it forwards.

2.3.2. Level 0: packet transport

A level-0 driver in this ideal model is a single process. A real network controller would probably use interrupts, which are best thought of as slightly special processes. In addition, most drivers have a housekeeping process. But for our purposes this detail can be suppressed, and a single process from which Pups emerge and into which Pups disappear, suffices.

One of the important functions of a level-0 Pup driver is address translation. Pup host numbers are integers in the range 0 to 255, but particular networks may have different addressing schemes. A Pup host number is associated with each Pup that crosses the level 0/1 interface. The output process supplies the immediate destination host number to which the driver should send the Pup. The driver supplies the immediate source host number from

which the Pup arrived. These are Pup host numbers and it is the network driver's responsibility to map between them and the network-specific addresses used when encapsulating and decapsulating Pups.

2.3.2.1. An Ethernet driver

A Pup is encapsulated in an Ethernet packet by adding Ethernet destination and source addresses and an Ethernet packet type. Pup host numbers and Ethernet host addresses map one-to-one, so no address translation is necessary. The immediate destination host number supplied by the level-1 output process is directly used as the Ethernet destination address.

By convention, Ethernet packets contain a packet type that indicates the format of the rest of the packet; the code for a Pup says in essence "I'm a Pup". Other internet architectures can also transport packets through Ethernets, using different packet types to distinguish them.

When a packet is received with good controller status, the Ethernet packet type field is checked, and non-Pups are discarded. As a consistency check, the length of the Pup plus the length of the Ethernet encapsulation is compared with the packet length reported by the controller hardware; mismatches are discarded. The surviving Pups are put on the level-1 input queue.

2.3.2.2. A leased line driver

A leased point-to-point or dial-up telephone circuit is often used to interconnect distant gateways. The Pup architecture organizes leased lines into a store-and-forward network. The store-and-forward switching function is done in the level-0 driver rather than in a separate computer as in the Arpanet.

In addition to Pups, the drivers must exchange routing information for the store-and-forward network; this should not be confused with the internet routing tables at level 1. A level-0 routing table maps a host address in the store-and-forward network to a leased line that will get packets closer, while the level-1 routing table maps a network number in the internet into a network driver that will get Pups closer. In the Pup implementation, drivers send routing packets out each line every 5 seconds. Routing packets have priority over other packets awaiting transmission.

A Pup is encapsulated for a leased line by adding source and destination addresses and a packet type. Since this network was specifically designed for Pup, its addresses map one-to-one with Pup host numbers. The level-0 routing table is consulted to pick an outgoing leased line. To improve the delay for interactive traffic, small Pups awaiting transmission down a line are promoted in front of large Pups. However, they are *not* promoted in front of large Pups with the same source and destination addresses, because this would cause them to be delivered out of order. While hosts must tolerate out-of-order delivery, it is often expensive and should

be minimized.

When a properly framed packet with a good line-protocol checksum is received, the packet type field is checked and if it is a level-0 routing packet, the driver updates its routing table. If it is a Pup, its length is checked against the packet length reported by the controller and mismatches are discarded. The surviving Pups are put on the level-1 input queue.

This network does not implement a proper broadcast. When asked to broadcast a Pup, it sends a copy down each line. This is enough of a broadcast to permit some simple broadcast-based applications to work, but it is not right. The correct solution is to implement a full broadcast using one of the techniques discussed in chapter 4.

2.3.2.3. An Arpa packet radio driver

The Arpa packet radio driver is the most complex Pup level-0 driver yet implemented. A packet radio host must use the packet radio network's *Channel Access Protocol,* or CAP, to send and receive packets. All packets must contain a CAP header, and hosts must periodically handshake with the network by exchanging *terminal-on* and *repeater-on* packets.

Packet radio addresses are 16 bits long, so the driver must map between them and the 8-bit Pup host numbers used by levels 1 and above. A subset of packet radio addresses contains all possible Pup addresses, and the driver periodically probes each address using a private (*i.e.* level-0) protocol to decide whether a Pup driver is active there. A Pup is broadcast by sending a separate copy to each active Pup driver.

Pups are bigger than packet radio packets so they are fragmented by the sending driver and reassembled by the receiving driver. A Pup fragment is encapsulated by adding a CAP header, a type field, and fragmentation information. When a Pup fragment is received, a queue of fragments awaiting reassembly is searched. If other fragments are found, the new one is added and if that completes a Pup, it is put on the level-1 input queue. Pups awaiting reassembly are timed out and discarded if all fragments do not arrive within a few seconds.

2.4. Pup routing

This section describes the Pup routing system, by which a host discovers the identity of its directly-connected networks(s) and locates gateways to other networks. The routing information protocol was the first application of broadcast packets in the Pup architecture. This is the first of several case studies of broadcast-based protocols; more are presented in the next chapter.

2.4.1. An example

Figure 2.5 illustrates Pup routing. The source process located in host ***s*** on network ***A*** sends a Pup to the destination process in host ***d*** on network ***B***. Since the two hosts are not on the same network, the Pup must be forwarded through a gateway.

The source host's output process looks up network ***B*** in its routing table. The entry for network ***B*** says that it is one hop away via gateway host ***g*** on network ***A***. The output process hands the Pup to the level-0 driver for network ***A*** with instructions to send the Pup to host ***g***.

Gateway host ***g***'s input process notices that the Pup is not addressed to a local socket and hands it to the output process (via the path shown as a dashed line in figure 2.4). The output process looks up network ***B*** in its routing table. The entry for network ***B*** says it is zero hops away (*i.e.* directly connected). The output process hands the Pup to the level-0 driver for network ***B*** with instructions to send the Pup to host ***d***.

Host ***d***'s input process notices that the Pup is addressed to local socket ***p*** and so it puts it on ***p***'s socket queue.

2.4.2. Hops—the routing metric

Each time a Pup passes through a gateway to another network, that transit is counted as a *hop*. Currently, Pup routing decisions are based only on connectivity. This strategy views a one-hop path through one kilobit-per-second phone line as more desireable than a two-hop path through two megabit-per-second Ethernets. Using hops as the routing metric has worked remarkably well up to the current internet size of about 50 networks, but it is time to improve it.

Excess bandwidth and delay are two other useful routing metrics. The route from ***A*** to ***B*** with the most excess bandwidth is not necessarily the same as the route from ***A*** to ***B*** with minimum delay or hops. Bulk data traffic (*e.g.* an FTP connection) wants maximum bandwidth, and delay is usually of secondary importance. Interactive traffic (*e.g.* a Telnet connection) wants low delay, and typically does not need much bandwidth. Two currently unused bits in the Pup transport control field could be used by clients to advise the internet on how to route a Pup. One bit would mean "route this Pup for minimum delay", and the other would mean "route this Pup for maximum bandwidth".

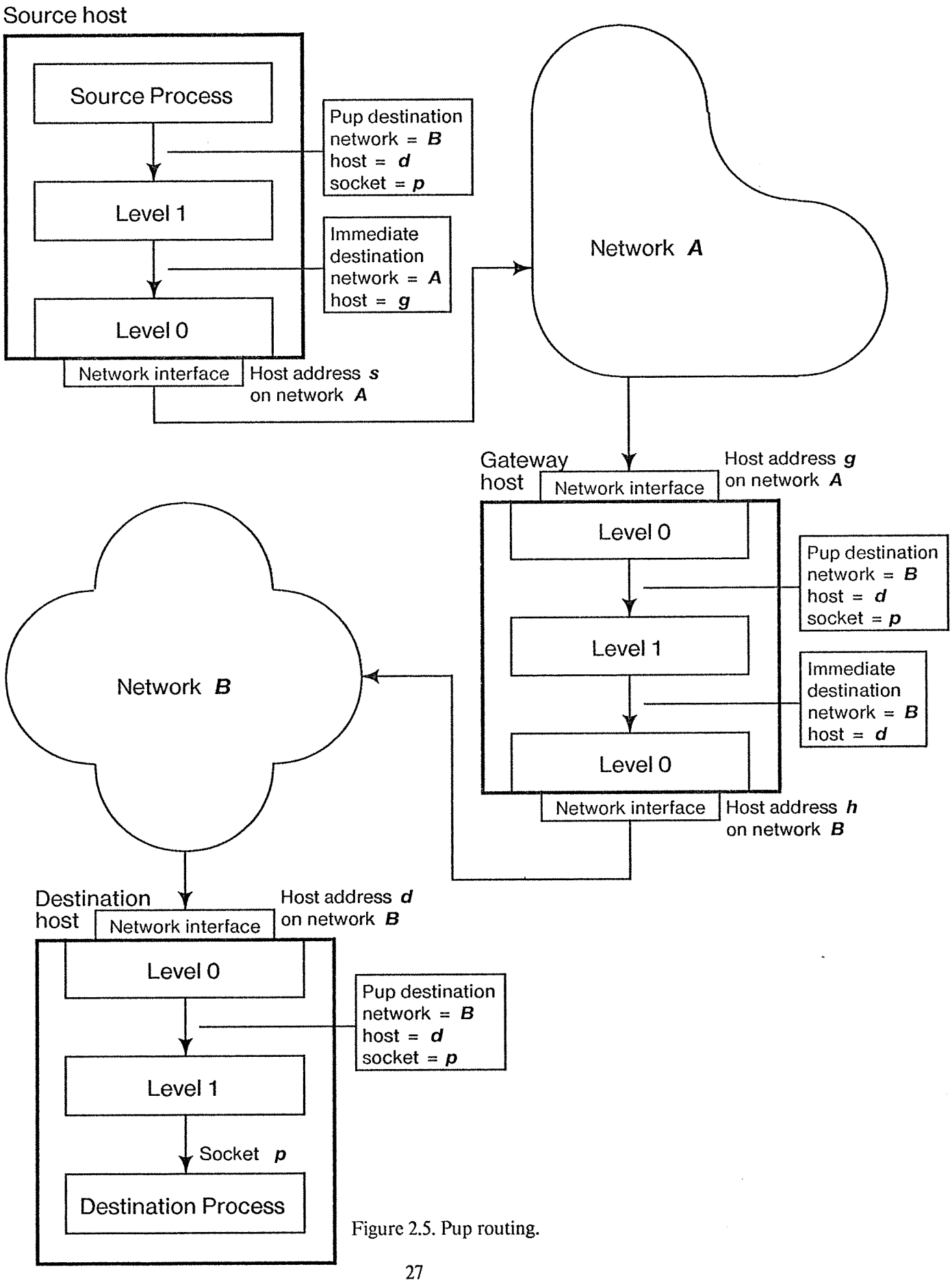

Figure 2.5. Pup routing.

The information in the routing table is useful to clients. The routing table amounts to a map of the internet, with the owning host at the center. If the routing system maintains several metrics, then there are several maps. *A* may be closer than *B* using hops as the metric, but *B* may be closer than *A* using the delay metric. The routing table should be readable by clients. That is, a process ought to be able to do things such as build a list of networks sorted by increasing delay from here to there. The broadcast techniques of chapter 4 make heavy use of the routing table.

It is possible for the routing tables in the internet to become momentarily inconsistent, and packets can end up travelling in a loop. To quench this behavior, a maximum path length for the internet, *maxHops,* is specified, and a subfield in the transport control field of a Pup's header is reserved for hop counting. When a Pup first enters the internet, its hop count field is zeroed. Each time it is forwarded by a gateway, the field is incremented. If maxHops is exceeded, the Pup is discarded. It is sometimes useful to know that a network exists but is inaccessible. To represent this information in a routing table, a hop count of *maxHops+1* is considered infinite.

2.4.3. Getting started

A Pup host contains a "built in" process that maintains the routing table by sending and receiving Pups through a well-known socket which it opens at initialization time. When a host starts up, it knows its host address on each network to which it is directly connected, but it does not know their network numbers and its routing table is empty. One of these packets, broadcast on each directly-connected network, solves both problems:

> *Pup type:* Routing information request
> *Pup contents:* none.

Gateways respond with:

> *Pup type:* Routing information reply
> *Pup contents:* (network-number, hop-count) pairs.

The (network-number, hop-count) pairs are the raw data from which routing tables are built. If the number of networks is so great that they will not all fit in a single Pup, a gateway may respond with several Pups. Since a newly-starting host does not know network numbers, it sets the network fields of the destination and source addresses of its routing information request packets to zero. Gateways *do* know their network numbers, so their reply packets contain the correct network numbers, and thus the host discovers them.

2.4.4. Keeping up to date

There are two update activities going on in the routing system and they are both accomplished with a single broadcast packet.

Gateway-gateway update. The routing algorithm requires a gateway to periodically update the routing tables in neighboring gateways. One way to do this is for the routing process to maintain a list of all other gateways on directly-connected networks, and to periodically send each one a copy of the table. Another way to do it is for a gateway to periodically broadcast its routing table on each of its directly-connected networks.

Host-gateway update. Regular, non-gateway hosts also need up-to-date routing information from nearby gateways. One way to do this is for each host to maintain a list of nearby gateways, and periodically ask each one for its routing table. Another way to do it is for the gateway to periodically broadcast its routing table on each of its directly-connected networks.

Having each gateway periodically broadcast a routing information reply packet on all of its directly-connected networks accomplishes both updates. Nobody needs to keep a list of nearby gateways, and fewer total packets are sent. Pup gateways broadcast routing information every 30 seconds, or right after a significant change occurs. The broadcast interval is a compromise. The smaller the interval, the more likely it is that all hosts will have the same information and the more timely that information will be. On the other hand, all hosts must expend computational resources processing these packets, so the cost goes up as the interval goes down.

When a routing information reply Pup is received (either in response to a host's initial request, or gratuitously), each (network-number, hop-count) pair is processed and the local routing table updated in the following way. If the network is directly connected (*i.e.* exists in the routing table with a hop count of zero), then no update is necessary. Otherwise, the table is updated if any of the following conditions is true:

1. No entry for the network presently exists in the local routing table.
2. The routing information Pup is from the gateway through which Pups are currently being routed.
3. The existing entry in the local routing table is more than 90 seconds old, suggesting that it may no longer be valid.
4. The hop count in the Pup, plus one, is less than the hop count in the existing local routing table entry.

Case two above gives special priority to the creator of a routing table entry. That is, if gateways ***a*** and ***b*** are both one hop away from network ***C***, but gateway ***a*** is listed in the table

as the path to *C*, then receipt of a routing info Pup from gateway *b* listing its path to *C* will not change the routing table entry for network *C*. This prevents path oscillations.

Since a gateway broadcasts its routing table every 30 seconds, case three above implies that three consecutive broadcasts have been missed from the gateway that created the entry. It is prudent to suspect that some catastrophe has befallen it. At this point the entry is eligible for update by any gateway that claims to have any route at all to the subject network. Entries more than 180 seconds old are simply purged.

2.4.5. Gateway considerations

To be a gateway, a host must implement the dotted line path from the input process to the output queue in figure 2.4, broadcast its routing table periodically, and respond to routing information requests. A gateway must also take into account some special considerations in managing its routing table.

When a gateway's routing table changes (either becuase one of its directly-connected networks has changed state or because it received a routing packet from some other gateway reporting a distant change in the internet's topology), it immediately broadcasts a routing information Pup reflecting the updated entry, rather than waiting for the next periodic routing broadcast. This ensures that changes in topology propagate through the internet immediately rather than at a rate limited by the background 30-second broadcast interval.

When a formerly accessible network becomes inaccessible, the gateway continues to include an entry for it in the routing table (with a hop-count of infinity) for 90 seconds. This ensures that the knowledge of the network's inaccessiblity will propagate through the internet quickly.

When a gateway is about to go down voluntarily, it broadcasts several routing information Pups in which all entries are marked inaccessible (infinite hop count), but continues to forward Pups for a few seconds. This ensures that other routes will be found immediately rather than as a result of the 180-second timeout.

2.4.6. Broadcasting on all directly-connected networks

Broadcasting a Pup on all directly-connected networks is used as a bootstrap routing strategy since it does not depend on the contents of the routing table. This is fortunate, since a host's routing table is initially empty when it is trying to locate gateways. The case studies in the next chapter also use this broadcast technique, but for different reasons. The old Arpanet routing algorithm used this technique too: periodically an Imp sent a copy of its routing table down each of its lines to its neighboring Imps. Chapter 4 calls this a *nearest-neighbor multicast.*

Directly-connected networks have routing table entries with hop counts of zero. Therefore one way to broadcast on all directly-connected networks is to enumerate the routing table and queue a copy on each network input queue pointed to by a routing table entry with a hop count of zero. The routing process can not do it this way, since the routing table might be empty; it must have a way to enumerate all network drivers independent of the routing table.

