Chapter 3

Patterns in Protocols

The trick with reductio ad absurdum is knowing when to stop.

Introduction

In this chapter, we begin to get to the crux of the matter: finding patterns in the architecture of networks. And not just any patterns, but patterns that go beyond natural history and make predictions and provide new insights. The task is made more difficult by the nature of computer science; that is, we build what we measure. Unlike physics, chemistry, or other sciences, the patterns that form the basis of our field are seldom fixed by nature, or they are so general as to provide little guidance. For us, it is more difficult to determine which patterns are fundamental and not an artifact of what we build.

Even in what we think of as the traditional sciences, finding the problem at the core of a set of problems is not always obvious. (Although it always seems so in retrospect.) For example, one of the major problems in the late 16th century was predicting where cannonballs would fall. Rather than proposing an elaborate and expensive project to exhaustively explore their behavior with a highly instrumented collection of cannons of various makes, caliber, and amounts of powder and from this try to determine the equations that would predict the path of the cannonballs, Galileo had the insight that the answer lay not with firing cannons but with some hard thinking about a simple abstraction that was at the core of the problem. The key for Galileo was to break with Aristotle and imagine something no one had ever seen or had any reason to believe could exist: frictionless motion. Then formulate what we know as the first law of motion, "A body at rest or in motion will tend to stay at rest or in motion…." (Fermi and Bernardini, 1961; a little gem of a book). Imagine how absurd and idealistic such a construct must have appeared to his colleagues. Everyone knew that an object put in motion slowed to a stop unless a force was

acting on it. One saw it every day. What was this dream world that Galileo inhabited? Push an object and it goes on forever? Absurd! [1]

Galileo could then confirm his insight by rolling inexpensive balls down an inclined plane or simply dropping them from high places. Had Galileo gone directly for the problem at hand, he would never have found the answer. It was far too complex. To start from nothing to find equations of motion that accommodate factors of air resistance, wind drift, shape of the projectile (not perfect spheres), and so on would have been all but impossible. Galileo had the insight to find the model at the core of problem. (One wonders if Galileo had the same problems getting funding for experiments that were not of immediate practical application that one would have today. Luckily, an inclined plane and a few balls don't cost much now or didn't then.) We must also look for the model at the core of our problem to find the concepts that will pull it all together. And like Galileo, we may find that some hard thinking is more productive and less expensive.

It would seem that because we have much more leeway in our choices and very little help from nature to determine which ones are right, that it will be difficult to justify choosing one over another. To some extent this is true, but we are not totally adrift. The experience of the 16th- and 17th-century scientists allowed, by the early 18th century, for science to arrive at some guidance we can fall back on: the *Regulae Philosphandi* from Newton's *Principia* of 1726 (as paraphrased by Gerald Holton, 1988):

1. Nature is essentially simple; therefore, we should not introduce more hypotheses than are sufficient and necessary for the explanation of observed facts. This is a hypothesis, or rule, of simplicity and *verae causae*.

2. Hence, as far as possible, similar effects must be assigned to the same cause. This is a principle of uniformity of nature.

3. Properties common to all those bodies within reach of our experiments are assumed (even if only tentatively) as pertaining to all bodies in general. This is a reformulation of the first two hypotheses and is needed for forming universals.

4. Propositions in science obtained by wide induction are to be regarded as exactly or approximately true until phenomena or experiments show that they may be corrected or are liable to exceptions. This principle states that propositions induced on the basis of experiment should not be confuted merely by proposing contrary hypotheses.

[1] While Galileo also uncovered other principles, this is probably the most counterintuitive.

Not only are these good for nature, but also for finding fundamental structures. As the reader is well aware, the path to finding such solutions is seldom a nice, straightforward progression. Along the way, there are always twists and turns, blind alleys, backtracking, and some intuitive leaps that we will only see later, the straightforward path that led to them followed by throwing away of one or more ladders. I will do what I can to protect you from the worst of these, while at the same time giving you a sense of how I came to these conclusions. All the while, we will try to listen to what the problem is telling us. We will assume that what others have done was for good reason and offers clues to patterns that may have remained obscured. But be aware that there is some hard thinking ahead. You will be asked to set aside preconceived notions to see where a new path leads, and some things you thought were fact were artifacts of our old ways. It is not so much that the old ways of thinking were wrong. They were necessary to a large extent for us to make progress. We had to see how the problem behaved to have a better understanding of the principles underlying it. In fact, it is unlikely that we could have gotten to a better understanding without them. Any scientific theory is always a working hypothesis: an indication of our current understanding; something to be improved on.

Before we can begin that process, however, we must address the great religious war of networking. A war that has raged for the past 30 years or more and at this point has been and remains probably the greatest barrier to progress in the field. The war revolves around two topics that are not only technical but also historical, political, and worst of all economical. (*Worst,* because ideas that change business models and who can make money are the most threatening, outside actual religion.) The conflict is between the two major architecture paradigms, beads-on-a-string and layers, and the conflict between connection and connectionless. The war continues unabated to this day. And although it may appear that because the Internet has been such a success that the connectionless layered approach has won the day, this is far from apparent. Virtually every proposal for new directions or new technology falls into one camp or the other. And it seems that beads-on-a-string proposals are once again on the rise. Proponents lobby hard for their favorites and demean proposals of

A Word of Warning

This topic has been the most hotly contested in the short 30+ year history of networking. No one has come to blows over it (as far as I know), but it has been awfully close at times. Strong emotions and shouting matches have not been uncommon. Conspiracy theories abound, and some of them are even true. So the reader should be aware of the intensity these issues tend to generate and the powers that are brought to bear. I will try to be even-handed and will conclude I have succeeded if I am criticized for being unfair by both sides. But not everyone can be right.

And full disclosure: As long as we are on a topic of such sensitivity, I should make you aware of my own history with these topics. I was involved in the early ARPANET an avid proponent of the connectionless approach found in CYCLADES and the Internet. I was one of those responsible for ensuring that the Europeans held up their bargain to include connectionless in the OSI reference model. It is not uncommon for supporters of the *post, telephone, and telegraph* (PTT) position (that is, bellheads) to change the subject of discussion when I walked in a room. On the other hand, some members of the *Internet Engineering Task Force* (IETF) probably assume I am a connection-oriented bigot (probably because I have been a long-time critic of the disappearance of the vision and intellectual risk taking that made the early Net a success).

the other camp. Some loudly champion views that they see as inherently good for the Net because they reflect some Utopian myth without showing how they solve real problems or yield real benefits. None of the proposals generate that sense of a "right" solution and, hence, none get any traction. Unless we can find a resolution to this crisis, especially one that provides a synthesis of connection and connectionless, networking will continue its imitation of the Faber Marching Band (in the movie *Animal House*).

One of the things that most impressed me about the early work of the ARPANET *Network Working Group* (NWG) was its ability when confronted with two extreme positions to find a solution that was a true synthesis. Not the typical standards committee approach of simply jamming both views together and calling them options, but a solution that went to depth and found a common model that encompassed the extremes as degenerate cases. Time after time, the NWG found these solutions. Not only were they an elegant synthesis, but also simple and easy to implement. OSI was too politicized to do it, and the IETF seems to have lost the spirit to do it.

It appears that there are times when both connections and connectionless make sense. After all, the architectures that support connectionless have connections, too. We need to understand when one or the other is preferred. I have believed there had to be something we weren't seeing: a model in which connections and connectionless were both degenerate cases. I have spent many hours over many years struggling with the problem looking for a model that maintained the best of both worlds. Perhaps we can find one here.

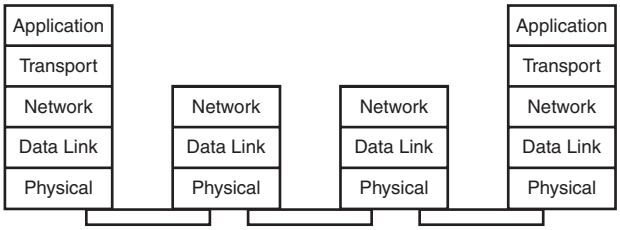

Figure 3-1 Typical network architecture of the early 1970s.

The Two Major Architecture Paradigms

The Layered Model

By one of those quirks of history, the first major computer networks (ARPANET, CYCLADES, and NPLnet) were built primarily, not by communications experts,

but by computer experts, in particular operating systems experts. In 1970, software engineering was barely two decades old, and design principles were only beginning to coalesce. Operating systems were the most complex programs of the day; and if computers were to use networks, it would have to be through the operating system. Therefore, it is not surprising that Dijkstra's paper (1968) on the elegant and simple layered design of the THE operating system and Multics (Organick, 1972), the basis of UNIX, would influence early attempts to find a structure for the new networks. This combined with the justification of the ARPANET as a resource-sharing network served to impart a strong influence of operating systems. The first applications were modeled on providing the major functions of an operating system in a network.

This exercise of finding abstractions to cover the variety of heterogeneous systems also led to a deeper understanding of operating systems. (In my own case, trying to replicate semaphores in a network lead to a solution to reliably updating multiple copies of a database [Alsberg, Day; 1976].) The primary purpose of Dijkstra's layers was the same as any "black box" approach: to provide an abstraction of the functions below and isolate the users of functions from the specifics of how the function worked and from specifics of the hardware. Higher layers provided higher abstractions. This also allowed the functions within a layer to be modified without affecting the layers on either side. In addition, the tight constraints on resources led Dijkstra to believe that there was no reason for functions to be repeated. A function done in one layer did not have to be repeated in a higher layer. The Dijkstra model had gained currency not only in operating systems but in many other application areas, too. It seemed especially well suited for the distributed resource-sharing network, where not only were computers sending information to each other, but the switches to move traffic between source and destination hosts were also computers, albeit minicomputers, but still general-purpose computers nonetheless. Hence, an architecture of at least five layers was fairly commonly accepted by 1974 (see Figure 3-1):

1. A physical layer consisting of the wires connecting the computers
2. A link layer that provided error and flow control on the lines connecting the computers
3. A relaying layer that forwarded the traffic to the correct destination
4. A transport layer responsible for end-to-end error and flow control
5. An applications layer to do the actual work

At least five layers; because as we will see in the next chapter, the ARPANET had adopted a quite reasonable approach of building one application on the

services of another; no one at this point believed they understood what this structure was, but it was assumed there was more structure. It was unclear how many layers there should be above transport.

Figure 3-2 A typical beads-on-a-string architecture of the early 1970s (same as Figure 2-1).