2.4.7. Arpa internet routing

Comparing the Pup routing scheme to that of the Arpa Internet highlights the benefits of broadcasting [Strazisar, 1979]. The basic strategy for building routing tables is the same; the difference is in how hosts and gateways discover each other.

Each Arpa gateway starts out life knowing the network numbers of the networks to which it is directly connected and the host numbers of other potential gateways on those networks. Not all of the host numbers are necessarily functioning gateways, and there may be other gateways that are not in the original list. Each gateway periodically polls each potential gateway address to discover if the host at that address really is a gateway. If a gateway receives a poll from another gateway that is not in its list, then it adds the new gateway address to its list and begins polling it. The gateway sends routing table updates in separately-addressed packets to each gateway in the list that is answering polls.

Each Arpa host starts out life knowing the numbers of the networks to which it is directly connected, and the addresses of potential gateways on those networks. In theory, each host should poll each potential gateway to discover whether it is functioning; in practice this generates a large load on the gateways, so hosts do not poll. When a gateway receives a packet for forwarding, it checks its routing table to see it there is another gateway on the same network which is a better route. If there is a better route, the gateway sends a packet back to the host telling it about this other gateway and the host updates its routing table.

The ability to broadcast permits the Pup architecture to have much less "wired-in" state and to avoid polling altogether. A Pup gateway starts out knowing the identity of its directly-connected networks, but not the locations of other gateways. A Pup host starts out knowing neither the identity of its networks nor the locations of other gateways. Minimizing wired-in state is essential in an environment dominated by personal computers in which the only stable storage is on removable media that can and routinely does move from host to host and network to network. Having state wired in is not as operationally painful for Arpa internet hosts since they are predominately larger systems in which stable storage is permanently associated with each CPU.

2.4.8. Summary

Getting a little ahead of the story, notice that broadcasts are used for two distinct purposes in the routing protocol. Hosts broadcast routing information request packets to locate gateways. Gateways broadcast routing information reply packets to distribute updated routing information to hosts and other gateways.

In both cases the broadcaster does not know the specific addresses to which packets should be sent. This is the fundamental reason for broadcasting. Hosts broadcast to *locate* gateways. Gateways broadcast to *distribute* routing information. Locating and distributing are the two principal situations in which addresses are unknown.

The routing process in a gateway is acting as a high-level store-and-forward gateway for routing information. A routing process updates its table from an incoming routing information packet and then forwards the updated table by broadcasting. Routing information moves from one network to another by passing through routing processes. That is, the protocol depends on routing processes being located in hosts that are directly connected to more than one network. This is not always possible for applications other than routing, as we will see in the next chapter, and the simple broadcast-on-all-directly-connected-networks technique needs augmentation.

Chapter 3

Why broadcast?

> The most straightforward way to keep up-to-date information would be to have each STASER [RSExec Status Server] process periodically "broadcast" its own status to the others. Unfortunately, the current connection-based Host-Host protocol of the ARPANET forces use of a less elegant mechanism.
>
> [Thomas, 1973]

This chapter consists of more case studies of situations in which broadcast packets have proven to be very useful. The applications described here are in everyday use in the Pup internet. Isolated instances of some of these protocols exist in other networks [Harrenstien, 1977] and internets [Pickens *et al.*, 1979], but I know of no other system in which so many different protocols have reached the level of sophistication and widespread use described here. This is primarily due to the presence of a broadcast facility in Pup.

The first broadcast mechanism built into Pup was for use by the routing system and allowed a host to broadcast a packet on all of its directly-connected networks. The protocols described in this chapter were developed using this broadcast mechanism. As the internet and these applications grew, it became clear that a more powerful and general mechanism was needed. The solution was *directed broadcast*, the ability for a host to broadcast a packet on any network in the internet, not just ones to which it was directly connected.

To motivate the need for the more general mechanism, the protocols described in this chapter do not use directed broadcasts. At the end of each protocol description, the shortcomings of only being able to broadcast on directly-connected networks are pointed out. Discussion of directed broadcasting is deferred until chapter 5.

3.1. Converting between names and addresses

This service was implemented early in the development of Pup and has undergone continuous improvement. The first version only converted names to addresses, and the service became known as a *name server*. Later, conversion in the other direction, from addresses to names, was also implemented. Being able to refer to a resource with a string name rather than a group of numbers significantly improves the quality of life perceived by users of the internet.

Names are strings meant for human consumption; addresses are fixed-format bit arrays optimized for use by the internet. Humans normally do not see addresses. Not only is this a kindness, it permits a resource to move around (*i.e.* change its address). Clients of the internet are exhorted to remember only names, and to delay binding to an address until just before sending packets. This way, if a resource moves, it suffices to change its entry in the host name database, and people will still be able to find it.

This name service is specifically for binding host names to internet addresses. A similar mechanism could be used to bind other kinds of names to addresses, for instance, user names to mailbox addresses. The generalization of this concept is called a *clearinghouse* [Abraham & Dalal, 1980]. Such a service might accept a context (such as "host names" or "mail boxes") and a person's name (such as "Boggs"), and return a list of addresses (or accept a context and an address and return a list of names).

The Pup name server uses a centrally maintained database. A master copy exists somewhere in the internet and it is distributed to all servers. When the system is in equilibrium, all servers have the same copy.

Where should a name server be located? In a host that is up most the time, and has some spare file space in which the database can be kept. Gateways and file servers are good candidates. When a server is placed in a gateway, care must be taken that its presence not adversely affect the gateway's primary function, namely forwarding packets. If there are several name servers on a network, then the probability that all of them are down will be less than the probability that the network itself is down.

The name service protocol divides into two parts: converting between names and addresses (*database lookup*), and distributing new copies of the database (*database update*). The user half of the lookup protocol is implemented by all hosts, while the update protocol is only implemented by servers.

3.1.1. Database lookup

There are two symmetric operations in a name server: given a name, return an address, and given an address, return a name. Both operations can fail for a number of reasons, in which case an error is returned instead of a result. Name database lookup must be fast, cheap, and widely available.

A name service request is sent in a single Pup, and its response (name, address, or error) is also a single Pup. A request is usually broadcast because the requestor does not know the address of a name server host. Many networks have several name servers, so a broadcast request can trigger several replies. They will be identical unless the request is for a database entry that has changed and the request was made while the updated database was being distributed. Replies should, but currently do not, contain the version number of the database so that a requestor can choose the most recent one. Most clients discard all responses after the first.

Pup type: Name lookup request
Pup contents: a name string

Pup type: Name lookup response
Pup contents: one or more addresses.

It is conceivable, but not very likely, that a host could have so many addresses that they would not all fit in a single Pup. In that case, as many addresses as would fit would be packed into a name lookup response Pup. It would then be transmitted, and another Pup filled, just as with routing information responses.

Pup type: Address lookup request
Pup contents: an address

Pup type: Address lookup response
Pup contents: a name string

If a name lookup or an address lookup fails, the server responds with an error packet. The contents is a string explaining why the lookup failed: bad syntax, no such entry in the database, *etc.*

Pup type: Database lookup error
Pup contents: an error string

A host can avoid repeated packet exchanges to look up the same name or address by caching recent answers. The cache should be flushed when a new version of the database is released, or when the entries get old (typically a few hours or days).

3.1.2. Database update

The update part of the protocol distributes new versions of the database to name servers. A new version should propagate through the internet to all servers quickly after it is released. If a server is down while the update is happening, then it should notice the new version soon after it restarts. Servers should not have to know where other servers are located.

The header record of the database includes a version number, and more recent databases have larger version numbers. A server that does not have a copy of the database is considered to have version number zero.

New versions of the database are released by briefly running a server on the host that creates the new version. This server broadcasts a Pup announcing the newer version of the database on all of its directly-connected networks:

Pup type: Database version
Pup contents: a version number.

A full-time name server nearby hears this broadcast packet and requests a copy of the new database. Once the new copy has been injected into one other server, the releasing host can go do other things. There is of course no need for that host to implement the lookup part of the protocol. The database version Pup is broadcast because the server does not know who might be interested in the new version (other servers and users who cache answers).

When a name server hears one of these Pups, it compares the incoming (remote) version number with the (local) version number of its copy of the database, and takes the following actions:

Remote = local. The remote name server has the same version of the database as the local server. No action is taken.

Remote < local. The remote name server has an older version of the database. The local name server broadcasts its version on all directly-connected networks.

Remote > local. The remote name server has a newer version of the database. The local name server requests the remote one to send a copy of the database.

Name servers also broadcast their versions once an hour. A typical network has one server, while some large networks have three or four. The cost of these background broadcasts is negligible.

When a server installs a new version, it immediately broadcasts the new version number. This causes the update to spread through the internet continuously, rather than in hops once per hour. If the version broadcast packet generated right after an update is lost in the internet, then the update will pick up again at most one hour later. The more frequent the background broadcasts, the more reliably an update will propagate; the cost is more network bandwidth and host processing overhead. A one hour interval between broadcasts has proven acceptable in the Pup internet where the database typically changes a few times a week.

Broadcasting the version number rather than sending it specifically to the out-of-date server (case two above) is an optimization. If there is only one out-of-date server, then all of the other servers within earshot will simply ignore the broadcast. If a new version is propagating, broadcasting serves to spread the word faster, and reduces the probability of the update stalling due to a lost "database version" Pup.

When a name server discovers another server with a newer version, it requests a copy:

Pup type: Send database
Pup contents: an address

When a name server receives one of these packets, it opens a connection to the specified address and sends the file. The requesting server does not destroy its old version until the new version has arrived safely, so if something goes wrong, its clients can still lookup names, albeit in a slightly out-of-date database.

Before installing a new database (and broadcasting its existence further), a server should examine it closely. Consider what would happen if a server had a corrupted version of the database that apparently had a very large version number. The corrupted version would spread throughout the internet, wiping out the correct version. For this reason, a name database contains a checksum which is checked before installation. Most servers also verify that the database version received is the same as the one they requested. When in doubt, it is best for a server to discard a suspect database and stick with the current version.

The system grows automatically; nowhere does there exist a list of all name servers to which a new one must be added before it can join in. When a server first starts, it broadcasts its current version on all directly-connected networks. This will cause its neighbors to inform it of any new versions that were released while it was not looking. A brand new server will have no database at all, and therefore it will broadcast a version number of zero.

3.1.3. Problems with the present protocol

A host that broadcasts a lookup request is searching for a name server with the same packet with which it is also requesting a database lookup. For this simple search strategy to work, there must be at least one name server on every network. This configuration constraint is usually acceptable.

Figure 3.1 illustrates the situation. A client, for example host ***a1*** on network *A*, converts a name into an address by broadcasting the request on his directly-connected network, *A*. The name servers in hosts ***a2*** and ***a4*** hear the request and respond (***a3*** hears but discards the packet since it is not a name server). Host ***a4*** is probably a gateway; host ***a2*** might be a file server.

A tighter configuration constraint is imposed by the way name servers propagate new versions of the database. Because a name server broadcasts its database version on its directly-connected networks, the only way for this information to move from one network to another is through a name server located in a host with connections to two or more networks. Such hosts are typically gateways that also happen to contain name servers. Notice the similarity between the propagation of new databases and propagation of routing changes. Routing information is distributed by broadcasting and moves from network to network by passing

through routing information servers located in gateways. There is little question of the appropriateness of locating routing information servers in gateway hosts; the argument for name servers is less compelling. A longer-range broadcast capability removes the restriction.

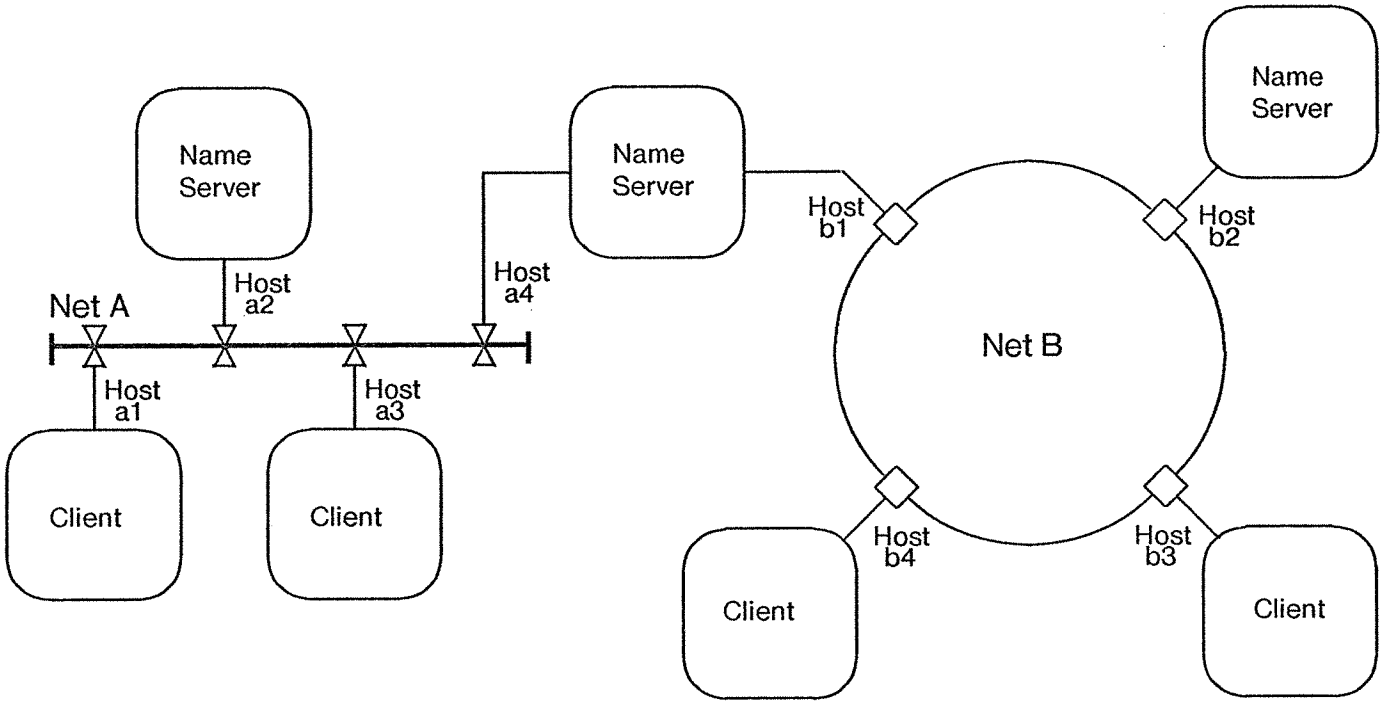

Figure 3.1. Name server example.

Imagine that a new version of the name server database is constructed by host ***b3*** on network ***B***. ***b3*** broadcasts the new version, which is heard by ***b2*** and ***b1***. They both request copies but ***b3*** only responds to ***b2***. ***b2*** now broadcasts the new version, and ***b1*** then gets it from ***b2***. ***b1*** then broadcasts the new version on both of its directly-connected networks, ***A*** and ***B***. There are no more takers on network ***B***, but host ***a2*** on network ***A*** requests a copy. Unless there is some way for any host on network ***B*** to broadcast a packet on network ***A***, the name server spanning the two networks is required for database propagation. The ability to broadcast a packet on a distant network is the topic of chapter 5.

3.2. Bootstrap loading

Bootstrap loading, or *booting* for short, is the operation of loading a program into a machine and starting execution, assuming a minimum about the machine's state. Most machines boot from some local device, such as a disk or tape, but in this application, the program arrives through the network from another machine referred to as a *boot server.*

The boot protocol consists of several smaller protocols: boot loading, locating servers and files, and distributing new file versions.

3.2.1. Boot loading

The first step in bootstrap loading is to get a boot loader into memory. The obvious way to do this is to find a boot server and ask it to send one. A boot server may be at any address and there may be several servers on a network, so a host in need of a boot would have to broadcast to locate one. Lacking a broadcast capability, one might fall back on a list of likely addresses. This is not only more complicated, but it also requires boot servers to be located at fixed addresses. Of course one could expand the list to include every host on the network, but this just amounts to doing a broadcast the hard way (this technique is called *separately-addressed packets* in chapter 4).