The Beads-on-a-String Model

The early research computer networks were not the first networks. The telephone companies had been building networks for nearly a century, and these were large international networks. Clearly, they had developed their own network architecture suited to their needs. However, the properties of the architecture reflected not only the circuit-switched technology of the telephone networks but also their economic and political environment. From their founding in the 19th century until after the middle of the 20th century, the telephony networks were electrical (physical circuits). Even in the last half of the 20th century as switches used digital communication for control, this control was *alongside* the telephony network. Interfaces were always between devices. Although this architecture has never been given an official name, I have always called it the "beads-on-a-string" model after the figures common in their specifications: different kinds of boxes strung together by wires.

The beads-on-a-string model reflects the unique environment occupied by the telephone companies (see Figure 3-2). It has been primarily promulgated by the CCITT/ITU and the telephone companies and manufacturers closely associated with them. First, until recently, all telephone switching was circuit switched, and hence there was only one layer, the physical layer. Strictly speaking, these networks consisted of two physically distinct networks: one that carried the traffic and a separate one for controlling the switching. More recently, these are referred to as the data plane and control plane. (Another indication of the Internet's slide into telephony's beads-on-a-string model.) The communication generated by these two planes is sometimes multiplexed onto a common lower layer rather than physically distinct networks. This split between the voice and switch control was well established. There was no reason for a layered model. This also leads to a very connection-oriented view of the world. Second, until recently telephone companies were monopolies that either manufactured their own equipment or bought equipment built to their specifications from a very

small number of manufacturers. Hence, where standards were required, they were to define interfaces between boxes, between a provider and someone else (that is, another telephone company or, if absolutely necessary, a customer). In the preferred solution, all equipment used by the customer is owned by the telephone company. This was the situation prior to deregulation. Therefore, a major purpose of the model is to define who owns what (that is, define markets). In this environment, it is not surprising that a beads-on-a-string model evolved. And herein lies one of the first differences between the layered and beads-on-a-string model: the definition of interface. In the layered model, an interface is between two layers internal to a system, within a box. In the beads-on-a-string model, an interface is between two boxes.

One can imagine the combination of confusion and indignation with which the telephone companies faced the idea of computer networks at the beginning of the 1970s. On the one hand, this was their turf. What were these computer companies doing infringing on their turf, *their* market? On the other hand, these research networks were being built in a way that could not possibly work, but did, and worked better than their own attempts at computer networks. In Europe, the telephone companies, known as PTTs, were part of the government. They made the rules and implemented them. In the early 1970s, it was a definite possibility that they would attempt to require that only PTT computers could be attached to PTT networks. Everyone saw that communication between computers was going to be a big business, although it is unlikely that the PTTs had any idea how big. (As late as the late 1980s, phone company types were still saying that data traffic would never exceed voice traffic.) The PTTs saw an opportunity for value-added networks, but they did not like the idea of competition or of companies creating their own networks.

The situation in the United States was very different. AT&T was a monopoly, but it was a private corporation. It thought it understood competition and saw this as a chance to enter the computer business. But the European PTTs knew they did not like the layered model because as we will soon see, it relegated them to a commodity market. The layered model had two problems for PTTs. Because most of the new (they called them value-added) services were embodied in applications and applications are always in hosts, which are not part of the network, there is no distinction between hosts owned by one organization and hosts owned by another. Any

Why Do We Care?

"This is all interesting history that may have been important in your day, but it is hardly relevant to networking today!"

Oh if that it were the case. First, it is always good to know how we got where we are and why we are there. There is a tendency in our field to believe that everything we currently use is a paragon of engineering, rather than a snapshot of our understanding at the time. We build great myths of spin about how what we have done is the only way to do it to the point that our universities now teach the flaws to students (and professors and textbook authors) who don't know better. To a large extent, even with deregulation, little has changed.

To be sure, the technology is different, the nomenclature has changed, the arguments are more subtle; but under it all, it is still the same tension. Who gets to sell what? Who controls the account? How does this affect

continues

continued

my bottom line? The same economic and political factors are still driving the technology. And the providers are still trying to find a way to turn the layered model to their advantage, whether they call it value-added service, AIN, IPSphere, or IMS. It is still the same game. Any time one hears of wonderful services for applications "in the network," it is one more ruse to draw a line between what they can sell and no one else can. Today, one hears router vendors making statements that sound uncannily like the PTTs of 1980s. Product offerings are never about elegant paragons of engineering but offensive and defensive moves in the game of competition. If you don't see how a particular product does that for a vendor, you aren't thinking hard enough.

new services were open to competition. The second problem was connectionless, which we cover in the next section.

The PTTs much preferred an approach that allowed them to use their monopoly. They dug in and would not concede an inch to the computer industry without a fight. This "bunker mentality" became and remains a characteristic of both sides. Each side was deathly afraid that the slightest concession to the other side would lead to a collapse of their position. For the layered advocates, it was the immense political power of the PTTs; and for the PTTs, it was the immense market and technological pressure of the computer industry that drove their fear. But also, the lack of any technical middle ground contributed to the tension. For both sides, there did not seem to be any ground to give that did not lead to the collapse of one's position.

To illustrate this mindset and how architecture was used for competitive ends, consider the simple example in Figure 3-3 from the period. In both the ARPANET and the PTT packet networks, many users gained access by dialing a terminal into a computer that had a minimal user interface that enabled the user to then connect to other computers on the network. In the ARPANET, there were two kinds of these: one that was a stand-alone host (usually if not always a PDP-11) connected to an *Interface Message Processor* (IMP), or a version of the IMP with a very small program to handle the user interface called a *Terminal Interface Processor* (TIP). (But the TIP software looked like a host to the IMP software.) As Figure 3-3 shows, the configuration of boxes was the same in both networks.

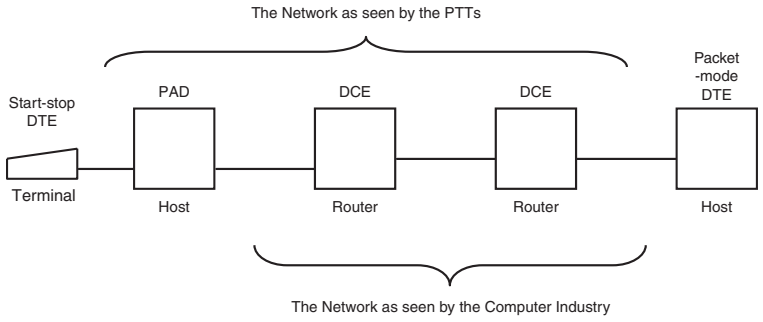

Figure 3-3 Two views of the same thing.

The PTTs called the equivalent of a TIP a *Packet Assembler Disassembler* (PAD). The PAD was not a small host or a switch but an *interface* between start-stop mode DTEs[2] (that is, simple terminals) and packet-mode DTEs (hosts). Why go to all this trouble? Why not just call it a small host? Because that would mean that someone other than a PTT could sell it. PTTs own the interfaces. The PTTs did not want competition in the network access business. Was it really that big a business? At a conference in the late 1970s at the unveiling of their network, the head of one such network when asked what he would think of PAD products being offered by other companies, replied in a moment of frankness, "Not very much!"

However, by defining an asymmetrical architecture, the PTTs essentially created a point product that had nowhere to go, an evolutionary dead end. This distinction between trying to cast what are clearly application functions as "in the network" and having some special status persists to this day. Its manifestation has become more and more interesting as phone companies have become populated with young engineers brought up on the Internet and as the Internet has become more and more dominated by carriers and router vendors trying to escape a business model becoming more and more a commodity. The PTTs were more than willing to consider layers within their interfaces. For example, X.25, the interface between packet-mode DTEs and DCEs defines three layers. The PTTs have consistently attempted to use the beads-on-a-string model to legislate what was "in the network." This approach is found in ISDN, ATM, MPLS, AIN, WAP, and so on.[3] It doesn't take long for even manufacturers of Internet equipment to find that selling on performance and cost is hard work and to begin to advocate more functionality "in the network." (Not long ago, a representative of a major router manufacturer very active in the IETF waxed eloquent about how future networks would have all of this wonderful functionality in them, sounding just like a telephone company advocate of 15 years earlier. Of course, the poor guy had no idea he was parroting views that were an anathema to the whole premise of the Internet.)

The fundamental nature of the beads-on-a-string model provides no tools for controlling complexity or for scaling. The desire to distinguish provider and consumer imparts an inherent asymmetry to the architectures that adds complexity and limits extensibility. We have already discussed some of the general advantages of layering, but there is an additional property that can be quite useful: scoping, which has been defined as the set of nodes addressable without

---

[2] Data Terminating Equipment, not owned by the provider. DCE, Data Communicating Equipment, owned by the provider.

[3] It is striking how similar the first WAP standards of today are to the videotex standards of the early 1980s; e.g., France's Minitel, a similarity that as far as I know went uncommented on.

relaying at a higher layer. In general, the scope of layers increases with high layers. For example, addresses for a data link layer must only be unambiguous within the scope of the layer, not for all data link layers. This provides another tool for controlling complexity and localizing effects, as anyone who has had to contend with a large bridged LAN can attest. Although the layer model avoids these problems, it also has problems. Layering can control complexity up to a point, but it does not address scaling. There has been a tendency to require layers where they are not necessary (causing inefficiencies). Attempts to improve the efficiency of layers by having fewer of them has the effect of reducing the layered model to the beads-on-a-string model or at the very least creating an architecture of "beads with stripes" as we see in today's Internet. But more on this later, now we must turn our attention to the other half of the problem.

The Connectionless/Connection Debate

Background

The preceding section introduced the two major paradigms of networking and began to contrast them. However, that is only half the story. Completely intertwined with the tension between the beads-on-a-string model and the layered model is the connection versus connectionless debate. Given its circuit-switched roots, beads-on-a-string is quite naturally associated with the connection approach, whereas with the popularity of the Internet, layered networks are generally associated with connectionless technologies. However, just to muddy the waters, there are connectionless architectures that try to avoid layers, and layered connection-oriented network architectures, too. In this section, we look at this aspect of the problem in more detail and see how it might be resolved.

Paul Baran, in a RAND Report, proposed a radical new approach to telecommunications (Baran, 1964). Baran's concept addressed the shortcomings of traditional voice networks in terms of survivability and economy. Baran's idea was twofold: Data would be broken up into fairly short "packets," each of which would be routed through the network; if a node failed, the packets were automatically rerouted around the failure, and errors were corrected hop-by-hop. Baran's ideas were the basis for the first packet-switched network, the ARPANET, a precursor to the connection-oriented X.25 networks of the 1970s. However, Louis Pouzin developing the CYCLADES network at IRIA (now INRIA) in France, took the idea one step further. Pouzin reasoned that because

the hosts would never trust the network anyway and would check for errors regardless, the network did not have to be perfectly reliable and, therefore, could be less expensive and more economical. Pouzin called this a datagram or connectionless network. Soon afterward, the ARPANET added a connectionless mode. The datagram network quickly became a *cause célèbre* of the research world, and connectionless became the religion of the Internet. CYCLADES was a research project that despite being an unqualified success and placing France at the leading edge in networking was turned off in the 1970s, primarily because it treaded on the turf of the French PTT. Had it been left to develop, France might have been a leader in the Internet, rather than a straggler. Clearly, some of the most insightful intellectual work was coming out of France. Consequently, the datagram concept had not been very thoroughly explored when CYCLADES ended. It is not clear that when it was adopted for the Internet there was an awareness that deeper investigation was still waiting to be done. In any case, the ensuing connection/connectionless debate created the bunker mentality that ensured it would not get done.

Was Packet Switching Revolutionary?

Interestingly, it depended on your background. If you had a telecom background, where everything was in terms of continuous signals or bits, it definitely was. If your background was computers, where everything is in finite buffers, it was obvious. What could be more simple, you want to send messages, pick up a buffer, and send it.

This proved to be a very contentious innovation. The computer science community embraced the elegance and simplicity of the concept (and its compatibility with computers),[4] while the telephone companies found it an anathema. The ensuing connectionless/connection war has been a Thirty Years War almost as bad as the first one. The war has been complete with fanatics on both sides and not many moderates. The PTTs have been the most ardent supporters of the catholic connection world to the point of being unable to conceive (at least, as many of them contended in many meetings) the existence of communication without a connection. The protestants of the Internet community, representing the other extreme, concluded that everything should be connectionless, while conveniently ignoring the role of connections in their own success. As with any religious issue, the only reasonable approach is to be an agnostic.

The first battle in the war was over X.25. The connectionless troops were late in discovering this push by the telephone companies into data networking and were largely unprepared for the tactics of the Comité Consultatif International Téléphonique et Télégraphique (CCITT) standards process.[5] Strictly

[4] The success of the early ARPANET had as much to do with the level of funding as the success of the technology.

[5] With decades of experience, the ITU participants had developed political maneuvering to a level of sophistication that the largely academic connectionless troops were unprepared for.

Not One, But Two 800-Pound Gorillas

Although not directly related to the problem we must address here, it should be noted that the new network paradigm not only threw a wrench in the PTTs view of the world, but in IBM's as well. Not because it was layered, but because it was a *peer* architecture. IBM's *System Network Architecture* (SNA) was a hierarchical architecture. A peer architecture can always be subset to be hierarchical, but a hierarchical architecture can not be extended to be peer. This new idea in communications had (inadvertently) invalidated the business models of two of the largest economic forces on the planet. This is no way to make friends! The two 800-pound gorillas were really…mad!

It is no wonder there was so much contention in the standards debates. IBM needed to delay as much as possible without appearing to. But for the PTTs, this was a direct threat; they needed to squelch this. But Moore's law was working against both of them. The falling equipment prices favored the new model. Although it was a real possibility that had IBM designed SNA as a peer architecture and subset it for the 1970s market, given its market position, none of this might ever have happened.

speaking, X.25 defined the interface to the network and not the internal working of the network. The initial PTT response to a datagram service in X.25, called Fast Select, was a single packet that opened, transferred data, and closed a connection in a single packet (a connectionless connection and a characteristic of this genre of proposals down to ATM and MPLS). Although a datagram facility was finally included in X.25, it was seldom, if ever, used.

The primary focus of the X.25 debate became whether hop-by-hop error control (X.25) could be as reliable as end-to-end error control or whether a transport protocol such as TCP was always required. The PTTs contended that their networks were perfectly reliable and never lost data, and end-to-end transport was therefore unnecessary. This was an absurd claim that the PTTs never came close to achieving. But the real point was this: Would any IT director who wanted to keep his job simply assume the network would never lose anything? Of course not. So, if the hosts were doing end-to-end error checking anyway, the network could do less error checking.

This brings us to the second problem PTTs had with the layered model. The transport layer effectively confines network providers to a commodity business by essentially establishing a minimal required service from the network (Figure 3-4). Commodity businesses do not have high margins. Much of the purpose of the PTT approach was to keep the computer manufacturers out of the network and the PTTs in a highly profitable business. End-to-end transport protocols and the layered model had the effect of either confining PTTs to a commodity business or creating high-margin, value-added opportunities open to competition—neither of which was appealing to the PTTs. As you will see as we work through this timeline, the debate over whether there should be a transport protocol further reinforces the bunker mentality in the two camps, entrenching a boundary that leaves the last vestige of beads-on-a-string in place and effectively arresting the development of network architecture for the next 30 years.

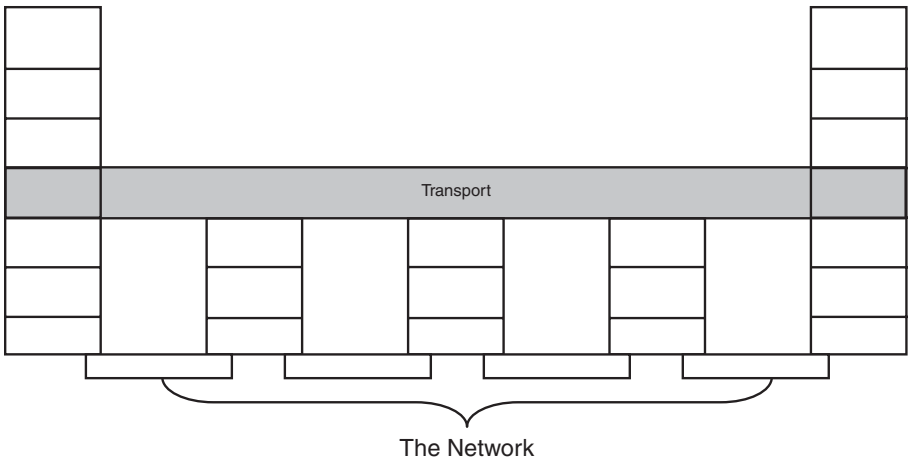

Figure 3-4 The transport layer effectively seals off the network from high-margin applications.

The second and much more contentious battleground for most of this debate was the OSI work. The connectionless proponents began with the upper hand, but the European delegations quickly capitulated to PTT influence,[6] leaving mostly the United States to champion connectionless. The insinuation of the PTT's strong presence was aided by a parochial stance by European computer companies to not adopt anything done in the United States and, thereby, not give the United States an advantage in the market. (Even though at this point, no major commercial vendor was considering product that supported Internet protocols.) The Europeans had a good transport protocol (developed for CYCLADES). This protocol among others, including TCP, had been the subject of an open process conducted by *International Federation for Information Processing* (IFIP) WG6.1 and was selected as a proposed international end-to-end transport protocol (Cerf et al., 1978) in January 1978.

[6] Actually, it was a combination of the PTTs and IBM. IBM chaired and was a heavy influence in all the European ISO delegations. There was an IBM chair in the United States, too, but the U.S. delegation had AT&T and DEC, among others, to balance IBM. The U.S. IBM members for the most part constrained themselves to the administrative issues and stayed away from the technical issues. European computer companies needed détente with the PTTs because they had to coexist with them and as balance against U.S. and Japanese companies. The desire of the European computer companies and PTTs to not simply adopt or improve on what the United States already had, along with others in the United States pushing competing agendas, played right into IBM's strategy. The ultimate example of this came in the battle over a transaction-processing protocol. Let's just say that the Europeans might have done better had they read Uncle Remus.

OSI began a few months later, and the IFIP recommendation was carried into ISO by the IFIP representative and adopted as the basis for Class 4 of the OSI Transport Protocol. But the Europeans insisted on X.25 as the connection-oriented network protocol.[7] After a bitter battle, the United States was able to insert connectionless in the architecture but had to accept constraints that made any interworking of connection mode and connectionless mode impossible. Connectionless could operate over connection mode or vice versa, but there was no means to interwork them as peers. As it turned out, this was less a problem. However, it was enough for the United States to move ahead to develop connectionless routing and data transfer protocols. As the 1980s wore on, X.25 was being relegated to a role as a subnetwork access protocol similar to its use in the Internet in the early 1980s, with connectionless operating over it as a subnetwork independent protocol. The European bunker mentality to not leverage the existing U.S. advances caused them to forfeit any opportunity they might have had to assume a leadership role. The European faith in the ability of centralist initiatives to override consumer demand lost them the war (sound familiar?). The connection/connectionless debate between the United States and Europe and the factions within the United States to dominate the direction of OSI (DEC, IBM, COS, NIST, MAP, and so on) created so much internal strife within OSI that it essentially self-destructed. It lost any cohesive market focus.[8] This left the Internet, a research network, as a production network without having ever been made into a product.

The Internet community (the U.S. and the international academic community) maintained a fairly pure parochial connectionless stance, while the ITU, European computer companies, and PTTs similarly maintained a fairly pure parochial connection

The Battle over Connectionless

As with any consensus organization (standards or legislative), a topic is never really resolved. Issues can always be raised again. Connectionless is a case in point. The United States voted No on the first ballot to move the OSI reference model (RM) to a standard in 1980 in Berlin. The vote was conditional on including connectionless. There was much angst among clueless American commentators who didn't understand that a Yes vote with comment under ISO rules meant that none of your comments had to be accepted; you had voted Yes, after all. Voting No conditionally was the only way to get your comments addressed. (National bodies with active participation in a standard always voted No on condition; that major comments were accepted for just this reason.)

That No vote was resolved with an agreement to develop a connectionless addendum to the OSI RM. There then ensued three years of meeting two or three times a year for a week at a time to hammer out a document that added connectionless to the RM. Every word was contested and argued

continues

[7] It is hard to believe in retrospect that as late as 1990, Europeans were still expecting X.25 to be the future of networking. How X.25 could have supported the bandwidths and applications is hard to imagine.

[8] By 1978, France, Germany, and the United Kingdom had all produced high-level government reports focusing on networking as a major future technology, and considerable funding followed soon after. However, all attempts to build a "European ARPANET," although technically successful, were never able to survive PTT pressures. In the end, the difference was not technological but that the United States was willing to provide far greater subsidies (through the DoD) for the Internet (and AT&T and IBM never saw it as compitition), while Europe more or less allowed the market to decide. The greater subsidies by the DoD and NSF allowed the Internet to achieve critical mass, while the competitive marketplace was still developing. However, these political and economic susidies were not accompanied by a commensurate transition from research demo to product. Leaving the result lacking critical structures.

stance. The European standards stance had a majority connection-oriented bias rooted in the influence of the PTTs and a pragmatic stance by some in the computer industry that the PTTs were part of the reality. The U.S. standards group was primarily connectionless with a connection-oriented tendency by the IBM participants. (It must also be recognized that in the 1970s and 1980s the split between connection and connectionless was to a large extent, but not entirely, generational. This is no longer the case.) Thus, the protectionist attitudes toward the status quo in Europe ultimately undermined Europe's strategy for economic parity (sound familiar?). The connectionless forces were woefully outnumbered. The "solution" was less than satisfactory and, in fact, never really resolved. The Europeans attempted to legislate connectionless either out of existence or constrain it to such a degree to make it useless. (There are arbitrary statements in the OSI RM limiting the use of connectionless.) The solution in OSI can only be likened to a ceasefire along a demilitarized zone rather than a real solution. By the same token, this was the forum where the concepts were given the greatest scrutiny. But, the scrutiny led to only heated debate and an uneasy truce, not to any deeper understanding.