This problem can also be turned inside out by having boot servers periodically broadcast the loader. Alto computers use this solution because it takes less code to implement (it is not necessary to transmit, only to receive). In previous sections we have encountered protocols that broadcast packets on all directly-connected networks; distributing Ethernet boot loaders is an example of an even more restricted specification of how to broadcast a packet. The loader is specialized to booting an Alto through its Ethernet interface, so it is only broadcast on the Ethernets directly connected to the host containing the boot server.

Packet type: Alto boot loader
Packet contents: A boot loader program

This packet is broadcast every 5 seconds. These periodic broadcasts are a nuisance to most hosts. Instead of sending them to the general broadcast address, a special Ethernet address is reserved; only Altos in need of a boot listen for packets sent to this address. This is an example of a simple multicast. A boot server transmits a 4096-bit boot-loader packet every five seconds, consuming 0.03% of an experimental Ethernet's bandwidth. Several Altos can receive and execute the same boot loader.

When the loader arrives from the network, it broadcasts a packet requesting any boot server to send a program. This program is normally a *NetExec*, a *network* version of an *executive* or command processor program.

Pup type: boot file request
Pup contents: boot file number

Several servers may respond by trying to open a connection. The loader locks on to one of the servers by acknowledging its packets and the others give up within a short time. When the entire core image has arrived, the boot loader starts it.

3.2.2. Locating servers and files

The NetExec needs to locate boot servers and to find out what boot files are available. Since a boot server can be at any address and there can be any number of them, this is done by broadcasting. Each boot server responds with its *boot directory*, *i.e.* a list of all of its boot files. Each directory entry contains a boot file number, which is how a boot file is identified in a boot request packet, a boot file name, which is how a boot file is identified to a human, and a file creation date, which is used as a version number.

Pup type: boot directory request
Pup contents: none

Pup type: boot directory reply
Pup contents: boot file <number, name, date> tuples

Since not every boot server keeps every file, the NetExec merges directory responses to build its directory of available boot files. Many servers keep more files than can be described in a single Pup. They pack directory tuples into as many reply Pups as are required.

When asked to invoke a program, the NetExec transfers control to a copy of the loader, passing the boot file number to be loaded and the address of the server from which to get it. In this case the loader does not generate any broadcast packets.

3.2.3. Distributing new versions of files

Boot servers implement a protocol for distributing new boot file versions which is very similar to that used by name servers to distribute new database versions. When a new version of a boot file is released, the copy in every boot server must be updated. Not all boot servers are up all the time; manually locating and updating every copy of the file is operationally impossible. To solve this problem, the following scheme was devised. A boot server periodically broadcasts its boot directory. When a boot server hears one of these Pups, it compares the creation dates of the remote server's boot files with its local files and requests copies of any newer files. This is accomplished without additional protocol by an ordinary boot request—it is of no concern to the sender that the receiver is another boot server that is writing the file on a disk rather than a machine that intends to immediately execute it. If a

boot server hears of a file with an older date, it sends a copy of its boot directory to the other server to trigger it to update immediately. If a user program such as the NetExec happens to be building a directory while a new version is propagating, it may hear about different versions from different servers. It should enter the latest version in its directory.

A host releases a new version of a boot file by briefly becoming a boot server. Broadcasting a directory listing a newer version of a file will cause another server to request a copy. Once a copy has been injected into the system, it will propagate without further intervention from the releasing host. A server that is down when the new version is released will discover it and update its copy when it restarts. Servers can be added to and removed from the system to adjust to changing loads. A new server requires no wired-in state, except, perhaps, a list of the boot files that it is supposed to maintain local copies of. When started, it locates other servers, requests copies of any files it wishes to keep, and perhaps tells the others of some new files only it has copies of.

Consider what happens if a boot file somehow becomes corrupted. Any machine that boots from that copy will crash. Any server that updates from it will spread the bad copy. If the creation date has been corrupted in such a way as to increase it, then all boot servers will update from this corrupted copy. Injecting a good copy into the system will not fix the problem, because it is "older" then the bad copy which will simply overwrite it. Therefore boot servers consider a file that appears to be created in the future as having a creation date of zero. This makes it eligible for update by any other copy and prevents its spread. Boot servers that hear of files with suspicious dates ignore them. Boot files also should, but currently do not, contain checksums.

3.2.4. Problems with the present protocol

This protocol has a problem similar to the name protocol of section 3.1. Boot files move from network to network through boot servers that have direct connections to two or more networks. Before directed broadcasts were implemented, the Pup internet got around this problem by locating boot servers in all gateways, but this is an unreasonable restriction. Boot servers need to be able to locate other servers that are not on directly-connected networks.

There is another impediment to file propagation, the hidden file problem; see figure 3.2. The boot servers on network ***A*** host ***a1*** and network ***B*** host ***b4*** keep copies of files 1,2, and 3; the boot server in the gateway (which has two addresses, network ***A*** host ***a4*** and network ***B*** host ***b1***) only keeps copies of files 1 and 3. Suppose that a new version of file 2 is released by the boot server in host ***a1*** on network ***A***. It broadcasts its directory listing the new version and the boot server in the gateway hears it. However that server does not keep file 2, so it takes no action. The server in network ***B*** host ***b4*** will never hear about the new version.

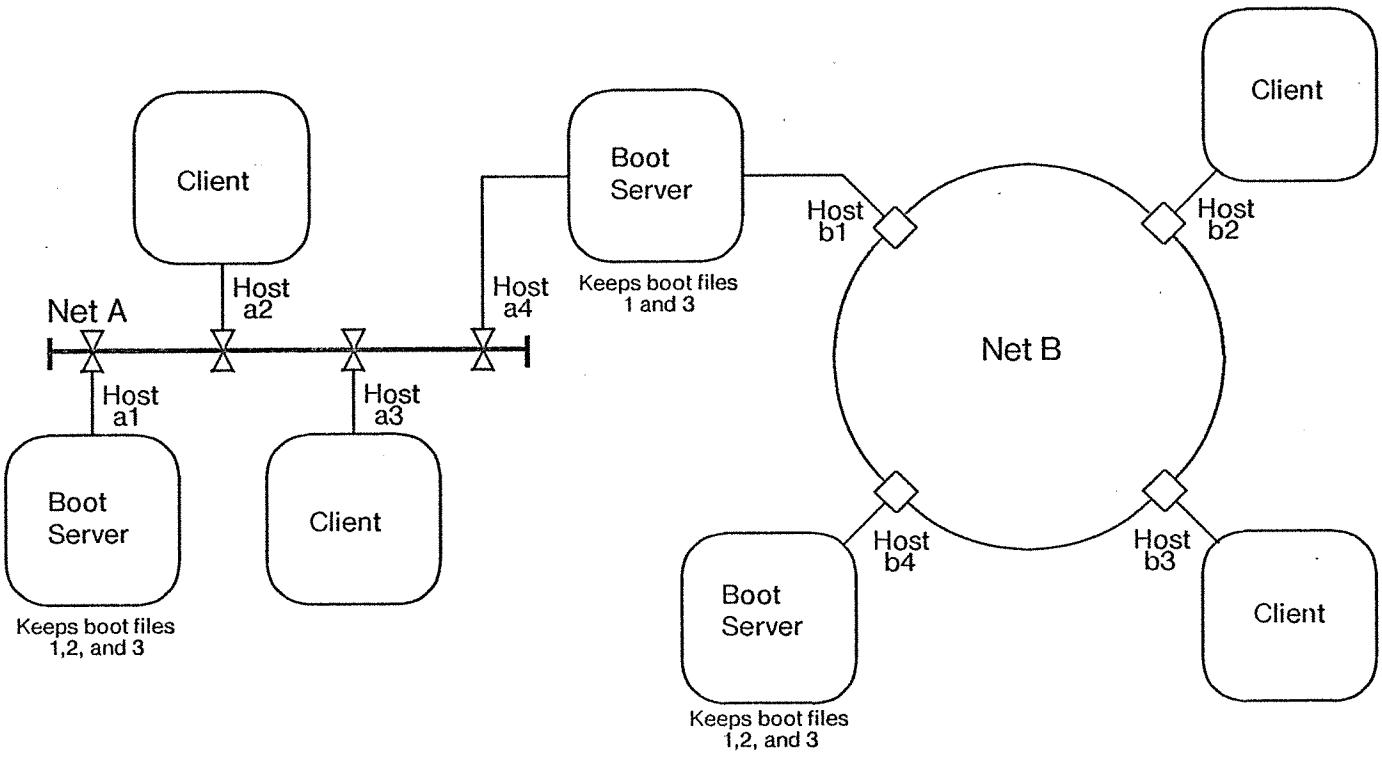

Figure 3.2. Boot server example.

One solution is to require the server in the gateway to remember and pass on the existence of the file and the address of the server from which it is available, though it need not keep the file. Still, this approach would require every server to keep directory entries for every boot file in the internet, and this could easily become a burden.

Another solution is for a boot server to periodically probe beyond its directly-connected networks. It can do this by broadcasting its boot directory on all networks up to n hops away, for some value of n. In this example, an n of 1 would propagate the new version in spite of the gateway. This is a directed broadcast, which is discussed in chapter 5.

3.3. Maintaining the date and time

The Pup time protocol exemplifies the simple connectionless protocol: a packet from client to server requesting the time, and a packet in the opposite direction with the response. The server does not have to remember anything after responding, and the client can recover from lost packets by retransmitting the request.

This time system is a calendar clock for use by humans. It is *not* suitable as a time base for a distributed system whose proper operation requires correct ordering of events that happen on several machines. At present few of our systems depend on an accurate global clock, but rather the time is displayed to a human who quickly notices if it is wrong. It is unrealistic to design a system that depends on a perfect global real time clock. A virtual clock not tied to real time is more useful [Reed, 1978; Lamport, 1978].

In discussing clocks, it will be useful to make a distinction between *precision* and *accuracy*. *Precision* refers to the size of the smallest unit of time with which we are concerned. *Accuracy* refers to the quality of the time: the amount of error present in a statement of what time it is. The precision of the time maintained by this protocol is 1 second; the accuracy is "a few seconds". The ultimate client of this grade of time service is a human being. Applications include file time-stamps ("did I recompile that module after editing it?"), display of a calendar clock ("time to go to the seminar"), and other similar uses.

3.3.1. Getting the time

The protocol for getting the time is similar to that for converting a name into an address or building a list of boot files: broadcast a time request Pup on all directly-connected networks; often several time servers will reply. It is the requestor's responsibility to sort out any disagreement; most settle for the first answer.

Pup type: time request
Pup contents: none

Pup type: time reply
Pup contents: date and time

3.3.2. Problems with the present protocol

The present time system is rather primitive. Time servers often live in machines that run for several months at a stretch without restarting. Even if the server's administrator has gone to the trouble of measuring the clock's drift and specifying a correction, in the current implementation, a clock can still drift by as much as 30 seconds per month. Many time servers do not correct their clocks at all.

The situation can be even worse. Assume there are two time servers on a network: *A* with an accurate clock, and *B* with a poor one. If *A* stops and restarts, it will set its clock from *B* during initialization. So *A* will carefully maintain *B*'s inaccurate time. The Pup internet currently has about 50 time servers; in any given 24 hour period about 10 of the host machines in which time servers live are restarted. The result is that time servers in the internet often disagree by several minutes.

The current solution to this problem is to run a program that locates all time servers and commands them to reset from a single source of accurate time. This program broadcasts a time request on each network listed in its local routing table to locate the servers, and then it sends separately-addressed reset commands to each one. In theory this program should be run periodically; in practice it is run when someone reports that the time in their machine seems to disagree markedly with that given out by the phone company's time lady.

3.3.3. A better time protocol

To achieve a better time service, a server machine might keep, in addition to the time itself, the absolute value of the worst-case error of its time. An improved time server would send its absolute error along with its time ("the time is 17:05:53, and I could be wrong by as much as 3 seconds"). The basic shape of the time protocol is unchanged. No new command/response packets are needed; the new protocol merely uses the new error field included in time reply packets.

Each machine also has two error thresholds. These thresholds are determined locally, and control the behavior of the local time protocol implementation. When the absolute error exceeds the *reset threshold*, the time module starts searching for a time server with better time (*i.e.* with smaller absolute error). It continues to search until it finds a time with an error smaller than its *acceptance threshold*, and resets its clock from that server. For a random personal computer, typical values for these thresholds might be ten seconds for the reset threshold and five seconds for the acceptance threshold; for a time server, they might be tighter. The difference between the thresholds influences how often the clock will be reset (and therefore the amount of network bandwidth and compute time the time service module will consume).

As a guard against not finding any time with acceptable error, a search can remember the server with the lowest error encountered during the search. As a last resort the clock can be reset from that machine if the error is less than that of the local clock; otherwise it is better to stick with the original time. If that server's error is below the reset threshold, another search will not happen until the error creeps back up to it. If the best time found is above the reset threshold, then the time module should go into a mode where it searches periodically, say once an hour.

The details of a time module's search for a time server are nearly independent of the rest of the time protocol. One search algorithm would be to broadcast a time request on each network zero hops away (*i.e.* directly connected), then one hop away, then two... until an acceptable response arrives. If the routing protocol maintains a delay metric as well as a hop metric, then the networks can be searched in increasing order of delay. This is better than using hops because the time received from a distant server must be corrected for the propagation delay from server to requestor, and the servers's total error must be increased by the error in estimating this delay.

The error in estimating propagation delay bounds the accuracy attainable with this scheme. Local-area networks have propagation delays on the order of a few milliseconds; long distance terrestrial links and satellite links have delays on the order of a few hundred milliseconds. By examining the route taken by the request and response Pups, it may be possible to justify a smaller error estimate [Mills, 1981]. On the other hand, the time a Pup spends inside a computer (requestor, intermediate gateways, or time server) often varies more than the propagation delays of the links traversed.

Using the improved time protocol, time servers with poor clocks and loose thresholds will not affect nearby servers as severely as in the current system. If a server's clock has more error than others nearby, it will simply be ignored. But if it is the only time available, it will be used.

3.4. Diagnostic protocols

These protocols illustrate some specialized uses of the Ethernet broadcast mechanism which border on multicasting.

3.4.1. DMT/Peek

The basic machine diagnostic for Alto computers is named *DMT*. The Alto Executive invokes DMT whenever it has been idle for more than 10 minutes, and whenever it notices that the Alto's local disk has been switched off. DMT's primary mission is to test the Alto's main and control memories. There are approximately 2000 machines scattered around the world sitting idle and hence running DMT an average of 12 hours a day. DMT also monitors the Ethernet and responds to some requests described in the following subsections.

Peek is an Alto maintenance program. Each evening, before going home, the technicians who fix Altos start up Peek on some machine. Periodically, DMT gathers up the memory error statistics it accumulates and sends them through the Ethernet to Peek, which records them in a disk file. When the technicians arrive in the morning, they stop Peek and print the file, which lists each machine by Ethernet address and identifies failing chips by board number and chip location. By the time most Alto users arrive at work, any bad chips have been changed. Peek also contains a boot server. This server usually keeps diagnostic programs and ignores utility programs, games, and other programs kept by public boot servers.

DMT can not assume the disk is accessible, so there is no place for it to keep long-term state. Peek can run on a different machine each night, and it is possible for Peek to move from one machine to another while some machines are running DMT. Some networks span two or more organizations, each with its own maintenance group. So how does DMT know where to send its error reports?

In effect, DMT broadcasts its reports and assumes that some other host(s) on the local Ethernet will collect them. If there happen to be two machines running Peek, owned by separate maintenance organizations, they both copy error report packets. The technicians simply ignore errors in machines they are not responsible for. This scheme is very simple to implement.

Each machine running DMT generates an error report every two hours. If a bad chip is detected, an error report packet is generated immediately (in case the machine is about to crash), and then the interval between reports doubles each report until it is back to once every two hours. A hundred machines running DMT on one Ethernet (not an uncommon situation) could broadcast a substantial number of error report packets. To prevent this traffic from burdening other hosts, DMT and Peek use a reserved Ethernet address. Peek runs special Ethernet microcode that will accept packets to the Peeking Alto's normal Ethernet address, the standard broadcast address, and the special DMT/Peek address. The use of this special address amounts to a multicast mechanism. Any host that wants to listen in on DMT error

traffic need only set up its Ethernet interface to accept packets to that address. The boot servers of section 3.2 use a reserved address in much same way to distribute boot loaders.