Connectionless was seen (by both sides) as having no shared state and requiring no establishment phase and was, therefore, fundamentally simpler, cheaper, more flexible, and elegant. Connections were seen by the connectionless crowd as too much unnecessary overhead and too much state to store for simple store-and-forward networks and by the connection-oriented crowd as the only way to provide any reasonable service.[9] The connection proponents forced the connectionless proponents to recognize that there was some shared state in connectionless that had to be created before communication could begin. But that (thought the connectionless types) was really minor stuff that could be ignored; connectionless was really different from having a connection! Although the connectionless advocates convinced the connection advocates (also known as bellheads) that connectionless did have advantages in routing and recovery from failures, everyone *knew* that any real network had to

continued

over by the Europeans. Every word and phrase was inspected to ensure that it did not give an advantage to one side or the other. Finally in 1983, in Ottawa, the document was ready to be progressed to a standards vote. The chair of the Architecture Working Group found me at lunch in the food court of the Rideau Center and said he wasn't sure the votes were there for it to pass. As Head of Delegation for the WG, this was my problem. We had an agreement, but that was three years ago. (Because the Chair was a connectionless advocate, he was sympathetic to our cause, but there was only so much he could do.) We had negotiated everything there was to negotiate. Finally, it came down to no connectionless addendum, no further U.S. participation.

It was passed and became an approved standard two years later. However, the issue was never really resolved and continued to be contested in every meeting.

[9] On the other hand, the PTTs insisted that no end-to-end reliability was required because their networks were perfectly reliable, even though everyone knew better. Here they insisted on the use of TP0 (at the transport layer because there won't be any errors), whereas in the application layer where no one will notice. There was Reliable Transfer Session Element (RTSE) (X.228), a transport protocol, pretending to be an application protocol. This was even found in Wireless Access Protocol (WAP) 1.0!

have connections. How else could you charge for usage? In particular, both groups saw the interface provided at the layer boundary as being very different for the two. Below the transport layer, existing network architectures kept the two very distinct, if they even considered both.

Fairly early in this debate, the moderates thought they saw a resolution by viewing connectionless and connections as simply two extremes on a continuum of a single property: the amount of shared state necessary among the protocol machines. Connectionless is not, as some have contended, no shared state; instead, it is a minimal shared state. However, the two models seemed so radically different that it seemed impossible to find a unified model that was a true synthesis that didn't just push the two techniques together. There seemed to be no way to make them interwork seamlessly. It seemed that there wasn't much of a continuum but more of a dichotomy: There was either very little state (as in IP) or a lot (as in X.25, TCP, and so forth) and few, if any, examples in between. What weren't we seeing?

Finding for a Synthesis: The Easy Part

In all architectures, these were represented as very different services; that is, they presented very different interfaces to the user, by very different mechanisms. Clearly any resolution would have to solve two problems:

1. A unified approach of the service. (The external "black box" view must be the same.)
2. A unified approach to the function, a single mechanism that encompasses both extremes.

The primary characteristic that any unified model would have to have would be that the behavior seen by the user would be the same whether using connectionless or connections (a common interface). As always, a good interface can cover up a multitude of sins, regardless of how successful we are at the second part of the problem. For connectionless communications to be possible, the sender must have some reason to believe that there will be an instance of the protocol associated with the destination address that will understand the message when it is received and some binding to a user of the protocol at the destination so that data can be delivered to someone. Hence, there are procedures that must be performed for this to be the case. This does not entail the exchange of protocol. Both connectionless and connections require some sort of setup. Both require that first, the addresses on which they are willing to communicate be made known. This is done in the enrollment phase. Then, when the user is

ready to send or receive data, resources need to be allocated to support the communication, and bindings must be made between an application and a lower-layer protocol machine. The only way to get a common interface would be for both to present the same behavior to the user.

Depending on the traffic characteristics, the operation of the layer can be made more effective and efficient if the layer maintains more shared state information to better control the rate of transmission and to recover from errors. The more of this shared state information maintained, the more connection oriented the communication. To some extent, it becomes a trade-off between putting information in the header of each message (and consuming more bandwidth, but less memory) and associating more state information with the protocol (and consuming more memory but less bandwidth). In any case, managing this shared state occurs entirely among the cooperating protocol state machines and is not visible to the user. Connectionless and connections are functions within the layer. The user of the service has no need to know which mechanisms are used by the protocol machines, only the resulting characteristics seen by the user. The internal behavior of the layer should not be visible outside the black box.

What's in a Word?

Words make a difference. They affect how we think about something. The terms chosen to describe a concept are a crucial part of any model. The right concepts with terms that give the wrong connotation can make a problem much more difficult. The right terms can make it much easier. Adopting the mindset of the terms may allow you to see things you might not otherwise see.

Given that some setup is required, even if PDUs are not generated, implies that a common interface behavior would resemble behavior of the connection interface (that is, create, send/receive, and delete). However, it does not have the sense of creating a "connection," so the common terms *connect* and *establish* seems wrong. A more neutral concept that truly leaves the decision of the mechanism internal to the layer is needed. Let's go back to the roots of networking to look for analogs in operating systems to find an appropriate abstraction of the operation we are performing. The user is requesting the service below to "allocate" communication resources, just as one requests an operating system to allocate resources (and bandwidth is just the first derivative of memory). *Allocate,* however, does not imply whether there is a *connection*.

Thinking in terms of allocate makes clear what probably should have been clear all the time: The layer should decide whether or not it uses a connection, not the user of the layer. The user should not be choosing how resources should be provided, but what characteristics the resources should have.[10] The user should be requesting communication resources with certain characteristics:

10 This is at odds with the current fad that the user should be able to control the mechanisms in the network. This is not unlike old arguments for assembly language because more control was needed than the compiler allowed. It is even less true here.

bandwidth, delay, error rate, and so on. It is the layer's task to determine how it should satisfy the request, given the characteristics from its supporting service and all other requests. How it is done is of no concern to the user. Working from the concept of allocate, it becomes apparent that the choice is a trade-off between static and dynamic allocation of resources; we are throwing away a ladder:

The more deterministic (less variance), the more connection-like and static the resource allocation;
The less deterministic (greater variance), the more connectionless and dynamic the resource allocation.

This fits our experience. As traffic becomes denser (that is, nearly constant), connections are more effective. When traffic is more stochastic, connectionless makes more sense. As one moves down in the layers and/or in from the periphery toward the backbone, one would expect traffic to shift from being less dense to more dense and, therefore, from being more connectionless in nature to more connection-like. To insist that either connectionless or connections is best in all circumstances is foolish.

Generally, connections are associated with greater reliability, but reliability is a separate consideration. The reliability of a communication is only one aspect of the characteristics associated with connections. Simply requiring more than best effort is not necessarily a reason to choose a connection. The choice to be made within the (N)-layer depends on the difference between what is requested by the (N+1)-layer and what is provided by the (N–1)-layer. It is quite possible to have a very reliable (N–1)-layer, and so the (N)-layer might use a connectionless mechanism when high reliability is requested by the (N+1)-layer. Or the (N+1)-layer might have requested ordering, and the (N–1)-layer was highly reliable but did not provide ordering. So, the (N)-layer would need a weak form of connection to satisfy the allocation requested. Reliability is not always the deciding factor, but traffic density is.

Connectionless and connection oriented are a characterization of functions chosen depending on the traffic characteristics and the QoS desired. They are not themselves traffic characteristics or QoS or anything that the user needs to be aware of. They are mechanisms that may be used to provide specific traffic characteristics. Every communication must be enrolled and must request an allocation of resources and only then may it send and receive data.

Now we need to address the second part of the problem: finding a model that unifies the function of connectionless and connection. This is a much harder problem. Solving the first one was relatively easy. We essentially created a rug to sweep the mess under. Now we have to deal with the mess! However, we'll need to put this off until after we have worked out more of the structure of protocols.

The Types of Mechanisms

A protocol mechanism is a function of specific elements of PCI (fields) and the state variables of the PM that yields changes to the state variables and one or more PDUs. These elements of PCI are conveyed to the peer PM(s) by one or more PDUs to maintain the consistency of the shared state for that mechanism. As we have seen, some mechanisms may use the same elements. (For example, sequence numbers are used both for ordering and lost and duplicate detection.) But is there more to say about the types of mechanisms in protocols?

To investigate the structure of protocols and the effect of separating mechanism and policy, I did a gedanken experiment in the early 1990s. In the 1970s, we had developed a standard outline based on the finite state machine (FSM) model for informal prose specifications of protocols. For each PDU type, the conditions for generating it and the action upon receipt are described. Carefully following the outline proved to be quite effective. For the experiment, the outline was extended to accommodate adding a new mechanism to a protocol (see Appendix A, "Outline for Gedanken Experiment on Separating Mechanism and Policy"). Then, in strict accordance with the outline, the mechanisms of a typical transport protocol were specified. Some of the results of this experiment were discussed in Chapter 2, "Protocol Elements." Looking at the result, one finds that the fields of the PCI associated with the mechanisms naturally cleave into two groups:

- Tightly bound fields, those that must be associated with the user-data (i.e., the Transfer PDU)
- Loosely bound fields, those for which it is not necessary that the fields be associated with the user-data

For example, tightly bound fields are those associated with sequence numbers for ordering, or the CRC for detecting data corruption that must be part of the PCI of the Transfer PDU. Loosely bound fields are associated with synchronization, flow control, or acknowledgments may be, but do not have to be, associated with the Transfer PDU. A protocol may have one PDU type, with all PCI carried in every PDU (as with TCP), or more than one (as with XNS, TP4, SCTP, or X.25).

Therefore, we can define the following:

- A tightly coupled mechanism is one that is a function of only tightly bound fields.
- A loosely bound mechanism is a function of at least one loosely bound field.

How Many PDUs in a Protocol?

One of the major (and much argued about) decisions to be made in the design of a protocol is the number and format of the PDUs. We know that there must be at least one PDU to carry the user's data, which we will call the Transfer PDU,[11] but how many others should there be? Beyond the considerations discussed in the preceding section, there would seem to be no architectural requirements that would require multiple types of PDUs. There are engineering constraints that would argue in specific environments for or against one or more PDU types to minimize the bandwidth overhead or processing.

An oft-quoted design principle recommends that control and data be separated. The natural bifurcation noted previously would seem to reinforce this design principle. Because a PDU is equivalent to a procedure call or an operator on an object, associating more and more functionality with a single PDU (and the Transfer PDU is generally the target because it is the only one that has to be there) is "overloading the operator." TCP is a good example of this approach. Minimizing the functionality of the Transfer PDU also minimizes overhead when some functionality is not used.[12] However, there is nothing in the structure of protocols that would seem to require this to be the case.

For data transfer protocols, the minimum number of PDU types is one, and the maximum number would seem to be on the O(m+1) PDU types—that is, one Transfer PDU plus *m* PDU types, one for each loosely bound mechanism—although undoubtedly, some standards committee could define a protocol that exceeds all of these bounds). For most asymmetric protocols, the maximum may be on the O(2m), because most functions will consist of a Request and a Response PDU. In symmetric protocols, the "request" often either does not have an explicit response or is its own response. Hence, there are O(m) PDU types.

Good design principles favor not overloading operators. It would follow then that there should be a PDU type per loosely coupled mechanism. Separate PDU types simplify and facilitate asynchronous and parallel processing of the protocol. (This was not a consideration with TCP, which in 1974 assumed that serial processing was the only possibility.) Multiple PDU types also provide greater flexibility in the use of the protocol. Loosely coupled mechanisms can be added to a protocol so that they are backward compatible. (For a protocol with one

11 The names of PDUs are generally verbs indicating the action associated with them, such as Connect, Ack, Set, Get, etc. Keeping with this convention, we will use Transfer PDU, completely aware that in many protocols it is called the Data PDU.

12 Overloading the Transfer PDU but making the elements optional may minimize the impact on bandwidth but increases both the amount of PCI and the processing overhead when they are.

A Closer Consideration of the Number of PDUs in a Protocol

There were two major arguments for a single PDU format in TCP: 1) ease of processing and 2) piggybacking acks. Piggybacking acks reduce the bandwidth overhead over the life of the connection. This is a classic engineering trade-off, whether to incur a little more overhead in some PDUs for less overhead overall (global efficiency of the connection) or have more smaller PDUs (local efficiency, but) with perhaps more overhead over all. In any protocol, each PDU type will have several common PCI elements, X, that must be present (addresses/socket-ids, opcode, and so on). With multiple PDU types, for our purposes, let's assume a Transfer PDU, T, and an Ack PDU, A, and then a protocol with different types of PDUs will have to generate XT and XA PDUs, in contrast with a protocol with one PDU type, which will generate XAT, potentially a savings of X bytes.

In the 1970s, measurement of network traffic found that piggybacking acks would reduce overhead by 30% to 40%. To understand why, consider that the vast majority of Net traffic in the 1970s was Telnet connections between terminals to remote hosts with the minority traffic being FTP for file and mail transfers. The most prevalent system on the Net was the BBN Tenex. Tenex insisted on character-at-a-time echoing over the Net. This generated many very small PDUs: The Telnet user would send a one-character packet with a one-character echo from the server, both which had to be ack-'ed. The length of an input string averaged about 20 characters (each one ack'ed) with an average of a 40-character output string, which would be in one message with one ack.

A quick back-of-the-envelope calculation will show that if T is 1 byte (character at a time), and X is bigger, at least 8 bytes (2 socket-ids, an opcode, and probably a couple of other things), and A is 4. This alone accounts for a 35% savings in the character echoing and the "efficient" response in one message. File transfers and such and Telnet connections that used local echo would not see this gain as much because there was more data in the packets and fewer acks were required. This traffic would bring the overall improvement down to the 20% to 30% range that was observed. There was considerable overhead per character sent and piggybacking acks provided a significant savings. Because each PDU required two address fields, simply concatenating a Transfer and Ack PDU (that is, XTXA) still incurred considerable overhead.

However, today's network traffic is not dominated by Tenex Telnet sessions. Even if we assume Request/Response pairs averaging 20 and 40 bytes, this reduces the advantage to roughly 10%. If Request/Response pairs are 100 bytes, the advantage is reduced to roughly 1%. So the advantage of piggybacking acks quickly becomes negligible. Given that Internet traffic data shows significant modes at 40 bytes (syns and acks with no data), 576 bytes (default maximum unfragmented size) and 1500 (Ethernet frames) would seem to imply that piggybacking is not as advantageous today as it was in 1974.

There is also little advantage to a single header format. The opcode of the multiple PDUs is replaced by control bits. Each control bit must be inspected in every PDU even if the state of the PM would indicate that some bits should not be set. This creates more processing overhead rather than less. The solution as proposed in several papers on code optimization of TCP is to treat the control bits as an opcode! If one looks at the implementation optimizations proposed for TCP in these papers (Clark et al., 1989), one finds that they fall into two categories:

1. They are equally applicable to any protocol, or
2. They make the single-format PDU protocol (e.g., TCP) more closely emulate a protocol with multiple PDU formats (e.g., TP4).

But even this leaves considerable discretion to the protocol designer. For example, are acknowledgment and negative acknowledgment separate mechanisms or the same mechanism that differ only in a single bit? Are data-based flow control and rate-based flow control different mechanisms or the same mechanism only differing in the units used for credit? Probably not worth worrying about. What PDU types will be generated most often by the traffic seen in the network?

PDU type, adding a mechanism will generally require changing the one PDU format.) Similarly, a mechanism can be made optional by simply using a policy that never causes the PDUs to be generated. (For a protocol with a single PDU type, the PCI elements must be sent whether they are used, or a more complex encoding is required to indicate whether the elements are present.) With multiple PDU types no overhead is necessary to indicate a PDU's absence. From this it would seem that more rather than fewer PDU types would generally be preferred.

Does this and the discussion in the sidebar mean that protocols should always have more than one PDU type? Not necessarily. This is not a case of right or wrong, true or false. The choices made are based on the requirements of the operating environment. As we have seen, TCP was designed for an environment with a very large proportion of very small PDUs. Clearly, the choice was correct; it saved 30% to 40% in bandwidth. In an environment that does not have a lot of small PDUs, and especially one with a large range of traffic characteristics, optimization for small PDU size loses its weight while the need for greater flexibility gains weight, indicating that under different conditions a different solution might be more appropriate.

Contrary to the often knee-jerk response to these decisions, the choice of a mechanism is not right or wrong but a case of appropriate boundary conditions. It is important to remember the conditions under which these sorts of choices are appropriate. Although it might seem unlikely that network requirements will ever return to the conditions of remote character echoing, history does have a way or repeating itself, although usually in a somewhat different form.[13]

The Types of Protocols

If we consider the list of mechanisms in Chapter 2, several patterns begin to emerge. First, there are mechanisms that might appear in any protocol, such as delimiting, allocation, policy negotiation, data-corruption detection, and so on. The other pattern that seems to emerge is the similarity in transport protocols and data link protocols. They both are primarily concerned with end-to-end error and flow control. It is just that the "ends" are in different places; they have different scopes. Similarly, network and MAC protocols are similar in that they primarily deal with relaying and multiplexing. But also, in the relaying and

[13] Which is what makes the topologist's vision defect so useful. These vision-impaired architects are able to recognize when this new doughnut looks like that old coffee cup!

multiplexing protocols, policy is always imposed by the sender; and in the error- and flow-control protocols policy is always imposed by the receiver: They are *feedback* mechanisms.

The distinction of loosely coupled and tightly coupled shared state is more definitive, not tied to qualitative ideas of more versus less or loose versus tight or hard versus soft shared state or the effects of lost state, but to an observable property: the presence of feedback. The tightly coupled protocols have feedback mechanisms; the protocols with more loosely coupled shared state have no feedback mechanisms. The operation of protocols with no feedback is less effected by inconsistent state than protocols with feedback. Thus, the distinction between flow and connection that has been found useful in finding a middle ground in the connection/connectionless controversy characterizes the presence or absence of feedback in a protocol. Connections include feedback mechanisms; associations and flows do not. Similarly, the more robust three-way handshake allocation mechanism is required for protocols with feedback (a two-way handshake for protocols with no feedback).

The astute reader will also have noticed one other pattern: These two types of protocols tend to alternate in architectures. The MAC layer does relaying and multiplexing, the data link layer does "end-to-end" error control; the network layer relays, the transport layer does end-to-end error control; mail protocols relay, hmm no end-to-end error control and sometimes mail is lost. We will consider this in greater detail in Chapter 6, "Divining Layers," but for now we can make two observations:

1. Relaying always creates the opportunity for PDUs to be lost. Therefore, to guarantee reliability, there must always be an error-control protocol on top of a relaying protocol.
2. This would seem to indicate that there are really only three fundamental types of protocols:
   - **Two data transfer protocols:** Relaying and multiplexing protocols and error- and flow-control protocols with different policies
   - **Application protocols**

To avoid repeating cumbersome phrases, error- and flow-control protocols will be often referred to as error-control protocols, and relaying and multiplexing protocols as relaying protocols. Keep in mind, however, that the other aspect is always there.

Policies will be chosen to optimize the performance based on their position in the architecture. Protocols nearer the application will have policies suited to the requirements of the application. Because there are many applications over these protocols, we can expect them to use a wider variety of policies. Protocols nearer to the media will be dominated by the characteristics of the media. Since these protocols have less scope and are specific to a specific media, we can expect them to use a smaller range of policies. Protocols in between will be primarily concerned with resource-allocation issues. Combining this with our observations about connection and connectionless, we can expect more connection-oriented protocols as we move toward the core and more connectionless protocols toward the edge.

The functions of multiplexing, routing, and relaying provide numerous opportunities for errors to occur. Analysis and experience has shown that relaying (that is, hop-by-hop protocols) can never be absolutely error free and that an error-control protocol operating with the same scope as the relaying layer is required to ensure error-free operation between a source and a destination. This was the great X.25 versus transport protocol debate of the mid-1970s and early 1980s.

This alternating of protocols is seen in traditional best-effort networks: The data link protocol provides "end-to-end" error control for relaying by physical or MAC protocols. The transport protocol provides "end-to-end" error control for relaying by the network protocols; one often resorts to "end-to-end" methods to ensure that relaying mail has been successful and so forth. This separation is seldom completely clean in existing protocols. In fact, we even see protocols designed with multiple roles in the same protocol.