3.4.2. Idle request

One hundred Altos interconnected by a three megabit broadcast network can be viewed as a powerful multiprocessor [Metcalfe & Boggs, 1976]. When a machine is running DMT, it is idle and available for use as a component processor in someone's multiprocessor [Shoch & Hupp, 1980]. To locate idle machines, one sends an Idle Request Pup:

> *Pup type:* idle request
> *Pup contents:* none,

and DMT responds with:

> *Pup type:* idle reply
> *Pup contents:* machine configuration description.

The machine configuration description consists of the machine type, microcode version, and other things that help a prospective user to decide whether the machine is suitable for his purposes.

There are several Ethernets currently in existence with over 150 machines each. If one broadcasts an Idle request on such a network, implementation considerations make it unlikely that 150 replies wil come back. At least two phenomena are likely to reduce the catch. A massive collision will develop on the Ethernet; as it sorts itself out, the reply packets will arrive in a clump, usually separated by the minimum inter-packet spacing. Most Ethernet receivers cannot handle such *back-to-back packets*; they do well to receive every other packet in such a situation. The ordering of the packets on the Ether is random, so repeating the broadcast several times and taking the union of the responses would yield a reply from most hosts.

The other reason why one probably would not receive 150 replies is lack of buffers. Assuming one's receiver hardware can handle back-to-back packets, one must pass 150 buffers underneath the Ethernet receiver in rapid succession. Most hosts can not afford 150 buffers, and it is unlikely that one could process all of the replies in real time. If the broadcast packet came from a remote network, a slug of 150 replies would move back through the internet, shrinking as it scraped against obstructions (*e.g.* overloaded links and gateways) along the way.

Both of these problems can be eliminated by spreading the replies out in time. One way to do this is to specify as part of the protocol that if the request packet is broadcast, a host should wait an amount of time proportional to its address before generating a reply.

Polling each host separately rather than broadcasting might appear to be better in this situation. But polling would generate twice as many packets (n requests and n replies for polling versus *one* request and n replies for broadcasting), and it can be difficult to know

which addresses to poll—only a few of the 2↑47 possible addresses on a (10 megabit) Ethernet will correspond to real hosts.

3.4.3. Kiss of death

If DMT replies to an Idle request and the machine is suitable to be used as a component of a multiprocessor, one can ask DMT to commit suicide, and as its last act, boot load another program from the Ethernet:

> *Pup type:* Kiss of death
> *Pup contents:* boot file number and boot server address.

DMT requests the specified boot file from the specified boot server using the boot protocol of section 3.2. If the boot server address is zero, then the request is broadcast, and public boot servers may respond. Usually though, the boot file specified is a private program not kept by public boot servers. In this case, one run one's own stripped-down boot server that only knows how to boot that particular program. It need not respond to boot directory requests—that way curious users can not discover its existence, and cause confusion by running the private program.

3.5. Summary

Broadcasting and unicasting are different forms of interprocess communication. The ability to broadcast a packet opens up new approaches for the design of distributed computations. Things that are easy to do with broadcast packets are hard to do given only unicast packets. Various hosts in the Arpanet have implemented time servers, but programs that use them have to wire in the addresses; this is always distasteful. The Arpanet never has developed a name server; after many years, the most that has ever happened is that there is a text file on a particular host which contains the "official" mapping from names to addresses. Lack of a broadcast mechanism is the primary factor contributing to this lamentable state of affairs.

3.5.1. When to broadcast

You broadcast a packet when you don't know where specifically to send it. There are two major categories of situations in which you don't know where to send a packet: when you are trying to find some information, and when you are trying to distribute some information. In the former case you don't know who might have the information you seek, and in the latter case you don't know who might seek the information you have.

When trying to locate something, you should try to arrange the protocol so that a single broadcast gets you off the ground so that thereafter you can use unicast packets. We have seen two good examples of this. In the boot protocol, a broadcast packet is used to find the boot servers and build a directory of boot files. A request to boot a particular program can then be specifically addressed to a boot server. Converting a name to an address is the first step in many protocols. A broadcast name lookup request returns the address of the desired resource, and the transaction with that resource can then proceed using unicast packets.

When distributing information, its timeliness and importance should influence how often it is broadcast. A routing information server broadcasts twice a minute, and immediately on detecting a significant change; current routing information is vital to the operation of the internet. Name and boot servers broadcast their file versions once an hour and immediately after getting a new version.

3.5.2. The cost of broadcasting

The urge to broadcast a packet must be tempered by consideration of the cost of broadcasting. This cost has two components: the cost to the network of delivering a copy of the packet to all hosts, and the cost to the hosts of looking at the packet, and in most cases discarding it. (Routing information packets are useful to everyone—nobody discards them—so this cost component is zero for the routing protocol.)

The Pup internet does not charge its users for packets; it assumes its clients are well behaved and will not generate excessive broadcast traffic. In the real world, of course, accounting must be done; a broadcast packet in a commercial network will almost certainly cost more than a unicast packet and this will restrict their use.

3.5.3. Broadcast pitfalls

To use a radio analogy, a broadcast facility amplifies a packet. Broadcast systems must be careful about the gain and phase of feedback to prevent oscillation. In this case, oscillation is analogous to an endless exchange of packets. The worst kind is an endless exchange of broadcast packets, since these consume resources in all hosts, not just in the unfortunate ones that are oscillating.

Many hosts can not handle large numbers of packets converging on them at once. Most of the time a broadcast packet is interesting to only a few hosts on a network so only a few responses come back; there are only a few name boot or time servers per network. When a broadcast is answered by most hosts on a network, broadcasts must be used with care and some protocol on top of the bare broadcast mechanism may be needed.

3.5.4. Similarities among the protocols

The route, name, boot, and time broadcast applications are distributed computations. The routing protocol computes the minimum number of hops from each network to each other network. The name update protocol propagates the directory with the maximum version number, wherever it may be, to all other name servers. For each boot file, the boot update protocol finds the copy with the largest creation date and propagates it to all servers. The improved time protocol minimizes (within the tolerance of individual servers' thresholds) the absolute error of time kept by the servers.

Routing tables, name databases and boot files are distributed databases. Because of the amplifier effect of broadcasting, incorrect information can spread quickly. Systems with distributed databases must be designed to minimize the local consequences of incorrect information and to limit its spread. A routing server can checksum its table and code, a name server can checksum the directory, and a boot server can reject files with unreasonable dates.

The information maintained by the route and time protocols is position-relative. The contents of a routing table and the error of a clock depend on the location of the host and the characteristics of surrounding components of the internet. The information maintained by the boot and name protocols is absolute: all sites have the same information. These latter protocols are much easier to design and think about.

The route, name, boot and time servers act as specialized, high-level, store-and-forward switches. Consider a routing server: it updates its table from an incoming routing information packet and then forwards the updated table to other nearby routing servers by broadcasting. Routing information moves from one network to another by passing through routing servers.

3.5.5. Multicasting

Of the broadcast protocols presented, distribution of routing information is the only one in which all hosts are interested in the contents of a broadcast packet. In all other cases where a broadcast was used, a multicast would be better. When a host broadcasts a name lookup request, it is in effect saying: "if you are a name server, then please convert this name into an address, *otherwise discard*". If a multicast mechanism were present, the packet could be sent to the (well-known) name server multicast address and it would be received only by name servers.

3.5.6. Broadcast streams

All of the broadcast protocols presented here are connectionless—each broadcast packet is independent and no state is maintained by the sender or any of the receivers. What about a broadcast stream? A human analogy would be a teacher instructing a class of students. Each person (student and teacher) is analogous to a host, and the classroom is a network. The teacher broadcasts a stream of information, which is received by all students. Some student will inevitably not hear some of the teachers words. When this happens, the student raises his hand and asks the teacher to repeat. In a broadcast computer stream, the request for retransmission and the retransmitted information need not be broadcast—each can be unicast. If many hosts asked for the same packet to be retransmitted, a clever implementation might broadcast it.

However, it would be a rare computer network that had only one broadcast stream. Returning to the teaching example, a more common situation would be several teachers in the same room all talking at once with subsets of the students interested in what each teacher was saying. The burden on the students to filter out words from unwanted teachers would be severe, and one teacher's words might drown out another's causing a high rate of requests to repeat. A good computer solution to this would be to use multicasting. Each stream source would send to a separate multicast address, and an interested receiver would set up its network interface to receive packets to the multicast address corresponding to the stream in which it was interested. (Of course, a computer might be interested in several streams—multiprocessing is easy for computers but hard for humans.) Lacking a multicast mechanism, broadcast streams are impractical except in special circumstances.

Chapter 4

How to broadcast

> "We beam it to Phobos; they can it for Mars and also put it on the high circuit for New Batavia [in Luna], where the Earth nets will pick it up and where it will be relayed for Venus, Ganymede, et cetera. Inside of four hours it will be all over the system..."
>
> *Double Star*
> © 1956 Robert Heinlein

The purpose of this chapter is show that it is *easy* to broadcast in all common types of networks.

The first section explores a number of issues which must be confronted when designing a broadcast mechanism for any network. Some basic concepts are laid out, and some ways to classify networks, which illuminate their similarities and differences, are presented. One of the most important points is that, as a natural extension of the Pup attitude toward reliability, broadcasts need not be reliable. If you accept this, everything becomes tractable.

The second section examines a number of current networks, and asks the question: "is a broadcast addressing mode available, and if not how might it be added?" The list of networks examined is not exhaustive, but I have tried to include at least one example of each type of network which is in widespread use. The point is not to suggest that people should rush out and retrofit their networks; this is rarely feasible. But I do hope that all networks designed from now on include a broadcast addressing mode.

Broadcasting in a store-and-forward network is, admittedly, more difficult than in other types. The problem has received a lot of attention, both theoretical, and practical; many schemes have been proposed and some implemented. Also, an internet is an intrinsically store-and-forward network, so broadcast techniques for this type of network are central to the thesis. For these reasons, the last section of this chapter focuses on how to broadcast in a store-and-forward network.

4.1. Broadcast design issues

4.1.1. Logical and physical connectivity

Logical connectivity characterizes what set of hosts in a network can be reached with a single packet. Physical connectivity characterizes the communication channels of a network. This chapter focuses on how to create broadcast logical connectivity out of whatever physical connectivity a network possesses.

Another name for logical connectivity is *addressing modes.* They form a spectrum from unicast through multicast to broadcast.

> **Unicast** logical connectivity means that a host can send a packet to any other single host in the network. This is the minimum addressing mode implemented by all networks that we will consider.
>
> **Multicast** logical connectivity means that a host can send a packet to a subset of hosts in the network. Note that multicast is the general case, and that unicast and broadcast are special cases in which the subset addressed is *one* or *all* hosts.
>
> **Broadcast** logical connectivity means that a host can send a packet to all hosts in the network. *All* includes the sending host too. For implementation reasons, some network hardware may not be able to receive its own packets. This shortcomming should be repaired in the network driver software.

Physical connectivity also forms a spectrum, from point-to-point through multipoint to broadcast channels.

> **Point-to-point** physical connectivity means that a channel has exactly two hosts attached to it. A ring network is a good example of a network that uses point to point channels.
>
> **Multipoint** physical connectivity means that a channel has more than two hosts attached to it. A packet radio network is a good example of a network that uses multipoint channels.
>
> **Broadcast** physical connectivity means that a single channel has all hosts attached to it. An Ethernet is a good example of a network that uses a broadcast channel.

To achieve broadcast logical connectivity, one must distribute the packet to every host and mark it in such a way that each host will make a copy. If the network has broadcast physical connectivity, the first requirement is taken care of, and all that remains is to set aside a broadcast logical address that every host will accept in addition to its physical address. When the physical connectivity of the network is point-to-point or multipoint, distribution of the packet requires extra work.

4.1.2. Forwarding and filtering

The two principal operations of packet switching are forwarding and filtering. Forwarding is the act of transporting a packet to its intended recipient(s). Filtering is the act of deciding whether or not to copy a packet as it passes by. Forwarding is most strongly affected by the physical connectivity of the network, and filtering is most strongly affected by the logical connectivity.

At this point it is useful to distinguish between a host and a packet switch. Sometimes the host and the switch are tightly integrated so that the distinction is more logical than physical, and sometimes the host and switch are distinct physical objects. Figure 4.1 shows a range of switch configurations which illustrate the forwarding and filtering functions. The complexity of the switch increases from left to right.

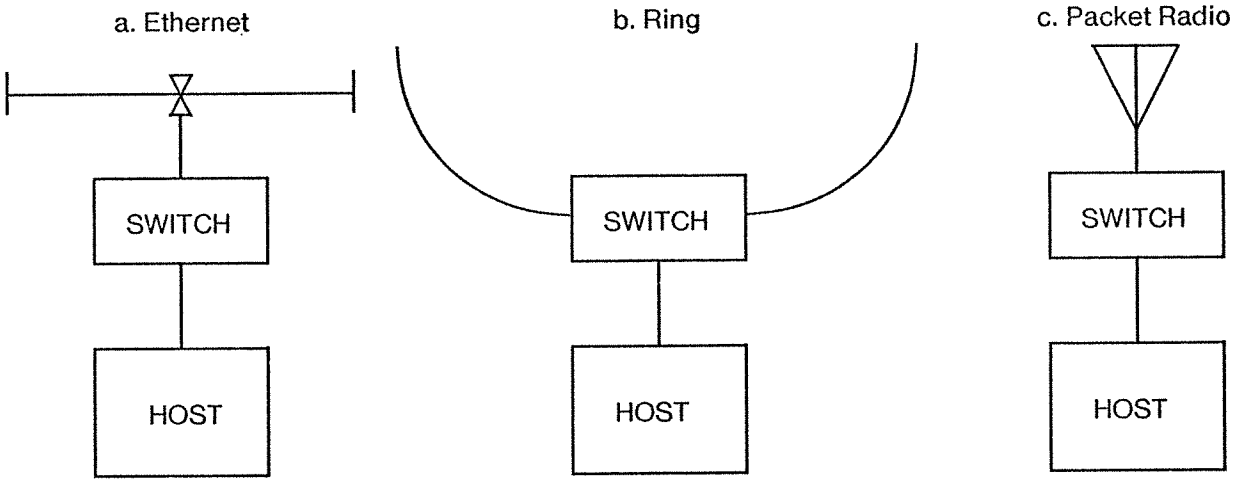

Figure 4.1. Packet switches of three common networks.

Figure 4.1(a) shows a host connected to an Ethernet. For unicast logical connectivity, the filter function must recognize the switch's address in a passing packet and make a copy for the attached host. For broadcast logical connectivity, the filter must recognize a distingushed logical address ("all hosts") as well as its physical address. Multicast logical connectivity requires an associative match against a list of addresses, one physical and the rest logical. Since the physical channel has broadcast connectivity, the switch does not have to do any forwarding. The switch is about as complex as a disk controller and is usually implemented in the same fashion. The connection to the physical channel, called a *transceiver*, is a simple level converter and contains no logic.

Figure 4.1(b) shows a host connected to a ring network. Address filtering is the same as for an Ethernet. The forwarding function consists of deciding whether or not to repeat an incoming packet on the outgoing channel. In most ring networks, a packet goes all the way around once and is removed by its sender. It is possible to be clever and remove a packet sooner: for a unicast packet, the only recipient switch can remove it; for a multicast packet, the last recipient switch can remove it; for a broadcast packet, the switch just upstream from the sender can remove it. Ring networks must worry about removing packets whose addresses have been corrupted, lest they not be recognized by any switch and circulate endlessly. The forwarding function is often located in a *repeater*, an external box to which the physical channels connect, while the address filter and the rest of the controller functions are located inside the host in the same manner as an Ethernet controller.

Figure 4.1(c) shows a host connected to a packet radio network. Arriving packets are briefly stored and then forwarded to another nearby switch that is closer to the packet's destination. Packets that arrive at a switch are not necessarily destined for the attached host; the address filter may reject the packet but it may still be forwarded. Here the forwarding function varies with the addressing mode: a unicast packet should be forwarded along those branches of a shortest path tree connecting the source and destination switches and whose root is the source switch; a broadcast packet should be forwarded along all of the branches of a shortest path tree rooted at the source; a multicast packet should be forwarded along only those branches of the shortest path tree rooted at the source and that pass through switches in the multicast group (a less efficient way is to use the broadcast forwarding function and rely on the address filters of switches in the multicast group to make copies). These forwarding functions are much more complex than for a ring and depend on data structures that should track changes in the physical connectivity between switches. Therefore forwarding is usually performed by software rather than hardware and the switch is often a separate computer from the host.