Although there is nothing inherently wrong with two adjoining error-control protocols or two adjoining relaying protocols, there are strong arguments against such configurations. Two adjoining error-control protocols is fairly pointless because the scope of the two layers must be the same. (There has been no relaying that would increase the scope.) Unless the first protocol is relatively weak, there should be no errors missed by the first one that will not be detected or corrected by the second. This is essentially doing the same work twice. If there are such errors, the second protocol should be used in place of the first. In a legacy network, however, it might not be possible to do anything about the existence or absence of the first protocol, in which case the second protocol may be necessary to achieve the desired QoS. This is probably the only instance in which one should find two adjoining error-control protocols. Therefore, we should conclude that the two kinds of protocols should alternate in the architecture, and if they don't, it is probably an accident of history.

The fundamental nature of relaying protocols is to be always treading on the edge of congestion, and PDUs are lost when congestion cannot be avoided. Two adjoining relaying protocols would tend to compound the errors, thereby decreasing the NoS of the second relaying protocol and thus impacting the QoS, and the performance, that an eventual error-control protocol could achieve. In addition, two adjoining relaying protocols will usually (but not always) imply that the (N+1)-protocol has wider scope than the (N)-protocol.

It is generally prudent (more efficient, less costly, and so on) to recover from any errors in a layer with less scope, rather than propagating the errors to an (N+1)-protocol with a wider scope and then attempting to recover the errors. If the probability of the relaying protocol introducing errors is low, an intervening error-control protocol may be omitted. For example, Ethernet LANs and related technologies are generally deemed sufficiently reliable that no "end-to-end" error control is necessary. For a wireless environment, however, the opposite could be the case. By the same token, the error-control protocol does not have to be perfect but only provide enough reliability to make end-to-end error control cost-effective. For example, the data link layer might tolerate an error rate somewhat less than the loss rate due to congestion at the layer above. However, these sorts of decisions should be based on measurements of the live network rather than on the hype of marketing or the conviction of the designers.

An application protocol can operate over any data transfer protocol. An application protocol can have direct communication only with systems within the scope of the next-lower layer. However, an application protocol operating over a relaying protocol will be less reliable than one operating over an error-control protocol with the same scope (for example, mail).

Relaying and error-control protocols have been embedded into applications. The examples are legions of people arguing that the overhead of a transport protocol is just too great for their application. But then, they are inexorably drawn through successive revisions, fixing problems found with the actual operation of their application until they have replicated all the functionality of a transport protocol but none of its efficiency because their application was not designed for it from the beginning. We have already alluded to the relaying protocol that is part of a mail protocol such as *Simple Mail Transfer Protocol* (SMTP). This application could easily be partitioned into a relaying protocol that did not

Why We Can't Design the Perfect Transport Protocol

There has been much discussion and many proposals for new and "better" transport protocols over the years to replace TCP—none with very much success. There was always some major category of traffic that the proposal did not handle well. On the other hand, we have been fairly successful with new data link protocols.

The separation of mechanism and policy makes the reason for this clear. Transport protocols are intended to support the requirements of their applications. There are roughly six or eight mechanisms in a transport protocol. By not separating mechanism and policy, we have (unintentionally) been saying that we expect to find a single point in an eight-dimensional space that satisfies all the requirements. When put this way, this is clearly absurd! There are no panaceas, but there is a lot of commonality! If we separate mechanism from policy, the problem can be solved.

Why was there more success with data link protocols? Data link protocols are tailored to address the requirements of a particular physical medium. Hence, their performance range is narrower. Thus, a single set of policies will satisfy the problem.

differ much from those found below it and the actual mail application. The traditional checkpoint-recovery mechanism found in many file transfer protocols is an error-control protocol. Less obvious is the traditional *Online Transaction Processing* (OLTP) protocols, which involve a two-phase commit. OLTP can also be implemented as a form of data transfer protocol with different policies.

The effect of separating mechanism and policy is to separate the invariances (the mechanisms) in the structure of the protocols from the variances (the policies). From this we see that there seems to be fundamentally two kinds of data transfer protocols: a relaying protocol and an error-control protocol. The major differences in the examples of the two sets are the differences in syntax and the requirement for fields of different lengths. At low bandwidth, large sequence numbers add too much overhead to the PCI and are unnecessary. At very high bandwidth, even longer sequence numbers might be required.

This seems to be a rather minor difference. Can we make it an invariant, too? A self-describing syntax (that is, tag-length-value) would incur too much overhead for protocols operating in performance-sensitive environments. In any case, such flexibility is not required at execution time in data transfer protocols. One could have the "same protocol" (that is, procedures) but a small number (three or four) of different PCI formats. These PCI formats would be very similar, only the lengths of the PCI elements would be somewhat different; and for data transfer protocols, the only fields that might vary are those related to addresses, sequencing, and flow control. It should be possible to define the protocols such that they are invariant with respect to syntax. (Addresses are just used for table lookups, and sequence numbers only differ in the modulus of the arithmetic.) Greater commonality in protocol behavior could have a beneficial effect on network engineering.

So, there actually are only two data transfer protocols described in terms of a single abstract syntax with a few concrete syntaxes. The only question is the degree to which various media-specific characteristics and other "bells and whistles" are necessary or whether they are part of some other functions?

The Architecture of Data Transfer PMs

A PM consists of actions associated with inputs, either from PDUs, from local timeouts and system calls for buffer management, or from the upper or lower interfaces whose sequencing is controlled by a state machine. The distinction in the types of mechanisms and protocols allows us to say more about the architectural structure of the protocol machines for data transfer protocols. Actions are divided between mechanism and policy. Further, the patterns, which we have

seen in the mechanisms, are not just a nice taxonomy but have implications for implementation. Clearly, some mechanisms must be done before others. For example, the CRC must be done before anything else in the PDU is looked at, lost and duplicate detection must be done before retransmission control, and so on. For now, let's ignore the normal engineering constraints and the constraints of serial processors and consider the amount of asynchrony inherent in a data transfer protocol. Let's follow the pattern we have seen of loosely coupled and tightly coupled mechanisms.

If we separate loosely bound mechanisms to the right and tightly bound mechanisms to the left (see Figure 3-5), a much more interesting thing happens. The left side consists of the processing of the Transfer PDU and consists only of fragmentation/reassembly, ordering, and queuing. The right side handles synchronization, flow control, and acknowledgments. CRC checking and delimiting is done in common to all PDUs. The left side has little or no policy associated with it, other than whether simple ordering is on or off. The connection-id (or port-ids) select the state vector. All operations are simple operations based on the sequence number and SDU delimiters and the queuing of the PDUs. The right side is all policy. The right side maintains the state vector and executes the policies for retransmission control (that is, acking and flow control). The right side requires general-purpose calculations; the left side requires much less general computation, just straightforward queuing and ordering algorithms. Others have noted a similar dichotomy but ascribed it to merely an artifact of certain implementation-specific considerations. Here we see that it underlies a much more fundamental structural property of protocols.

But what is striking is that the two sides are not at all tightly coupled! In fact, they are virtually independent. The only effect the right side has on the left is to start and stop queues and infrequently discard one or more PDUs from a queue, and it probably isn't too particular which ones. Furthermore, we have made no assumptions about the protocol other than to assume the mechanisms it should contain. The design implied by this diagram supports either an in-band or out-of-band design simply by mapping the left and right side to the same or different output queues. It also allows retransmission (acks) and flow control to be completely asynchronous from the data transfer. Protocol processing has generally been seen as serial in nature (for instance, TCP); an implementation along these lines exhibits a high degree of parallelism. This approach could be pipelined and would seem to indicate significant improvements in protocol performance. This is not the place to consider this in detail, but it should be mentioned that security mechanisms fit nicely into this architecture, too.

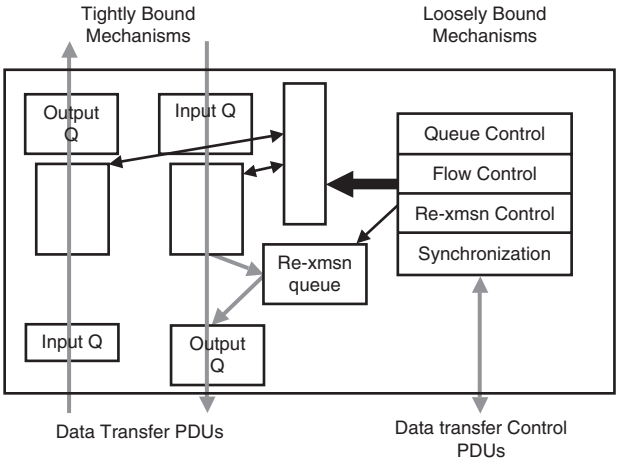

Figure 3-5 Error- and flow-control protocols naturally cleave into a data transfer part with only tightly bound mechanisms and a control part with only loosely bound mechanisms.

Clearly, the optimal protocol design to take advantage of this would have separate PDUs for the loosely coupled mechanisms, which would be processed by the right side and a Transfer PDU that was processed by the left side. This would maximize the parallelism in the processing and the parallelism in the interactions with the remote PM. (This is not a hard requirement. This approach could be used with a protocol that did not have separate PDUs, but more complex hardware would be required to duplicate headers and route them. One would be making the mistake that OSI made of allowing desire to go against the grain of the problem, its nature. Nature generally gets its retribution.)

As we have seen in the past, there has been a school of thought that tried to minimize the number of PDU types. This was based on assumptions unique to software and hardware constraints and traffic characteristics of the mid-1970s. A single PDU type simplified processing and allowed acks to be piggybacked with the return traffic. These conditions are no longer relevant. This additional overhead just to carry a 16- or 32-bit sequence number was rightfully considered excessive. From the principles we have developed here, we can see that there is no multiplexing in an error-control protocol, so such overhead is virtually nonexistent, and the other conditions that held then no longer exist. If the loosely coupled mechanisms are implemented with separate PDUs, they become degenerate cases. If the policies for them are null, these PDUs are simply never sent. There needs be no explicit action to not accommodate them.

This also begs the question as to whether this is one protocol or two. Protocols with only tightly bound mechanisms need only a two-way handshake. Such a protocol would have the left side of Figure 3-5 and only state initialization on the right. Protocols with loosely bound mechanisms would use a three-way handshake and have more functionality on the right side of Figure 3-5 for the bookkeeping to maintain shared state. Even more intriguing is the fact that if one looks at the format of the Transfer PDU with just tightly bound mechanisms, one finds that it bears a strong resemblance to UDP-like protocols. The only difference being whether the "message-id" field is interpreted as a sequence number. This raises the possibility of a single protocol structure accommodating the entire range from pure connectionless to full connection by merely changing policy.

Finding a Synthesis: The Hard Part

Earlier in this chapter, we considered the problem of finding a synthesis of the *connection/connectionless* (co/cl) schism that has plagued networking from its beginnings. We identified that there were two parts of the problem: a common interface model and a common functional model.

We need both to accommodate co/cl as degenerate cases. For the service/interface model, we abstracted the problem into a resource-allocation model and required the user to specify the parameters of the communication being requested so that the choice is inside the black box. The first part was relatively easy to solve. (Sweeping a problem under the rug always is.) Now we must look at the more difficult problem: finding a common model for the function of connection and connectionless.

Generally, the difference between co/cl has been characterized by the amount of shared state and degree of consistency (that is, how tightly coupled the state is required to be for correct operation). In this view, co/cl are extremes along a continuum from less state loosely coupled to more state tightly coupled. If this is the case, there should be other operating points along the continuum.

Analysis of this problem begins quite early with Belnes's analysis (1976) of the requirements to deliver a single message reliably. He concludes that a five-way exchange will reliably deliver a single message even if either host resets. The four-way exchange will suffice as long as neither host fails. This gives some idea of the amount of shared state required for reliable delivery. Watson (1981) developed a timer-based approach to a transport protocol based on the rather ingenious perspective that all connections exist and have existed for all time. We maintain state information about them in a cache only when they are active. The exchange of PDUs merely serves to refresh the cache, and when activity

ceases for some time, we no longer keep the cached information on the exchange. This yields a very robust and simple approach to a transport protocol. But all of these techniques are concerned with creating the reliable delivery of data over an unreliable network. Watson is able to prove that the timers in delta-t are both necessary and sufficient. Watson's delta-t protocol warrants far more attention than it has received.

Lamport et al. (1982) analysis of the Byzantine Generals problem somewhat confuses the issue with an unfortunate analogy that proves an interesting puzzle. For networking, the result of this analysis is that within a single protocol operating over an unreliable media, it is impossible to determine whether the last message arrived at its destination. This is not the same thing as saying that the generals can't know whether an ack was received, or perhaps more precisely it does if the generals fail to construct a proper communication architecture.

This is a beads-on-a-string view of the problem predicated on overloading the semantics of the acknowledgment so that it acts both to acknowledge the conversation of the generals and to acknowledge the delivery of messages. In a properly layered architecture with explicit establishment and release at both the application and transport layers, the generals can know the state their conversation terminated, although it is not possible to know whether their reliable communication channel terminated correctly. Here, the general's last message is not the last message sent. In other words, we can't know whether the transport connection terminated correctly, but we can know whether the application layer protocol terminated correctly. After that is done, we don't care whether the transport connection terminates correctly. Note that with a protocol like Watson's delta-t with no explicit establishment and release this won't be the case. However, if the connection/release discipline is followed by the application layers, the generals can know whether they reached agreement and whether the application connection terminated correctly. They still won't know whether the last message (a disconnect ack of some sort) arrived.

Spector (1982) considers the same problem as Belnes from a somewhat different perspective. Rather than consider whether an exchange is reliable and under what conditions it is not, Spector considers the problem from the other end, so to speak: Given a particular exchange, what can we say about whether an operation took place? Spector considers the perspective of request/response PDUs for client/server applications (just becoming popular then) and considers the semantics of the reliability of a remote operation performed in the presence of lost messages (but not duplicates):

- "Maybe," a single PDU with no response
- "At least once," a simple request response (that is, two-way handshake)

- "Only once," a three-way handshake, which has two subtypes depending on failure cases

Clark (1988) characterizes the shared state problem in more qualitative terms of soft versus hard state and whether failures are subject to "fate-sharing." Clark contrasts replicating state with fate-sharing. The abstraction he is contrasting is the hop-by-hop nature of X.25 with the end-to-end error control of TCP, TP4, or delta-t. Today, we would compare the TCP/IP approach with ATM or MPLS. The X.25 protocols are definitely more complex, but the reason is somewhat counterintuitive. Although Clark's point is fundamentally correct, this comparison is significantly different from the case Spector was considering or the case below that applies Clark's soft state concept to signaling protocols. But to see it, let's first consider later attempts to formalize the soft state concept.

While intuitively appealing, the soft state idea has resisted formalism until recent attempts: (Raman and McCanne, 1999; Ping et al., 2003; and others). But here again, the gradations of shared state are found in application protocols (for example, signaling and peer-to-peer (sic) protocols, not so much in data transfer protocols). Ping et al. describe five cases of decreasing "softness" between pure soft state and hard state. A careful look at them in comparison with what we have already looked at is helpful but that the results here assume that the only operation being performed is "replacement," not any operation that depends on the previous state (that is, increment or decrement):

1. The first case is *pure soft-state* (ss), which is described as a sender sending a trigger PDU to a receiver. The sender sets a timer and refreshes the state whenever the timer expires by sending the current value of the state information in a new trigger PDU. The receiver records the content of the trigger PDU when it arrives and sets its own timer, which is reset whenever a new message arrives. If the receiver's timer expires, it deletes the state.

   This characterization is a weak form of Spector's maybe case with Watson's view of connections. (We are assuming that "trigger" and "request" are roughly equivalent, although recognizing not entirely equivalent.) The receiver may not receive any of the sender's PDUs. Also, Ping distinguishes a "false removal" of state when the receiver's timer expires before a new trigger PDU arrives. This constitutes looking at the system from the point of view of an omniscient observer (see the sidebar), rather than strictly from the point of view of the participants. How does the receiver know the intent of the sender? Perhaps, the trigger is late because recent events at the sender caused the sender to change state from "stop sending" to "more to send." Essentially the sender had intended to let the state expire and then

did otherwise. (Anthropomorphizing state machines is dangerous.) How does the sender even know that the state is being discarded? Does it care? Whether state is kept or discarded by the receiver has no effect on the sender. From the sender's or receiver's point of view, there is no such thing as "false removal." This is dangerous reasoning. The action of protocol machines cannot be based on what is surmised to have happened elsewhere, but only on the inputs it sees. No assumptions can or should be made about what generated those inputs.

2. The second case is soft state with explicit removal (ss+er), which is described as the first case with an explicit ER PDU sent by the receiver when its timer expires, and the state is deleted telling the sender that the state has been removed. This gives us a weak form of "at least once." If the ER is received, the sender will know that at least one of its trigger PDU was received, but it does not know which one, and there is no indication in the problem state that it matters to the sender. Note that the ER PDU is only useful from the point of view of the observer, not the sender. The sender need not take any action whatsoever. If the sender did not intend the state to be deleted, the next trigger timeout will rectify the situation with no explicit action by the sender in response to the ER. If it did intend the state to be deleted, and it was, there is nothing the sender needs to do.[14] This is a case of not recognizing a degenerate case and, worse, turning it into a special case. The semantics of the ER is as an "indeterminate ack" or that "at least one trigger PDU arrived." There is no indication in the problem specification that this affects the state of the sender. From Watson's point of view, the ER as explicit removal is redundant because a connection/flow is never terminated; the receiver just ceases maintaining cache space for it. Once again, we see distinctions that are in the eye of the observer, not the organism.

3. The third case is soft state with reliable trigger (and explicit removal) (ss+rt), which is defined by the receiver sending an ack to each trigger. The sender starts a retransmission timer when the trigger PDU is sent. (We assume this is the only reliability mechanism, that there are no lower layer functions providing reliability.) If it expires with no ack being received, it retransmits the same trigger PDU. State removal is explicit, as in the second case. This corresponds to either a strong form of the "at least once"

[14] It is easy to see that any timeout value for the ER is going to arrive at a time less than the normal update period. Because the ER is not an ack, it must be sent after more than one trigger is missed, and therefore we can conclude that missing more than one trigger is not critical to the system. If it is, the sender should have had a higher update rate to begin with.

case or a weak form of Spector's "only once," depending on whether one takes a glass half-full or half-empty view. If an ER PDU is received and there is an ack outstanding, the sender will not know whether its last trigger PDU was received. Ping et al. are unclear, but it does appear that if an ack is not received by the time the trigger timer expires, a new trigger PDU is sent regardless. (This follows from our assumption that replacement is the only operation.) This would imply that the old ack is no longer relevant and the sender is now waiting for the ack from the new trigger PDU. This could persist so that the sender never receives an ack or an ER. Similarly, if the ER is lost, the sender will not know that the state has been deleted, but then it doesn't affect the sender at all.

4. The fourth case is soft state with reliable trigger/removal (ss+rtr), which is defined as using "reliable messages to handle not only state setup/update but also state removal" (italics in original). Ignoring the fact that Ping et al. do not make clear how the explicit removal is made reliable without running afoul of the Byzantine generals problem, this is the same as the previous case—because, as we have seen, the only information the ER adds to the semantics is the knowledge that at least one trigger was received, and there is no indication that the sender needs this information.

5. The fifth case is hard state (hs), which is defined as using reliable PDUs to set up, update, and remove state, but it is not stated how they are made reliable. Ping et al. state that "neither refresh messages nor soft-state timeout removal mechanisms are employed," and go on to say that a significant problem with hard state is the removal of orphaned state and reliance on an external signal to remove it.

The authors do not make it clear, but generally hard state has been applied to protocols such as TCP or delta-t. It is unclear here whether we are considering this class of protocols as some impractical straw man with no timers. The problem of orphaned state arises directly from the requirement that both the sender and receiver know that they have terminated normally. It is the Byzantine generals problem that ensures that this cannot be the case. In the

Getting the Proper Perspective

We have made the point that it is always important to view things from the point of the view of the "organism," not the observer (a lesson I learned from Heinz von Forester, a pioneer in cybernetics). He often illustrated the point with a story to show the fallacy in B. F. Skinner's theory of conditioned response.

Heinz would begin his story with that twinkle in his eye in his Viennese accent: "We are going to teach an urn!" At this point, the student is thinking, an urn? A vase? What does he think he is doing? "We put an equal number of red and white balls in the urn," Heinz would continue. "We are going to teach the urn to only give us red balls. To teach the urn, we reach in and pull out a ball. If it is a white ball, we *punish* the urn and throw it away. If it is a red ball, we *reward* the urn and put it back. Eventually, the urn learns to only give us red balls!"

Clearly, the reward/punishment is in the eye of the observer, not the organism, the urn. These false distinctions turn up often. It is important to keep a clear perspective about to whom distinctions are important. This is often the source of significant simplifications.

> case constructed by Ping et al., that both sender and receiver know that a common point in the conversation was reached was not a requirement.

So to some extent, we are comparing apples and oranges. This class of hard state protocols does (where it matters) use timers and uses them to eliminate orphaned state. In the case of delta-t, however, orphaned state cannot occur by definition. The primary characteristic of hard state protocols is that the operations of some mechanisms are such that if the state of the sender and receiver are inconsistent, the protocol will deadlock. Hard state protocols have the mechanisms to avoid these conditions, increasing their complexity. It is the degree of consistency and amount of shared state necessary to avoid these conditions that makes them hard state protocols.