Being able to disable the address filter and receive all packets is sometimes useful. A host operating in this mode is said to be *promiscuous* [Metcalfe & Boggs, 1976]. This is an escape mechanism for those situations in which the standard filter is too restrictive. Disabling the filter in the switch allows the host to implement an arbitrarily complex filter of its own design. If packets can arrive faster than the host can filter them, then operating in promiscuous mode will result in more lost packets. Promiscuous mode is most useful when the design of the network is such that every switch sees every packet.

4.1.3. Reliability

The issue of reliability has been a major stumbling block to previous attempts at providing a broadcast addressing mode. A network is *not* the only place where packets are lost. It is a waste of effort to make transport of a packet through a network perfectly reliable when there

are so many other places a packet can be lost. A packet may be discarded by the source host due to resource limitations before it even gets to the network. If a packet makes it into a network, all sorts of bad things can happen to it, some of which not even a network's designer can anticipate. A packet may make it into the destination host and then be discarded before it can be delivered to the end process. The end process may even be forced to discard it without acting on it.

Some networks provide an acknowledgement mechanism. A packet on most ring networks has some bits that indicate whether any address filter recognized the destination address and whether any host copied the packet. Many store-and-forward networks return an *ack* to the sender indicating that a packet reached the destination switch or was delivered to the destination host. Most networks built on broadcast channels do not do either. Network-level acks for broadcast packets are usually hard to provide in any type of network and this is used as an argument against a broadcast addressing mode ("what use is a broadcast packet if you can't tell who got it?"). Note that network-level acks merely indicate that a packet made it to the host's front door; they do not say that a packet made it to an end process.

When asked to broadcast a packet, a network should do its best to deliver a copy to each host. Occasionally some copies may arrive damaged, or not arrive at some hosts at all. It is sufficient that *most* hosts receive each broadcast packet and that *no* host repeatably misses the same packet broadcast by the same host due to some artifact of the network. Acks are not necessary. Such an "unreliable" broadcast mechanism satisfies most users most of the time. In the unsatisfied cases, a higher-level protocol tailored to the particular situation can be added by the client, just as with unicast packets.

4.1.4. Cost

The added cost of implementing a broadcast mechanism must be justified by the added benefits. In my experience, broadcast packets constitute a small fraction of the total packets transported by a network, but if they are available *every* host uses broadcasts at least occasionally. It is unreasonable for a network designer to insist that the cost of the added mechanism be proportional to the number of broadcast packets. A broadcast capability makes some old things easier and some new things possible. On the other hand, the cost of the extra mechanism should not dominate the cost of providing a unicast capability.

In Ethernets and rings, a broadcast packet does not consume any more channel bandwidth than a unicast packet. The switch must keep another piece of state, the broadcast address, and the filter must be augmented to accept packets with this destination address. These additional costs are small and easy to justify.

It is store-and-forward networks where the cost of a broadcast addressing mode has seemed too high to justify. I suspect that the reason is because the proposed broadcast facilities were trying to do too much. Not attempting reliable delivery lowers tremendously

the cost of a broadcast. Since a broadcast packet traverses more physical channels than a unicast packet, broadcasts consume more channel bandwidth. The forwarding function is more complex, but not tremendously so, as we shall see in section 4.3. Some broadcast forwarding algorithms require auxiliary state, but some use just the switch's existing routing table.

Finally, a word about prices. If a broadcast packet costs a network more to transport than a unicast, it is reasonable for the network to charge a higher price for the service. The price can be used to control the fraction of packets that are broadcast. With a properly chosen price, those who truely require a broadcast will be able to do it, but capricous use will be discouraged.

4.2. Some real networks

In this section, we examine the addressing modes offered by current implementations of three network types: broadcast networks, ring networks, and store-and-forward networks. The object is to catalog addressing modes beyond basic unicasting, and to answer the question: "is a broadcast mode available, and if not, how might it be added?"

4.2.1. Broadcast networks

4.2.1.1. Ethernet

There are two versions of the Ethernet, the *experimental Ethernet*, developed by Xerox research in the early 70's [Metcalfe & Boggs, 1976], and the *standard Ethernet*, whose specifications were developed and published jointly by Xerox, DEC and Intel in 1980 [Xerox *et al.*, 1980]. The standard Ethernet differs from an experimental Ethernet in details but not in overall philosophy. Two detail changes are relevant to this discussion: the size of addresses and the addition of a *multicast* bit [Dalal & Printis, 1981].

Addresses grew from eight bits in the experimental Ethernet to 48 bits in the standard Ethernet. Addresses on the experimental Ethernet are *network relative.* Imagine a machine with host address ***a*** on network ***A***. For some reason, it is moved to network ***B***, where host address ***a*** is already in use. The person who moves the machine must realize that address ***a*** is in use and pick a new address, ***b*** (and change any high level name to address mappings). If this isn't done, two machines will have the same address, and they will step on each others toes, causing great confusion. This problem is eliminated in the standard Ethernet by making addresses *absolute:* an address is unique in space and time; every host on every Ethernet has a different address. For this to be possible, the address space must be large, thus 48 bits were used.

One of the 48 address bits is used as a physical/logical, or multicast bit. If this bit is zero, the other 47 bits identify one and only one physical host anywhere on any Ethernet. If the bit is one, the other 47 bits identify a *multicast group:* a group of hosts whose address filters will accept packets with this logical address as well as packets with their individual physical addresses. The broadcast address is just one of these logical addresses—the address with all 48 bits set to one.

The experimental Ethernet interface designed for the SUN workstation handles multicasting by brute force [Bechtolsheim & Baskett]. The receiver's address filter contains a 256-bit memory, one bit for each possible address on an experimental Ethernet. The filter indexes into this memory using an incoming packet's destination address. If the memory bit is set, the packet is copied, otherwise it is ignored. During initialization, bit zero (the broadcast address in the experimental Ethernet) and the bit corresponding to the interface's physical

address are set, causing the filter to act as a normal Ethernet address filter; to be promiscuous, all of the bits are set. Unfortunately, this technique does not scale to a 48-bit address; some more clever encoding of the table is required.

Multicasting in the standard Ethernet is not fully developed; it is clear that the address space needs to be split into physical and logical addresses, thus the multicast bit, but the strategies for using this new capability have not solidified. Unicasting and broadcasting are in everyday use on both versions of the Ethernet, and many multicast techniques have been tried on the experimental Ethernet.

4.2.1.2. Arpa wideband packet satellite network

The Arpa wideband packet satellite network is composed of a broadcast satellite channel and a number of PSATs, *i.e.* Pluribus Satellite IMPs (IMP = packet switch) [Falk & Koolish, 1980]. Many hosts can attach to a PSAT, some may be colocated with it and others may be at the end of a long-haul link such as a phone line. Splitting out the packet switch into a separate computer gives the network interface some of the flavor of a store-and-forward network with separate switches, as in the Arpanet. However the physical channel interconnecting PSATs is broadcast, which is not at all like the point-to-point channels interconnecting Arpanet IMPs.

Each host is assigned a 16-bit *permanent address* which is independent of the physical port on the PSAT to which it is connected; this is similar to an absolute physical address in the standard Ethernet. Each PSAT maintains a list that maps permanent addresses to host ports. It can deduce the host connected to a port by monitoring the source addresses of packets arriving through the the port. This implies that hosts must remember their permanent addresses in stable storage.

A host may also be a member of a *multicast group.* A host creates a multicast group by interacting with a virtual service host within the network, which assigns an unused address (*i.e.* one which is neither a permanent address nor a currently assigned group address). The host then communicates this address to the other hosts that should be in the group. They in turn interact with the virtual service host to register themselves as members of that group. When a packet arrives on the satellite down-link, each PSAT looks up the destination address in a table to determine which host ports should get copies. If the address is permanent, all but one PSAT will find no entry and discard the packet, and the remaining PSAT will find an entry listing a single host port. If the address is a multicast group some of the PSATs will deliver copies to some of their host ports.

There is no difference between a permanent (unicast) address and a group (multicast) address except that a group address may point at several host ports within a PSAT and may exist in the tables of several PSATs. The wideband satellite network's addressing mechanism implements general multicast, and is therefore a perfect illustration of how a broadcast (and a

unicast) are just special cases. To implement a broadcast facility, some host simply sets up a group and tells every other host in the network to join it.

4.2.2. Rings

4.2.2.1. Cambridge ring

The Cambridge ring is very popular in England and is in commercial production [Wilkes & Wheeler, 1979].

Packets in the Cambridge ring are very short, about 40 bits (8 destination address, 8 source address, 16 data, plus some control bits). These "hardware" packets are referred to as *minipackets*; "software" packets, *i.e.* what most people think of as packets, are assembled by a receiving host out of some number of minipackets. Between software packets, a host's receiver is *source promiscuous:* it will accept a minipacket from anyone. Once an acceptable minipacket has arrived, it rejects minipackets from other sources until it has fully assembled the software packet. This procedure is not strictly necessary, as a very capable host can keep a table of pointers to software packets in the process of being assembled, and index into this table by the source address of an incoming minipacket to find out where to put the fragment.

The Cambridge ring only implements unicast logical addressing. A station on *one* of the networks at Cambridge has been modified for promiscuous filtering mode, but this is not a standard feature of the hardware. The minipacket design mentioned above makes it difficult to provide broadcast addressing unless all hosts implement the more complex packet assembly scheme. Any host that uses the simple reassembly scheme and that is in the middle of receiving a unicast packet when the first minipacket of a broadcast is transmitted will ignore the broadcast because the host is already committed to receiving the other packet. A broadcast addressing mode was not provided because the designers of the ring felt that the loss rate for broadcast packets would be too high to make them useful [Hopper, 1981].

4.2.2.2. U.C. Irvine ring (V.0 LNI)

The Irvine ring was designed in the early '70s by a group led by David Farber [1975]. The Arpa/MIT V.2 LNI, described below, is a direct descendant of it; one could call the Irvine ring the V.0 LNI. Only one Irvine ring was ever built; it had 3 hosts.

Addresses were 16 bits, and each station contained a 16-slot associative memory. An incoming packet's destination address was looked up in this memory, and if any entry matched, the packet was copied. This mechanism permitted full multicast. Broadcast was implemented by simply picking some address (*e.g.* zero) and entering it in the associative memories of all interfaces.

A distributed operating system called DCS was implemented on this hardware [Rowe, 1975], and entered process names in the associative memory. This permitted a process to move from one host to another transparently. The cost of the associative memory was high, and the limited number of slots made it necessary to treat them as scarce resources; cheap, numerous processes were not possible.

4.2.2.3. Arpa/MIT V.1 and V.2 LNI

In the mid '70s, a group at Irvine began to redesign the ring, with the intention that it serve as a local network for Arpa-sponsored research sites. This became known as the Arpa Local Net Interface (LNI), and later when it was redesigned again, as the version 1 LNI (V.1 LNI) [Clark *et al.*, 1978; Mockapetris *et al.*, 1977]. The V.1 LNI used 32-bit addresses, an 8-slot associative address memory, and program-settable masks to determine which bits of a packet's address and which bits of a filter address should actually be matched. A TTL-MSI implementation took over 350 chips (the intention was to quickly convert to LSI so the TTL design was not optimized); a few tens of copies of this version were used at UCLA and MIT.

In the late '70s, the ring design effort moved to the Laboratory for Computer Science at MIT. The version 2 Local Net Interface (V.2 LNI) shrunk addresses to 8 bits and omitted the associative address filter of the V.1 LNI. The basic addressing modes proven by the experimental Ethernet, namely unicast, broadcast and promiscuous mode, were adopted. This version of the design is in commercial production.

4.2.3. Store-and-forward networks

There are two important types of host in a store-and-forward network: those that are internal to the network and enjoy special privileges, and those that are external and do not. The internal hosts are usually just processes within the computer controlling the switch; the Arpanet calls them *fake hosts.* A typical internal host might maintain the routing table for the switch, collect accounting statistics, or permit remote debugging of the switch control program.

4.2.3.1. Arpanet

The physical links in the Arpanet are point-to-point. Some links are implemented on satellite channels, but no advantage is taken of the broadcast physical connectivity. Unicast addressing is the only mode available to external hosts. Internal hosts can know which channel packets arrived on, and can force their packets to depart on specified channels, circumventing the normal routing mechanism. With these two capabilities, enhanced addressing modes are possible (but only for the internal hosts); they are discussed in section 4.3.

A broadcast addressing mode has been talked about for years, but never implemented. At first this was because nobody could figure out how to do it cleanly and simply. This objection was demolished when Dalal invented the reverse path forwarding algorithm [Dalal, 1977]. Then the objection was that there was not enough code space in the Imps for the extra mechanism (plausible in some of the older models). However, I think part of the real reason is the basic incompatibility between the type of service the Arpanet strives for (flow controlled reliable datagrams) and what can be achieved with a broadcast mechanism. Broadcast flow control would be difficult since it would involve not just two IMPs coordinating their activity but all IMPs. High reliability broadcasting would require the network to dedicate many buffers for long periods of time and to generate many internal acknowledgement packets. The problems are further complicated by the Arpanet's policy of fragmenting and reassembling user packets. It would not be too hard to provide a broadcast addressing mode in which packet size was limited to a single internal packet (1K bits) and no host-level acknowledgement was provided by the network.

In the late 70's, the Arpanet routing algorithm was redesigned so that routing internal hosts sent routing data not just to their immediate neighbors, but to everyone, using a broadcast. Many people hoped that this meant that a broadcast facility was about to be made available to clients, since it had been implemented inside the network to support the new routing algorithm. This impression is mistaken; the reasons are discussed in section 4.3.

4.2.3.2. Arpa packet radio network

The physical channels in the Arpa packet radio network are multipoint [Kahn *et al.*, 1978]. Some of the sites where PRnets have been established have such good radio propagation properties that the channels turned out to be broadcast—to the consternation of the network designers, who had to reduce the radio transmitter power to test the forwarding function. When the channel is not broadcast, frequency reuse is possible: two or more packets can be in flight between switches on the same frequency. As with the Arpanet, there are internal hosts with special capabilities from which enhanced addressing modes can be constructed, but regular external hosts can only use unicast addressing.

Two special situations that amount to multicasting are worth mentioning. Before a packet radio can forward packets, it must register itself with a *station*, a central control point in the network. PRNets typically have one station, which is the source of all routing information (as development has proceeded, provision for operating without a station and with more than one station has increased). The station does not have a distinguished address, so there must be some mechanism for an unregistered PR to get its registration packet to a station without knowing what the station's address is or how to get there. To do this, it emits a packet with a special type, which all registered PRs will accept and forward to a station. This special registration packet type is in effect being used as a multicast address.

A packet radio monitors all traffic it can hear, even if it is not for itself. From this information, it builds a *neighbor table*, which looks remarkably like a conventional routing table. Notwithstanding this, routes are centrally maintained and distributed by the station, and packets contain hop-by-hop routes. Low-level acknowledgements are used within the radionet. When a PR fails to get an ack, it retransmits the packet with a bit set which will cause nearby PRs to forward it even though the packet is not addressed to them and they are not part of the specified route. That special bit acts as a sort of "physical multicast" address, overriding the normal hop-by-hop route.

4.2.4. Connectivity matrix

Figure 4.2 is a matrix with physical connectivity along one axis and logical connectivity along the other. A slot is filled with one of our example networks if it implements that combination of physical and logical connectivity. This figure summarizes the information in the previous sections.

		Logical Connectivity		
		Unicast	Multicast	Broadcast
Physical Connectivity	Point to Point	V.0, V.1, V.2 LNI Arpanet Cambridge ring	V.0, V.1 LNI	V.0, V.1, V.2 LNI
	Multipoint	PRNet		
	Broadcast	Ethernet Satnet	Satnet	Ethernet Satnet

Figure 4.2. Physical/logical connectivity matrix.

Since unicast logical connectivity is the minimum service level, all of the networks are somewhere in the left-most column. Note that many of the networks are also in the right-most column; the only ones missing are the Cambridge ring, the Arpanet and the Arpa packet radio network. The middle column, multicasting, is nearly empty.

Crossing the matrix in the opposite dimension, note that the top row is fairly crowded. This is because the technology for point-to-point links is old and well understood, reducing the risk in that part of a network design. The lowest row is sparsely populated, but this technology is new; communication satellites are less than 25 years old, and the Aloha network was only invented in the late '60s. The middle row, multipoint physical channels, are usually realized as radio channels. Electromagnetic spectrum space will always be a scarce resource and this row will never be very full.

4.3. Store-and-forward broadcast techniques

In this section, we examine techniques for achieving broadcast logical connectivity in store-and-foward networks, which folklore has dictated to be hard.

The discussion assumes the presence of a conventional routing table for unicast operation; for more on routing see McQuillan [1974]. Yogen Dalal did the fundamental work on broadcasting in a store-and-foward network [1977]. David Wall extended Dalal's broadcast techniques, and considered ways to adapt them to multicasting [1980]. Much of the material in this section is drawn from these two sources.

After some preliminary explanations, we get down to specific ways to broadcast (and sometimes multicast). The collection of techniques presented is not exhaustive, but most of the practical techniques are included, and also some impractical ones when their theoretical underpinnings shed light on the practical techniques.

4.3.1. The basic model of a packet switch

Figure 4.3 is a block diagram of a basic store-and-foward packet switch. The *switch* moves a packet among the channels and hosts, based on the packet's addresses and the information in the *routing table.* There are a number of communication channels that connect to other packet switches; these may be point-to-point or multipoint channels. There are a number of *hosts* connected to the switch; some of the hosts are *internal,* and some are *external.*

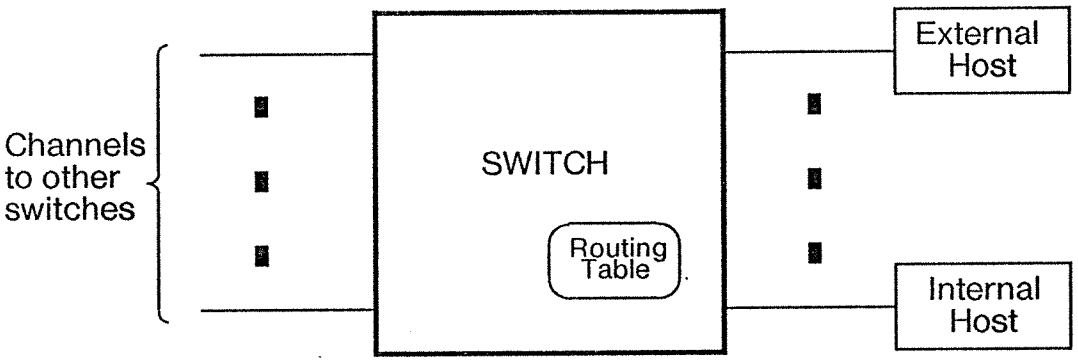

Figure 4.3. Block diagram of a basic packet switch.

The internal hosts enjoy special privileges that the external hosts do not. At a minimum, an internal host knows which channel or host a packet arrived from, and can force a packet it generates to depart on a specific channel or to a specific host. The most important internal host is the one that maintains the routing table.

4.3.2. Criteria for evaluating broadcast techniques

There are four key parameters by which to judge the suitability of a broadcast scheme in a store-and-foward network: bandwidth, delay, state, and computation. Theoretical discussions of broadcasting focus on the first two. The amount of state a packet switch must maintain, and the computational expense of providing a broadcast facility are of interest to implementors.

Bandwidth is referred to in theoretical discussions as *cost*, as in the *edge cost* of a graph. Store-and-foward networks typically span large distances, and given the economics of long-haul channels, bandwidth should not be squandered.

There are many ways to waste bandwidth. For most network configurations and broadcast techniques, it is not necessary to send a packet over every channel. It is almost always redundant to send a copy in both directions on a channel. It is also bad form to send the same packet down a channel more than once (this includes packets which differ only in their destination addresses).

Delay is the time interval from when a sending host hands a broadcast packet to its local switch, until the last copy is delivered to the last host. Because copies of a broadcast can be moving over many channels and through many switches in parallel, delay is a fundamentally different parameter than bandwidth. Techniques that minimize delay usually do not minimize bandwidth, and vice versa.

State is the extra information that must be maintained to implement a broadcast mechanism. Because broadcasting in a store-and-foward network necessarily entails creating multiple copies of a packet (the amplifier effect of section 3.5.3), some techniques may have to remember whether or not a broadcast packet has been seen before, so it can be discarded rather than infinitely re-amplified. An implementor must be careful to keep the need for this kind of state information bounded.

Another form of state is the instructions for the outgoing channels on which to route a broadcast packet. This may depend on which channel it arrived on, and which host (or switch) it originated from. The regular unicast routing table is this kind of state information. Most broadcast techniques require some degree of synchronization, or agreement, among the packet switches about this kind of state. It is unrealistic to assume that agreement will be perfect all the time. Algorithms should be designed so that state converges and so that newly started packet switches can rapidly aquire necessary state.

Computation by the packet switch while forwarding a broadcast packet should be minmized, as should the computation expended maintaining any extra broadcast-related state.

4.3.3. Techniques that do not scale

Two of the simplest broadcast techniques do not scale up to large networks, but deserve mention. One is eminently reasonable for networks up to a few tens of nodes, and the other is appropriate for networks of up to a few hundred nodes.

Multidestination addressing. Actually, this is a technique for full multicast, so it works for broadcast too. The idea is to include a list of addresses in the header of a packet. When such a packet arrives at a switch, multiple copies are created and the addresses of the incoming packet apportioned among them. More precisely, for each address in the incoming packet, the switch consults the normal unicast routing table, to find out which outgoing channel (or locally attached host) will get the packet closer to that address. It then creates a copy of the packet for that channel and copies the address into the new packet's header (or if it has already decided to send a packet down that channel because of a previously processed address, it merely copies the address). When all of the addresses in the incoming packet have been processed, the switch destroys the incoming packet and forwards the newly created copies.

There are two ways to put multiple addresses in the header of a packet: use fully specified addresses, or use some encoding, such as a bitmap. There are two drawbacks to using full addresses. First, the length of the header will be variable; this is a nuisance to all who must process it, increasing their computational load. The other disadvantage is that the header gets very long very quickly—especially in the broadcast case where every address must be listed.

A bitmap is usually a better choice because it is of fixed length: this makes the header fixed length and less expensive to process. The principal disadvantage is in networks with large addresses such as the Arpanet (24 bits): a bitmap would be 2 megabytes long! More clever encodings are possible, especially if the bitmap is normally sparse, but this gets back to variable length headers and more computation to interpret the encoding.

Multidestination addressing tends to minimize both delay and cost. There are also no problems with stability (creation of infinite copies) or even creation of *any* duplicates at all. When the Arpanet was young and its addresses were small (256 hosts), a simple bitmap would have worked very well.

Separately-addressed packets. The scaling problems of this technique are even more obvious. A host simply sends a copy of the broadcast packet to each possible destination; of course, this works for multicast too. This technique works quite well for small numbers of addresses. A special case of separately-addressed packets is discussed below under the name of *nearest-neighbor multicasting;* its generalization to an internet, in chapter 5, is surprising.

The main disadvantages of this technique are that it is wasteful of channel bandwidth, since several copies differing only in destination address are likely to transit the same channel; and it is wasteful of switch computation, since those virtually identical copies will flow through many of the same switches. There are also two minor drawbacks: it takes the source host

longer to generate the multiple copies and it requires extra computation to update the destination address between each transmission. But for up to a few tens of hosts, this technique may be superior to all others.

4.3.4. Forwarding along trees

The customary theoretical approach to the problem of broadcasting in a store-and-foward network is to consider it to be a graph, with the switches as vertices and the channels as edges. Some type of tree is superimposed on this graph and copies of a broadcast packet are forwarded along the branches of the tree. The principal disadvantage of such schemes is that it is often difficult to build the required trees in a distributed manner.

Consider now three classes of trees that have been extensively studied by Dalal and Wall. For simplicity of exposition, assume that each packet switch has only one host, that is, it suffices for a packet to reach every node (=switch).

Shortest-path trees. A shortest-path tree is one in which the path length from the root node to any other node is the shortest possible. Shortest-path trees are generally not unique, and there is usually a different one for each root node.

If each node maintains a copy of all of the shortest-path trees to all of the other nodes, a broadcast packet can be forwarded over the branches of the shortest-path tree whose root is the packet's source. More precisely, when a broadcast packet arrives, the switch uses the source address of the packet to find its copy of the shortest-path tree whose root is that address. It then delivers a copy to its locally attached host and forwards copies out any channels that are part of the tree.

This broadcast method is also called *source-based forwarding*, because the path is determined by the source address of the broadcast packet. It minimizes both bandwidth and delay. On the other hand, it requires a large amount of state information (proportional to the square of the number of nodes in the network), and substantial computational expense to maintain it.

Minimum spanning trees. A *spanning tree* is one which includes all switches (vertices) of the network (graph); the *cost* of a tree is the sum of the costs of all of its channels (edges). A *minimum spanning tree* is a spanning tree whose cost is as small as possible over all spanning trees. An MST is not necessarily unique, but all nodes must agree on *one* MST.

If all nodes maintain copies of a minimum spanning tree, then a broadcast packet forwarded over the branches of this tree will incur the least cost to the network as a whole. That is, the minimum number of copies will be forwarded over the minimum number of channels. Unfortunately, it is difficult to predict the delay. Some nodes will experience little delay, but others may experience long delays.

The primary drawback of this broadcast technique is that building and maintaining a minimum spanning tree is rather involved. Dalal solved this problem, and Wall has refined it some, but the amount of computation, code, and packets required to implement it is non-trivial.

Centered trees. Centered trees were invented by Wall [1980], as a way to get the benefits of maintaining a single tree (as in the minimum spanning tree case), while controlling delay (as in the shortest-path trees case). The idea is to pick a node which is in some sense "in the center" of the network, and have all nodes use its shortest-path tree for broadcasting. While the delay along this tree will not be optimal, it will be reasonable, and Wall succeeded in deriving bounds on just how reasonable it could be.

Wall considered three centered trees: one that minimized the maximum delay, one that minimized the average delay, and one that was the shortest-path tree with minimum diameter. The algorithms that he developed to pick these trees had two locality restrictions. First, the amount of interaction among nodes (*i.e.* bandwidth) should be minimized, and second, the amount of information used by each node (*i.e.* state) should be minimized and local. Ideally, each node would compute a figure of merit for how good a center it was, and then by means of a special protocol, all of the nodes would agree on who had the best figure of merit, and construct a tree centered on that node.

An implementation based on these algorithms sounds appealing, but the "special protocol" by which the nodes pick the center node and construct the tree is hard. This can be rather involved, though it is not as complex as building a minimum spanning tree.

4.3.5. Reverse path forwarding

Dalal invented a good way to approximate source-based forwarding without requiring each node to maintain the shortest-path trees from all other nodes. It is another case of forwarding along trees; the difference is that the tree topology is derived from information that the nodes must maintain anyway for unicasting. In its simple, non-optimal form, reverse path forwarding uses just the unicast routing tables; if adjacent nodes exchange a small amount of extra information, it can be made to be optimal. This is a good technique for implementing a broadcast mechanism in a store-and-foward network when no special assumptions can be made.

When a broadcast packet arrives, the switch consults its regular unicast routing table and asks the question "if I were to send this packet back to its source, would I use the channel on which it just arrived?" If the answer is "yes", then the switch forwards the packet over all other channels and delivers a copy to its local host; otherwise it discards the packet.

The key point is *whether the broadcast arrived over the channel that the switch would use to send the packet to the source*, because this is a branch of the shortest-path tree to the source.

The suboptimality comes from the switch's behavior while forwarding the packet. The unicast routing table tells it that the *incoming* channel is part of the shortest-path tree, but it does not know which *other* channels are part of it. Since *any* of them might be, just to be safe, it forwards copies out *all* of them, confident that the adjacent switches on channels that are not part of the tree will simply discard them. The cost of this behavior is wasted channel bandwidth and computation in some of the adjacent switches.

To make this scheme optimal, a switch must supply each of its neighbors with a list of addresses for which the neighbor is the shortest path. Broadcast packets from any other source forwarded down that channel by the neighbor would be discarded and therefore should not be sent. This information can travel piggybacked on the normal unicast routing packets that are periodically exchanged by routing internal hosts.

If the delay over a channel is the same in either direction, then the route taken by a reverse-path-forwarded broadcast packet is the shortest-path tree whose root is the source. Delay and bandwidth are both minimized. If a node fails to tell its neighbors the optimizing information, the algorithm can still function provided that the neighbors adopt a conservative attitude and always send it copies.

4.3.6. Flooding

Flooding was first proposed by Baran *et al.* under the name *hot-potato routing* [1964]. Originally it was to be used as a unicast routing method; later proposals for its use have focused on broadcasting. In its simplest form, each switch forwards a copy of an incoming packet out all lines except the one on which it arrived. This is superficially similar to reverse path forwarding, except for the stability problem. The hot-potato routing scheme was proposed for a military network where the wide distribution of packet copies would make it difficult for an evesdropper to deduce traffic patterns, and would increase the probability that at least one copy arrives at its destination; conserving bandwidth was not important to the application.

Many schemes have been proposed for discarding redundant copies, thereby preventing their endless reamplification. One scheme often put forward is to have each switch that originates a broadcast packet assign it a sequence number. Switches keep track of the highest sequence number seen from each possible source and discard old copies. Problems here include switches that crash and forget their sequence numbers, and networks that become partitioned, then sequence numbers wrap around, then the partitions rejoin.

Another approach is to measure the "age" of a packet and discard it when it is too old. There are many ways to do this. Each time a switch forwards a copy, it can increment a *hop count* field in the packet and discard it if it gets too big; a plausible value is the maximum path length of the network. Even for small networks, this technique generates enormous numbers of copies. Yet another approach is to define a maximum packet lifetime for the

network, put that time in the header of the packet, and have each switch decrease it by an estimate of how much time has elapsed since it was seen by the previous switch. Maximum packet lifetime and average packet lifetime are usually quite different, so the packet can ricochet around for a long time before being discarded.

None of these can single-handedly control packet regeneration, but combinations of them, in specific, well-constrained circumstances, *can* be made to work. The "new" Arpanet routing algorithm uses a combination of sequence numbers and packet lifetimes to keep its flooding algorithm stable [Rosen, 1979].

The new Arpanet routing algorithm depends on keeping the information maintained by the routing internal hosts consistent. Whereas the "old" algorithm only exchanged information between adjacent nodes, the "new" algorithm broadcasts to all nodes, and this broadcast must be reliable. Reliability is achieved in the following way: when a routing internal host receives a packet whose sequence number indicates that it is "new", the switch forwards it out *all* channels, including the one on which it arrived. Then the switch waits until it has both sent and received the packet on all channels, in either order, retransmitting as necessary. Thus, a packet is its own hop-by-hop acknowledgement. Every packet crosses each channel exactly twice if there are no lost packets.

This is a *very* special form of broadcast. It depends on the presence of a routing internal host at *each* switch, and such a host must be able to know the channel that a packet arrived on, and control the channels its packets depart on. The routing hosts maintain sequence number state, and take advantage of many timing assumptions peculiar to the routing protocol. It is *by no means* a general scheme for broadcasting, and it can not be extended for use by external hosts.

4.3.7. Nearest-neighbor multicast

For all the years of its existence, the Arpanet has used an interesting special case of multicasting. A routing internal host (in both the old and new algorithms) bypasses the normal packet routing procedure, and causes its switch to forward a copy of a routing packet down every channel. I call this procedure a *nearest-neighbor multicast.* Routing processes in the Pup internet do an analogous thing: they forward copies of their routing tables out each directly-connected network. In the Arpanet case, an Imp unicasts the packet on each channel, since channels are point-to-point and connect to one other switch; in the Pup case, a gateway broadcasts the packet on each network, since each network may connect to many other gateway hosts. The nearest-neighbor multicast of the Arpanet becomes a *nearest-neighbor broadcast* in the Pup internet.

All gateways in the Pup internet contain name and boot servers. The update protocols *depend* on this fact. A server spreads the word about a newer file version by broadcasting on all directly-connected networks, just like a Pup routing information server, but it is only when

a server has connections to more than one network, that the information can propagate from network to network.