As we can see over the years, the emphasis of this issue has shifted from the shared state of data transfer protocols[15] to the shared state for client/server applications to signaling protocols and more generally to what are essentially distributed databases, largely following the popular topics of the time. To some extent, this also parallels our increasing understanding of where the crux of the problem lies. As we have seen, the results from Ping et al. reduce to variations on Spector's cases as seen through Watson. These results are consistent with our findings here. For a data transfer protocol, there are basically three cases: a pure connectionless protocol (maybe), a protocol with only tightly coupled mechanisms providing feed forward requires only a two-way handshake for synchronization, and a protocol with loosely coupled mechanisms providing feedback (at least once) and requiring a three-way handshake for synchronization (only once). Three points don't make much of a continuum.

In addition to the lack of points on a continuum, the shared state approach also doesn't address the hop-by-hop versus end-to-end aspect of the issue. In the traditional connection solutions, resource management is done on a hop-by-hop basis. Connectionless has always been associated with a "best-effort" service with a greater degree of "quality" provided end to end by a higher-layer protocol. However, as connectionless networks have grown and their use broadened, this has presented problems. As the range of applications has grown (mainly over past decade or so), the perceived need for levels of services other than "best effort" have also grown. While overprovisioning has generally been seen as a solution for this collection of issues, it is realized that this is not a long-term solution, nor is it always available. The conundrum for those of us who prefer the flexibility and resiliency of the connectionless model has been that any

---

15 I say data transfer protocols here because with X.25 we are comparing error control at the data link layer and resource allocation and flow control at X.25 Layer 3, with resource allocation of IP at Layer 3, and TCP error and flow control at Layer 4.

attempt to address these issues has led directly to a connection-like solution. One or the other, no synthesis. However, this has not led to the conclusion that the connection model was right after all. The connection-oriented solutions have suffered from the classic problems of static resource allocation, tended to not perform much better if not worse than the connectionless ones, and have encountered complexity and scaling problems. And yes, in all honesty, we must admit there has been a bit of ego involved in not wanting to admit connectionless might not be the answer.

But even among the calmer minds, there has been a sense that there is something we have missed. The intensity of the networking Thirty Years War has tended to push us to one extreme or the other. If one were in the debates at any time over the past 30 years, one could see that the fear of not knowing where a compromise would lead kept compromises from being explored and reinforced the combatants into their trenches. From looking at the proposals, that certainly seems to be what has transpired. All of this and only the three points on our continuum is perhaps an indication that the "degree of shared state" is a red herring, or at least not an approach that will yield the synthesis we were looking for. But perhaps the preceding observation will give us an opening.

What is most interesting is to return to Clark's characterization that the problem with hard state is that "because of the distributed nature of the replication, algorithms to ensure robust replication are themselves difficult to build, and few networks with distributed state information provide any sort of protection against failure." X.25 provides error control on links with a variation of HDLC, LAPB. HDLC, as we saw, does "end-to-end" error control; the ends are just closer together. The replication is primarily associated with resource allocation and flow control. Given what we have just seen with Ping et al. analysis of signaling protocols, one begins to realize that it is not possible to compare just the data transfer protocols. The different approaches to data transfer protocols tend to include functions that the other assumes are elsewhere or vice versa. This is an apples and oranges comparison.[16] One must include the signaling or "control plane" aspects in the analysis, too. We need to look at the whole problem.

When we do this, something quite remarkable begins to emerge: The connection-oriented approach is actually trying to minimize the amount of shared state. Its fragility derives (as Clark alludes to) from its inability to respond to failure. If a link or node along a virtual circuit breaks, the nodes participating in the virtual circuit don't have sufficient information to take action. They don't have *enough* state information. *It is brittle because when it breaks, no one knows what to do*. The connectionless approach avoids this in a somewhat

[16] I have to admit I dislike this terminology because it is so heavily associated with the beads-on-a-string model.

counterintuitive manner: It distributes everything to everyone. Everyone knows everything about the routing of any PDU. And each PDU contains all the information necessary for any system to handle it.

In a very real sense, connectionless is *maximal* shared state, not minimal. (Hey, I was as surprised as you are!) The interesting difference is that by widely disseminating the information, the complexity (hardness) of the protocols required is reduced. The connection-oriented approach is forced to hard state protocols because it tries to minimize the information in each PDU and in each node in the virtual circuit, centralizing routing and resource-allocation functions elsewhere as much as possible. This forces the use of much more complex mechanisms to maintain consistency. Whereas connectionless puts as much information in the routers and the PDUs as possible, thereby being less affected by failures. From a historical perspective, this is also consistent. The connection-oriented PTT advocates were from an older, more Malthusian tradition that believed that everything must be used sparingly and tried to minimize resources in favor of increased (deterministic) complexity, whereas the connectionless advocates derived from a younger tradition of plenty where memory was cheap and getting cheaper, and they were more comfortable with stochastic processes that might consume more resources but were simpler and had better overall characteristics.

The concept of shared state has always included both the amount of state and the degree of the coupling. An idea of how inconsistent the shared state could be for some period of time, always hard to quantify. Initially, our focus had been on just the data transfer protocols, but the properties we saw there were more affected by the routing and resource-allocation functions of the layer than we cared to admit. We weren't looking at the whole problem. Once we do, it is apparent that connection-oriented protocols are trading off complexity to minimize resources (state) but making the system more brittle in the process, while connectionless protocols disseminate information (state) widely in favor of simpler more resilient characteristics.[17] Because everyone has been given sufficient information for routing to know what to do with any PDU, and because every PDU contains the necessary information for anyone who sees it to act on it, the loss of state by any one system has minimal impact. Consistency is still a concern. But we extend considerable effort on routing protocols to ensure that Routing Information Bases become consistent quickly. (Perhaps the dumb network "ain't so dumb" after all!) In a sense, we have orthogonal axis; connec-

[17] If there were a means to quantify "degree of coupling/complexity" with amount of shared state, I suspect we would find that, looked at from this perspective, there is not much difference in the total "shared state," although there is a difference in the characteristics of the result.

tionless represents maximal state in terms of space, while connections are maximal state in time.

The connectionless approach focuses entirely on routing and ignores the resource-allocation problem, whereas the connection-oriented approach centralizes routing (that is, forces it to the edge) and concentrates on resource allocation along the path.

So, how does this insight contribute to our desire to find a unifying theory that integrates connection and connectionless? We would like to be able to support flows with different characteristics (QoS) while retaining the flexibility and resiliency of connectionless. We have also noted that the proper place for connections is when traffic density is higher (more deterministic) and for connectionless when it is lower (more stochastic). We distribute connectivity information to everyone, but we have always insisted on only distributing resource-allocation information along the path. No wonder our attempts at providing QoS always turn out looking like connections; we are doing what they do! We were able to break with the past and not think that routing had to be associated with a path, but we didn't do it when it came to allocating resources! We need to treat resource allocation just as we treat routing: as distributed.[18]

Going back to the "other" characterization of the difference between co/cl: In connectionless, the processing of each PDU is independent of the processing of other PDUs in the same flow. With connections, the processing of each PDU in a flow is determined by the processing of the previous PDUs. Or to turn it inside out, with connection-oriented resource allocation, information (including routing) is only stored with nodes on the path of the connection; whereas with connectionless resource allocation, information is stored with every node. Let's consider *this* to be our continuum? The probability that this PDU is processed precisely like the last PDU in this flow varies between 0 and 1.

For routing, each router computes where it should forward a PDU based on information from other routers it receives. Let's take a similar approach with flows. Based on information from other routers, each router computes the probability it will be on the path for this flow and if so what resources should it "allocate." An aspect of the exchange might include an indication of the sender's intent to spread the resources over several paths.

This leads to a model where flows are spread across different paths and resources are allocated on the basis of the probability of their use. If we assume that there are m paths between A and B, a connection is represented by one path having the probability 1 of being used. For example, there might be 5 paths between two points with probabilities: 0, .2, .6, .2, and 0. Pure connectionless is

[18] Our attempts at QoS reflect this dichotomy. We have tried the connection approach ATM, MPLS, InfServ (RFC 1633, 1994), and so on, and the pure connectionless approach, DiffServ (RFC 2430,1998). RSVP (RFC 2205,1997) is essentially connection establishment for IP.

represented by each path being equally likely or 1/n. Distributions may be anything in between. The question is what resources should be allocated for these probabilities given the desired QoS. This reorients the problem to one of distributed resource allocation, not flow reservation. We have considered the problem as one of allocating flows, not allocating distributed resources.

Each node has resources to allocate. Each initiator of a flow requests distributed resources based on the QoS parameters of the request. Each member of the layer can then compute the probability that it is on the path and allocate resources accordingly. This gives us a model for doing quite a bit more than just integrating connection and connectionless under a single model. In a sense, this model extends Watson's "particle physics" model of connections. That is, all connections always exist; we just don't cache state for them, unless resources are allocated for them. Here we view flows as a probability distribution, or we might characterize a flow as smeared across several paths. Routers allocate resources based on the probability that they will receive PDUs for a flow. This allows routers to reserve resources to better respond to failures during the transients while resource-allocation information is updated in response to a failure.

This is not a "major breakthrough," nor is it intended to be. We were looking for a unifying model of connection and connectionless. This one seems to hold promise, and it is the only one we have found that does. This will give us a tool to aid our thinking. How does thinking in terms of this model change how we look at the problem? However, as interesting as it might be, this is not the place to pursue it. We need to continue our journey before we have all the machinery necessary to apply this approach to solving this problem. Although we can't say as yet whether this approach is tractable, we can certainly note that it does integrate connectionless and connection. In fact, one of the appealing aspects of this model is that it seemingly makes the distinction between connections and connectionless meaningless (although far from proof, this is definitely one of the trademarks of a "right" answer), similar to when a solution turns out to solve problems other than the one it was intended for.

Conclusions

In this chapter, we have found that separating mechanism and policy uncovered some interesting patterns in the structure of protocols and revealed that the plethora of data transfer protocols we have today actually have more similarity than differences. The data transfer protocols seem to divide into two kinds: relaying and multiplexing protocols and error- and flow-control protocols. We

have found that the range of error- and flow-control protocols, from the weak UDP-like to the strong TCP- or delta-t-like protocols, can be simply accommodated by a single structure. We have also looked at the Thirty Years War over connections versus connectionless. We have been able to characterize the proper role of each and have made some surprising insights into the difference between connectionless and connections and made great strides in finding a synthesis that avoids the damaging schism they have caused in the past. But, we have gone about as far as we can looking at individual protocols. We need to consider the structures that result from assembling protocols with different scope. But before we can do that, we need to collect some more data to have a good understanding of the landscape. The next two chapters review the state of our knowledge of the "upper layers" and of addressing.