Sometimes placing a server in every gateway is not even sufficient to propagate the information, as with the hidden boot files of section 3.2.4. Broadcasting on all directly-connected networks is broadcasting on all networks zero hops away. A generalized version of the nearest neighbor broadcast eliminates dependence on nearby servers to help propagate information. A server might broadcast the existance of a newer version of something on all networks 1, 2, 3 or more hops away; however many hops it took to link up with a kindred server and pass on the information. This algorithm is discussed in more detail in chapter 5.

Chapter 5

Directed broadcasting

> "Have you any notion how expensive an identification search can be?"
>
> "It can't be much. I can't see why you are making such an aching issue of it. I want a clerk to get off his fundament and look in the files. I doubt if they'll bill us. Routine courtesy."
>
> "I wish I thought so, sir. But you've made this an unlimited search. Since you haven't named a planet, first it will go to Tycho City, live files and dead. Or do you want to limit it to live files?"
>
> "No"
>
> "Too bad. Dead files are three times as big as live. So they search at Tycho. It takes a while, even with machines—over twenty billion entries. Suppose you get a null result. A coded inquiry goes out to vital bureaus on all planets, since Great Archives are never up to date and some planetary governments don't send in records anyhow. Now the cost mounts, especially if you use n-space routing... Of course if you take it one planet at a time and use mail..."
>
> "No"
>
> *Citizen of the Galaxy*
> © 1957 Robert Heinlein

Having demonstrated that broadcasting is useful, and that it is feasible in particular well-known networks, we consider in this chapter the architecture of a broadcast mechanism for an internet.

The broadcast mechanism offered to clients of the Pup internet is called a *directed broadcast*. With it, a client can broadcast a Pup on any single network in the internet. This simple mechanism permits clients to build a spectrum of broadcast protocols.

The first section of this chapter discusses some issues that must be considered when designing any internet broadcast mechanism. The second section discusses internet analogs to the store-and-forward broadcasting techniques of chapter 4 and shows where directed broadcasting fits in. The final section discusses ways to use directed broadcasts in the Pup internet to accomplish the two primary uses of a broadcast: search and distribution.

5.1. Internet broadcast design issues

5.1.1. Internets

An internet is a store-and-forward network. Internetwork gateways are its packet switches, and ordinary networks are its communication channels. Figure 5.1 is a block diagram of a basic gateway; compare it with figure 4.3, a block diagram of a basic store-and-forward switch. The network switch of figure 4.3 connects to channels to other packet switches and to locally attached hosts. The internet switch of figure 5.1 connects only to networks. If two hosts are connected to the same network, the network takes care of the switching; a gateway is only involved when the hosts are on different networks.

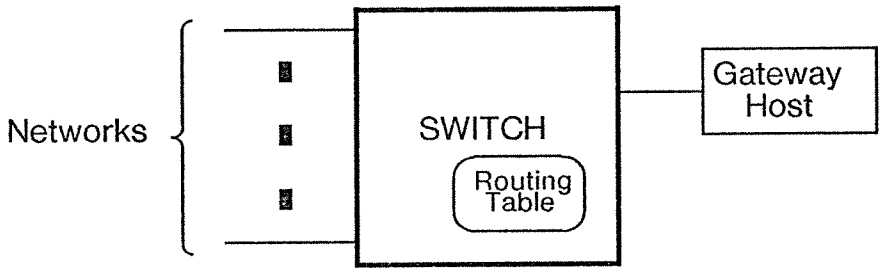

Figure 5.1. Block diagram of a basic gateway.

The store-and-forward network switch of figure 4.3, in addition to forwarding packets to distant switches, implements what amounts to a network for the hosts directly connected to it. Imagine taking all of the locally attached hosts of figure 4.3 and connecting them to their own switch, then connect this switch to the other switch by a channel (see figure 5.2). A surprisingly large amount of Arpanet traffic never leaves the local switch; Kleinrock and Naylor call this *incest traffic* [1974].

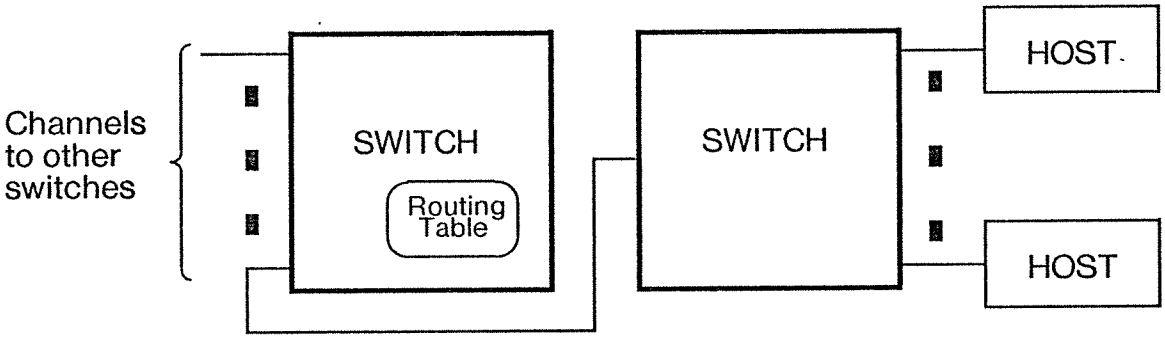

Figure 5.2. A packet switch is also a local network.

In ordinary store-and-foward networks, the *switches* are the center of attention: hosts connect to them and routing tables list them. Even though an internet has a store-and-foward structure, it is the *networks* that are the center of attention: hosts connect to them and routing tables list them. This shift in focus from the switch to the network must be reckoned with when adapting store-and-foward mechanisms to an internet.

5.1.2. Size considerations

As an object becomes more distant (*i.e.* more hops away), details about it become harder to obtain. The further away a host is, the longer it takes for a packet to get there, and the more it costs to transport it. Generally, the bandwidth available along the path decreases, and the probability that a packet will get lost or damaged increases. As the delay increases, the picture of the distant object, built up with great effort, lags reality, and any decision based on it must take this into consideration. It may be better to maintain an *agent* at the distant site and to delegate authority to make certain decisions, within general guidelines. The *broadcast broker*, discussed in section 5.3, is an example of this.

Sometimes the sheer size of the internet can be a problem. Some of the broadcast applications to be discussed use the information in the internet routing table. In a large internet, it may not be practical for a host to possess the entire table (with 255 networks maximum, size is not a problem for Pup, but the information about distant networks may still be out of date). In an internet with hundreds of thousands of networks and millions of hosts, where do you begin when you are looking for something? What search pattern should you use? Who, out of all of those hosts might be interested in the information you have or might have the information you seek?

5.1.3. Reliability

Reliability is relative. A broadcast that is adequately reliable for one system will not be adequately reliable for another. Reliability could mean the delivery of a copy to every host whose receiver was enabled (*i.e.* no copies would be lost by the transmission medium). Or it could mean that delivery, plus the return of an indication of the hosts that received the packet. Or it could mean returning an indication of which hosts had sockets for the packet. Or it could mean returning an indication of the hosts that had processes that acted on the contents of the packet.

In the Pup internet, each network provides an unreliable broadcast. This means that the level-0 driver gives its best efforts to deliver a copy to each host whose receiver is active; no indication is returned of the actual receipt of those copies. This is the basic broadcast mechanism available to level 1.

5.2. Internet broadcast mechanisms

5.2.1. Internet analogs to store-and-forward broadcasting

Since an internet has a store-and-forward structure, let us begin by trying to apply the store-and-forward broadcast techniques of section 4.3 to an internet.

Multidestination addressing requires space in the packet header for the list of networks. To avoid burdening unicast packets with this overhead, its inclusion should be optional. In a large internet, the list of destination networks could dwarf the rest of the packet, so there would have to be a practical upper bound on the number of networks listed in any one packet. This would imply splitting the list among several packets—moving in the direction of the next internet broadcast technique: separately-addressed packets.

Separately-addressed packets are the result if the list in a multidestination-addressed broadcast packet is shrunk in size to one network. This is the internet broadcast technique implemented in Pup. Using a *directed broadcast* packet (see below), a host can broadcast a Pup on any single network in the internet.

Forwarding along trees as an internet broadcast technique suffers from a problem alluded to in section 5.1.1: such an approach distributes copies of a broadcast packet to each *gateway*, whereas it should distribute copies to each *network* (the secondary switch of figure 5.2). This causes duplicate packets on networks with multiple gateways, and can lead to stability problems.

Reverse-path forwarding is a special case of forwarding along trees and suffers from the same problems.

Flooding techniques must work hard to maintain stability—they must generate enough packet copies to cover all destinations without too many duplicates, but at the same time they must insure that those copies die out eventually and do not endlessly regenerate. As we saw in section 4.3, techniques for doing this usually rely on properties of the communication channels and require switches to maintain additional state. The channels (*i.e.* networks) in an internet are rarely homogenous, and requiring that they all have certain properties may limit the types of networks that can be integrated into an internet. Requiring gateways to remember what packets they have seen is always distasteful and the effects on stability when this state is (inevitably) lost must be carefully considered.

5.2.2. *Internet-wide broadcast*

An internet-wide broadcast would deliver a packet to *all* hosts on *all* networks. This is the most obvious candidate for an internet broadcast mechanism and, in my opinion, the worst. In a large internet that one simple act of handing a broadcast packet to level 1 could cause millions of hosts to receive (and usually discard) a copy of the packet. That is too easy; a capricious or malicious program could cause long-distance grief far out of proportion to its other capabilities—so could a program gone berserk.

One has no control over an internet-wide broadcast—once a broadcast has started, it is impossible to stop. It will flow out to the ends of the internet, bothering *everyone.* Ruling out internet-wide broadcasts eliminates tree-based and flooding techniques from further consideration; we are left with techniques based on multiply- and separately-addressed packets. Here we have some control, and being able to reach network-sized batches of hosts with one packet is still sufficiently powerful. Choosing between these two approaches turns mainly on the architectural choice of whether to use variable-length packet headers. In the Pup architecture, per-packet processing overhead must be small; requiring gateways to modify packet headers, or worse, change their lengths runs counter to this policy. The Arpa internet [Postel *et al.*, 1981], on the other hand, has embraced variable-length headers, so an internet broadcast mechanism in which a packet could specify a moderate number of destination networks fits very nicely. Arpa gateways are already required in some cases to inspect and change the length of packet headers as they are forwarded.

5.2.3. *Directed broadcast*

In the Pup internet, a directed broadcast allows a client to broadcast a packet to *all* hosts on *one* network. A directed broadcast Pup travels from gateway to gateway through the internet just like a unicast packet. When the Pup arrives at a gateway that is directly connected to the destination network, the gateway's output process hands the Pup to the level-0 driver for that network, which broadcasts it, in whatever way is proper on that network. We saw in chapter 4 that broadcasting is done in different ways in different types of networks.

An illustrative nickname for the directed broadcast packet is "letter bomb." The packet looks perfectly innocent as it wends its way through the internet. Finally it arrives at the victum network where it explodes into a broadcast.

An internet-wide broadcast can be accomplished by enumerating all of the networks in the internet, and sending a directed broadcast packet to each one. This is the internet analog to broadcasting with separately-addressed packets.

As the internet grows, so does the number of networks. In a large internet, it may be difficult to find out about the existence of all of the networks, especially those far away. In the next section we will discuss a solution to this problem.

5.3. Using directed broadcasts

Broadcasting is useful for search and distribution; this section describes some techniques for accomplishing these in an internet.

5.3.1. Expanding ring searches

Imagine that you are searching for a resource in the Pup internet, and you have no idea where it could be. It might be in a host in the next room, or in a host on another continent. Once you have found it, you wish to open a connection and engage in some complicated transaction with it. There might be several copies of what you seek; it would be useful to find the closest one.

The problem as stated sounds rather vague. Here are some concrete examples of it with which you are already familiar:

> **Name Lookup.** You are looking for a name server. You want to convert a name into an address. The broadcast packets with which you search contains the lookup request, a server will respond with a unicast packet, and that will be the end of the transaction.
>
> **Boot file update.** You are a boot server. You want to find other boot servers and compare creation dates of any boot files you have in common. If you find a newer version, you will then open a connection and retrieve a copy of the newer file.
>
> **Time maintenance.** You are a time user or server. The error of your clock has hit your reset threshold, and you are looking for a time server whose error is less than your acceptance threshold.

Name servers also periodically search for newer versions of the name database, much the same way boot servers look for newer boot files. This can also be viewed as a distribution problem ("I have this version; if yours is older, get a new copy from me").

Since the thing you seek could be anywhere, you need some strategy for conducting the search—a search pattern that covers the terrain in some logical way without searching the same area twice or missing some area. One such pattern is the *expanding ring search.* It is simple to describe: first you search (*i.e.* broadcast on) the networks zero hops away. If you do not get a suitable answer, then broadcast on all networks one hop away, then two, then three, until you have searched the entire internet or found what you are looking for.

The broadcast on all networks zero hops away covers the circle of networks with you at the center, which is to say that it covers all networks to which you are directly connected. Broadcasting on all networks one hop away skips that circle, and covers a ring of networks around the circle. Two hops away skips the first two rings, and covers a bigger ring; thus the name *expanding ring search.* Of course, if you are not in the topological center of the internet,

the expanding rings may intersect the edge of the internet before the search is complete, and the remaining rings will be arcs—some may even be disconnected.

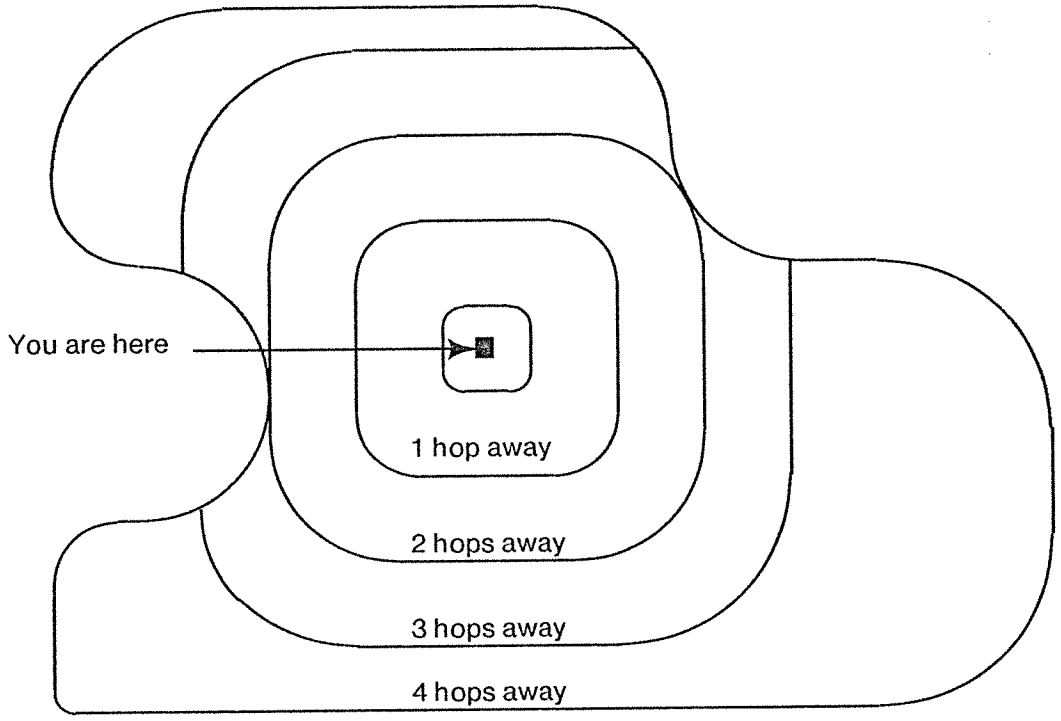

Figure 5.3. Expanding ring search.

In the worst case, you will end up broadcasting on all networks in the internet. However, if you find what you are looking for, you can stop the search before reaching networks further out. There are also situations in which you might want to limit the depth of the search: go out *n* hops and stop even if you have not found the search object. This might be essential for really large internets because you probably will not be able to afford to broadcast on *every* network.

There are several strategies for pacing the search. You could wait for answers after broadcasting on each network. Or you could generate a flurry of packets to all networks that are the same number of hops away and then wait for any answers to come in. Or you could step through the networks at a stately pace, and stop when you get to the end, or get a suitable reply. Since the broadcast is not necessarily reliable, you might want to broadcast on each network several times.

If the routing system maintains other metrics, you could use these to control the search. Instead of sorting the routing table by hops, you might sort by delay, or bandwidth, or cost.

5.3.2. Broadcast brokers

Suppose you are in Boston, and you know that the thing you are looking for is somewhere in the San Franciso Bay area. San Francisco is far away, and your local routing table does not contain the right information to tell you which networks that are in the Bay area.

If you knew of a network in the Bay area, then you could broadcast a routing information request packet on that network. The gateway(s) on that network would reply with routing information packets, and you could build a routing table just as if you were a host on that distant network. You could then use this table to control an expanding ring search. Your search packets and any responses would travel all the way across North America.

Another possibility is to use a *broadcast broker.* He can act as your agent in the Bay area, and conduct the search for you. You describe what it is you are looking for and how much you are willing to spend. The broker is an expert at searching, and knows the ins and outs of the San Francisco area topology—such as which networks are *really* in the Bay area (a satellite network with a ground station in the Bay area will be only a few hops away from most networks there, but it is not really in the Bay area, and broadcasting on it is probably a waste of time and money).

There is the problem of how to find a broadcast broker in the Bay area. He advertises, and is a member of the broadcast brokers association, which maintains directories on all major networks. A search starting from just about anywhere will find a copy of the brokers directory without looking very far. Locating a broker is conceptually no different from locating a name server.

5.3.3. Name database and boot file distribution

The process by which new versions of the host name database and boot files propagate around the Pup internet are very similar and were described in detail in chapter 3. Briefly, a new version of a file is prepared with a version number higher than the currently released version. The machine possessing the newer file then broadcasts the new version number, a nearby server hears it, retrieves a copy, and in turn broadcasts it, spreading the word further.

In the original design, each server broadcast on its directly-connected networks. For a new version to leave one network and get into another required that there be a server with connections to both networks. The server would hear the broadcast on one network, get a copy, and broadcast the new version on both networks, thereby spreading the word to the other network. Gateways are convenient places to locate such servers, and they are connected to multiple networks, so as long as every gateway containes a server, everything works. Such a server is acting as a "high-level gateway", a store-and-forward switch for the files.

This system configuration constraint, that gateways must contain name and boot servers, is eliminated by a simple application of directed broadcasts. When a server retrieves a newer file, it broadcasts this fact on its directly-connected networks (those zero hops away) so others on those networks will find out, and it also broadcasts the new version on networks a few more hops away to spread the word into nearby networks.

5.3.4. The island problem

How many hops away should a server distribute version information by broadcasting? If the information is not spread far enough, it is possible for *islands* to form in the internet. An island is a group of networks containing some servers such that if a new version is released inside, all servers on the island will get it, but servers outside, on other islands will not. Locating a server in every gateway guarantees that no islands can form even though a server only broadcasts on directly-connected networks.

In the worst case, assuming nothing about where other servers might be located, a server must broadcast new version information on all networks in the internet to insure that islands do not form. To see this, imagine an internet in which the maximum number of hops between any two networks is n. Now imagine that there are exactly two servers and that they are on networks which are separated by n hops. They will not discover each other except by broadcasting out to n hops—an internet-wide broadcast.

Fortunately most situations are more constrained than this. Networks in the Pup internet divide into two classes: end-user networks and transit networks. Workstations are connected to end-user networks; but transit networks only connect gateways. Almost every end-user network has name and boot servers; transit networks usually do not. It is those transit networks that can cause problems. If two regions of the internet are only connected by a long string of transit networks, then the servers at each end of the string must broadcast further to reach each other. In reasonable configurations of a Pup internet, end-user networks are not separated by more than a few concatenated transit networks. Pup name and boot servers broadcast version information frequently on directly-connected networks, and less frequently further out. How much further out is a local configuration parameter for each server. The default value, for the Pup internet today, is three hops.

5.3.5. Summary

The key point is that the various broadcast-based strategies for searching and distributing should be implemented by *hosts*, not by the internet. All the internet should provide is one simple, easy-to-implement and efficient mechanism: directed broadcast.

Chapter 6

Conclusion

> "...I'll let a message echo around the inhabited planets that the Archives are here, too."
>
> *Time Enough for Love*
> © 1973 Robert Heinlein

6.1. Summary

My thesis is that broadcasting should be a standard addressing mode of all packet switched computer networks and internets. The argument was divided into four parts:

The Pup internet. The Pup internet is the largest such system in the world. Throughout the thesis, it is used as a real environment in which to judge and compare things.

Why broadcast? Broadcasting is a form of interprocess communication distinct from unicasting. Broadcasting makes possible new forms of distributed computations. Four broadcast-based distributed applications from the Pup internet demonstrate the power and utility of broadcasts.

How to broadcast. An "unreliable" broadcast is a sufficient basic mechanism and is easy to implement. Just as with unicasting, higher-level protocols outside of the network can be used to tailor this basic mechanism to the needs of particular applications.

Directed broadcasting. Having demonstrated why broadcasting is useful, and how to broadcast in individual nets, I described Pup directed broadcasting, a simple broadcast mechanism from which many types of internet broadcast can be constructed.

My experience with the Ethernet and the Pup internet is that broadcasting is a powerful tool. I hope that, after reading this thesis, nobody will ever again design a network or an internet that does not provide broadcast logical connectivity.

6.2. Areas for future work

6.2.1. Distributed computing

The character of a distributed computation is strongly affected by the available interprocess communication facility. Broadcasting is a different form of interprocess communication than unicasting. It makes different forms of distributed computing possible, such as the route, name, time and boot services described here. I'm sure people can improve on the simple protocols I have presented. What is needed now is for lots of people to try lots of uses for broadcasts. I know there are many surprises left to discover.

6.2.2. Multicasting

Multicasting needs a thorough treatment. I think the situation is analogous to broadcasting. Before the arrival of the Ethernet, in which broadcasting was easy, the dominant network structure was store-and-forward, in which broadcasting seemed hard. Because it seemed hard, nobody did it, so nobody discovered the power of broadcasting. Once it became clear that store-and-forward networks would be in the minority, and adding broadcasting to them would make it possible to offer broadcasting in an internet, the barrier was overcome.

What we need now is a widely-accepted network which offers a useful multicast capability. David Wall and others are working on multicasting in store-and-forward netwoks; Xerox Corporation is wrestling with multicasting in Ethernets. Very large scale integrated circuits will soon permit low-cost implementation of multicast mechanisms in hardware, and people should be thinking of applications.

References

[Abraham & Dalal, 1980]
S. M. Abraham and Y. K. Dalal, "Techniques for Decentralized Management of Distributed Systems" *IEEE Computer Society International Conference (CompCon)*, Spring 1980, pp. 430-437.

[Baran *et al.*, 1964]
P. Baran, S. P. Boehm, J. W. Smith, "On Distributed Communications", An 11-volume series of the RAND Corporation, summarized in vol. 11, Memorandum RM-3767-PR, August 1964.

[Bechtolsheim & Baskett]
A. Bechtolsheim and F. Baskett, "The SUN workstation", unpublished draft of a Stanford University Computer Science technical report.

[Boggs *et al.*, 1980]
D. R. Boggs, J. F. Shoch, E. A. Taft, and R. M. Metcalfe, "Pup: An Internetwork Architecture", *IEEE Transactions on Communications*, COM-28(4), April 1980, pp. 612-624.

[Cerf and Kahn, 1974]
V. G. Cerf and R. H. Kahn, "A Protocol for Packet Network Interconnection", *IEEE Transactions on Communications*, COM-22(5), May 1974, pp. 169-178.

[Clark *et al.*, 1978]
D. D. Clark, K. T. Pogran, and D. P. Reed, "An Introduction to Local Area Networks", *Proceedings of the IEEE*, 66(11), November 1978, pp. 1497-1517.

[Cohen, 1977]
D. Cohen, "Issues in Transnet Packetized Voice Communication", *Proceedings of the 5th Data Communications Symposium*, Snowbird, Utah, September 1977, pp. 6-10 to 6-13.

[Crocker *et al.*, 1972]
S. D. Crocker, J. F. Heafner, R. M. Metcalfe, and J. B. Postel, "Function-oriented Protocols for the ARPA Computer Network", *AFIPS Conference Proceedings (SJCC)*, vol. 40, 1972, pp. 271-279.

[Dalal, 1977]
Y. K. Dalal, "Broadcast Protocols in Packet Switched Computer Networks", Technical Report No. 128 (Ph.D. thesis), Digital Systems Laboratory, Stanford University, April 1977.

[Dalal & Printis, 1981]
Y. K. Dalal and R. S. Printis, "48-bit Absolute Internet and Ethernet Host Numbers", *Proceedings of the 7th Data Communications Symposium*, Mexico City, October 1981.

[Davidson *et al.*, 1977]
J. Davidson, W. Hathaway, J. Postel, N. Mimno, R. Thomas, and D. Walden, "The Arpanet Telnet Protocol: Its Purpose, Principles, Implementation, and Impact on Host Operating System Design", *Proceedings of the 5th Data Communications Symposium*, Snowbird Utah, September 1977, pp. 4-10 to 4-18.

[deJardins, 1981]
R. desJardins, "Overview and Status of the ISO Reference Model of Open Systems Interconection", *Computer Networks*, 5(2), April 1981, pp. 77-80.

[DoD, 1980]
DARPA, "DoD standard internet protocol", Internet Experimental Note (IEN) No. 128, January 1980.

[Falk and Koolish, 1980]
G. Falk and R Koolish, "Wideband Packet Satellite Network Host Access Protocol", W-Note 17, Bolt Beranek and Newman, November 1980.

[Farber, 1975]
D. J. Farber, "A ring network", *Datamation*, February 1975, pp. 44-46.

[Farber *et al., 1973*]
D. J. Farber, J. Feldman, F. R. Heinrich, M. D. Hopwood, K. C. Larson, D. C. Loomis, and L. A. Rowe, "The Distributed Computing System", *Digest of Papers, CompCon*, February 1973, pp. 89-98.

[Fienler & Postel, 1978]
E. Feinler and J. Postel, eds., Telnet Protocol Specification, *Arpanet Protocol Handbook*, January 1978.

[Harrenstien, 1977]
K. Harrenstien, "Time Server", *Arpanet Protocol Handbook*, January 1978, p. 479, also RFC 738.

[Hopper, 1981]
A. Hopper, personal communication.

[Kahn, 1977]
R. E. Kahn, "The Organization of Computer Resources into a Packet Radio Network", *IEEE Transactions on Communication*, COM-25(1), January 1977, pp. 169-178.

[Kahn *et al.*, 1978]
R. E. Kahn, S. A Gronemeyer, J. Burchfiel, and R. C. Kunzelman, "Advances in Packet Radio Technology", *Proceedings of the IEEE*, 66(11), November 1978, pp. 1468-1496.

[Kleinrock & Naylor, 1974]
L. Kleinrock and W. E. Naylor, "On measured behavior of the ARPA network", *AFIPS Conference Proceedings (NCC)*, 1974, pp. 767-780.

[Lamport, 1978]
L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, 21(7), July 1978, pp. 558-565.

[McQuillan, 1974]
J. M. McQuillan, "Adaptive Routing Algorithms for Distributed Computer Networks", Bolt Beranek and Newman Report No. 2831 (Harvard Ph.D. thesis), May 1974.

[McQuillan & Walden, 1977]
J. M. McQuillan and D. C. Walden, "The ARPA Network Design Decisions", *Computer Networks*, 1(5), August 1977, pp. 243-289.

[Metcalfe, 1973]
R. M. Metcalfe, "Packet Communication", Technical Report No. 114 (Harvard Ph.D. thesis), MIT Project MAC, December 1973.

[Metcalfe & Boggs, 1976]
R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM*, 19(7), July 1976, pp. 395-404.

[Mills, 1981]
D. L. Mills, "Time Synchronization in DCNET Hosts", Arpa Internet Experimental Note (IEN) No. 173, February 1981.

[Mockapetris, 1978]
P. V. Mockapetris, "Design Considerations for the ARPA LNI Name Table", Technical Report No. 92, U.C. Irvine Dept. of Info. and Computer Science, May 1978.

[Pickens *et al.*, 1979]
J. R. Pickens, E. J. Feinler, and J. E. Mathis, "The NIC NameServer (NICNAME)—a Datagram-Based Information Utility", *Proc. 4th Berkeley Workshop on Distributed Data Management and Computer Networks*, August 1979, pp. 275-283.

[Peacock *et al.*, 1979]
J. K. Peacock, E Manning, and J. W. Wong, "Synchronization of Distributed Simulation using Broadcast Algorithms", *Proc. 4th Berkeley Workshop on Distributed Data Management and Computer Networks*, August 1979, pp. 237-259.

[Plummer, 1977]
W. W. Plummer, "Internet Broadcast Protocols", Arpa Internet Experimental Note (IEN) No. 10, March 1977.

[Postel, 1979]
J. Postel, "Internetwork Protocols", *IEEE Transactions on Communications*, COM-28(4), pp. 604-611.

[Postel *et al.*, 1981]
J. Postel, C. Sunshine, and D. Cohen, "The ARPA Internet Protocol", *Computer Networks*, 5(2), April 1981, pp. 261-271.

[Reed, 1978]
D. P. Reed, "Naming and Synchronization in a Decentralized Computer System", MIT Ph.D. thesis, September 1978.

[Rosen, 1979]
E. C. Rosen, "The Updating Protocol of the ARPANET's New Routing Algorithm", *Computer Networks*, 4(1) February 1980, pp. 11-19.

[Rowe, 1975]
L. A. Rowe, *The Distributed Computing Operating System*, DCS Technical Report No. 66, U.C. Irvine Dept. of Info. and Computer Science, June 1975.

[Shoch, 1978]
J. F. Shoch, "Internetwork Naming, Addressing, and Routing", *17th IEEE Computer Society International Conference (CompCon)*, September 1978, pp. 72-79.

[Shoch, 1979]
J. F. Shoch, "Design and Performance of Local Computer Networks", Stanford Ph.D. thesis, August 1979.

[Shoch *et al.*, 1980]
J. F. Shoch, D. Cohen, and E. A. Taft, "Mutual Encapsulation of Internetwork Protocols", *Computer Networks*, 5(4), July 1981, pp. 287-300.

[Shoch and Hupp, 1980]
J. F. Shoch and J. A. Hupp, "Notes on the "Worm" programs—some early experience with a distributed computation", Xerox PARC technical report SSL-80-3, September 1980.

[Shoch & Stewart, 1979]
J. F. Shoch and L. C. Stewart, "Interconnecting Local Networks via the Packet Radio Network", Xerox PARC technical report SSL-79-4, February 1979.

[Sproull & Cohen, 1978]
R. F. Sproull and D. Cohen, "High-Level Protocols", *Proceedings of the IEEE*, 66(11), November 1978, pp. 1371-1385.

[Strazisar, 1979]
V. Strazisar, "How to Build a Gateway", ARPA Internet Experimental Note (IEN) No. 109, August 1979.

[Sunshine and Dalal, 1978]
C. Sunshine and Y Dalal, "Connection Management in Transport Protocols", *Computer Networks*, 2(6), December 1978, pp. 454-473.

[Thacker *et al.*, 1980]
C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs, "Alto: A Personal Computer", in *Computer Structures: Readings and Examples*, second edition, Siewiorek, Bell and Newell, editors, McGraw-Hill, 1981, pp. 549-572.

[Thomas, 1973]
R. H. Thomas, "A Resource Sharing Executive for the ARPANET", *AFIPS Conference Proceedings*, 1973, pp. 155-163.

[Wall, 1980]
D. W. Wall, "Mechanisms for Broadcast and Selective Broadcast", Technical Report No. 190 (Ph.D. thesis), Computer Systems Laboratory, Stanford University, June 1980.

[Wilkes & Wheeler, 1979]
M. V. Wilkes and D. J. Wheeler, "The Cambridge Digital Communication ring", *Local Area Communications Symposium*, Mitre and NBS, Boston, May 1979, pp. 47-61.

[Xerox *et al.*, 1980]
Xerox Corporation, Digital Equipment Corporation, and Intel Corporation, "The Ethernet: A Local Area Network; Data Link Layer and Physical Layer Specifications, version 1.0", September 1980.

[Zimmermann, 1979]
H. Zimmermann, OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection, *IEEE Transactions on Communications*, COM-28(4), April 1980, pp. 425-432.