An Argument for Soft Layering of Protocols

Geoffrey Howard Cooper

May 1983

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

An Argument for Soft Layering of Protocols

by

Geoffrey Howard Cooper
B.S., Massachusetts Institute of Technology (1980)

Submitted to the
Department of Electrical Engineering and Computer Science
on April 8, 1983 in partial fulfillment of the requirements
for the Degree of Master of Science

Abstract

This thesis is about the efficiency of protocol layering. It examines the technique of protocol layering in an abstract way and finds two major sources of inefficiency in protocol implementations which are caused by the imposition on them of a layered structure. The conventional approach to making layered protocol implementations run efficiently — for avoiding the sources of inefficiency discussed herein — are all independent of the protocol specification, and thus all decrease the value of the protocol specification as a guide for implementing protocols.

In this thesis, we introduce a new means of avoiding the problems of layered protocol implementations which operates within the domain of the protocol specification. We allow an increase in the flow of state information between the layers of a layered protocol implementation in a very controlled manner, so as to decrease the modularity of the protocol architecture as little as possible. The increased flow of information is specified in the protocol specification as a model of all the layered protocols that use the protocol being specified, called the "usage model." Since our approach decreases the rigidity of the layered structure without entirely eliminating it, we coin the term 'Soft Layering' for the approach.

Key Words: computer networks, layered protocols, soft layering

Acknowledgments

A thesis is the result of individual efforts, which are supported by a group effort. I would like to acknowledge the help I received from all the members of the Computer Systems and Communications, and Computer Systems Structures groups at the MIT Laboratory for Computer Science during the development of this thesis. Special thanks are due to Dave Clark, for supervising the thesis, and to Larry Allen and Lixia Zhang for reading it in draft form. Thanks are also due to my office-mates, for "keeping the hordes away" while I was trying to finish up the thesis.

Finally, thanks are due to Marianne, for standing behind me all the way through, and for pushing me to finish.

Table of Contents

Table of Figures

Chapter One

Introduction

In the last few years there has been a great increase in interest in packet-switched networks. Of particular interest is the introduction of low-cost local-area networks, such as Ethernet [8] and ring nets [3], which can be expected to bring about a dramatic increase in the number of networks in operation in the next few years.

Just what is a network? At one level, a network is a switching and communications mechanism which enables a number of machines to transfer bits to one another. Usually, it is not possible for all the machines to transfer bits to all the other machines without restriction (in particular, it is rarely possible for all to send bits at the same time), but most networks endeavor to allow communications to take place on as unrestricted a basis as can be allowed by the scarce resources available to it, namely: communications lines, buffering, node processing time, etc.. In effect, the network is viewed as an abstraction of a vast number of wires.

At a higher level, we note that it is insufficient for the network merely to provide standards which couple the hardware of different computer systems one to the other. Two computer systems which are able to transfer bits back and forth between themselves may still be unable to communicate, because they have no way of understanding the bits that are transferred. The network must also support the establishment of conventions which allow computers to correctly interpret the bits that are transferred over it. These conventions are formalized in communications *protocols*.

A protocol is the means by which two cooperating processes[1] communicate. It consists of a set of agreements governing the form of the data that is sent over the communications channel, and the conventions for what data is to be sent under what circumstances. Put simply, a protocol specifies *what* may be sent *when*. The reader is referred to Pouzin and Zimmermann [25] for a good introduction to the world of protocols.

To generalize the notion of a protocol, it is common to replace the word "process" with the

[1]We use the term 'process' to mean a program in execution.

word "*entity*." An entity is anything (inside or outside of the computer) which is capable of participating in communications — anything which may use a protocol. In a computer, an entity is thought of as the body of code which is responsible for generating and handling the messages of a particular protocol.[2]There are typically many different entities in a computer system at any one time, and the entities may be divided up in a manner completely orthogonal to the process boundaries of the application.[3] The network interface hardware must be shared among these entities; we refer to this sharing as *multiplexing*.

From yet a higher level perspective, a network is viewed as a means of providing entities on machines connected to it with a means to access a larger number of *services* than they would otherwise be able to use. For example, a process running on a computer system receives 'file storage' services from the local file system. When connected to a network such as the Arpanet [20], it might also be able to make use of the file storage services of other computer systems.

The concept of a network as a vehicle for accessing services has been finding increasing use. Most notable are the attempts that have been made to produce standardized collections of communications protocols which can be implemented on many machines. For a process to access the services provided by another, both must implement a compatible set of protocols. While it is sufficient to choose any protocol to allow two cooperating processes to communicate (in particular, two processes may agree to communicate using some specialized 'custom' protocol), agreement on standardized protocols is necessary if a single process wishes to connect to many others.

Specific examples of standardized collections of protocols include the Arpanet protocols [20], the newer U.S. Department of Defense Internet protocols [23], IBM's SNA protocols [7], and the Xerox *NS* protocols [19]. A collection of protocol designs is sometimes referred to as a *family* of protocols, or a *protocol architecture*. We shall prefer the latter term here.

[2]In this thesis we will use the term 'message' to refer to communications between entities at the same level in a protocol hierarchy. The term 'packet' will be reserved to represent the combination of protocol messages which is actually presented to the hardware interface.

[3]It is important for the reader to understand that there is a distinction between network entities and the asynchronous processes of a computer system. There need be no one-to-one correspondence between entities and processes; one process can contain many network entities, and a network entity can be implemented using several processes. The latter situation is less common; the 'Telnet' remote-login protocol [22] provides a good example. The keyboard-to-remote-host and network-to-terminal functions of telnet are easily separated. In computer systems such as Unix [26] where any I/O input operation can cause the requesting process to block indefinitely, the telnet protocol is commonly implemented using two processes, one to connect the keyboard to the network and the other to connect the network to the printer or display.

1.1 Modularity in Protocol Architectures

To produce a complex collection of protocol designs, and to implement these designs correctly and in a reasonable period of time, it is desirable to find common services needed to support different applications, so that it is possible to share both the designs for these services and the code that implements them. To this end, protocol architectures must enforce modularity.

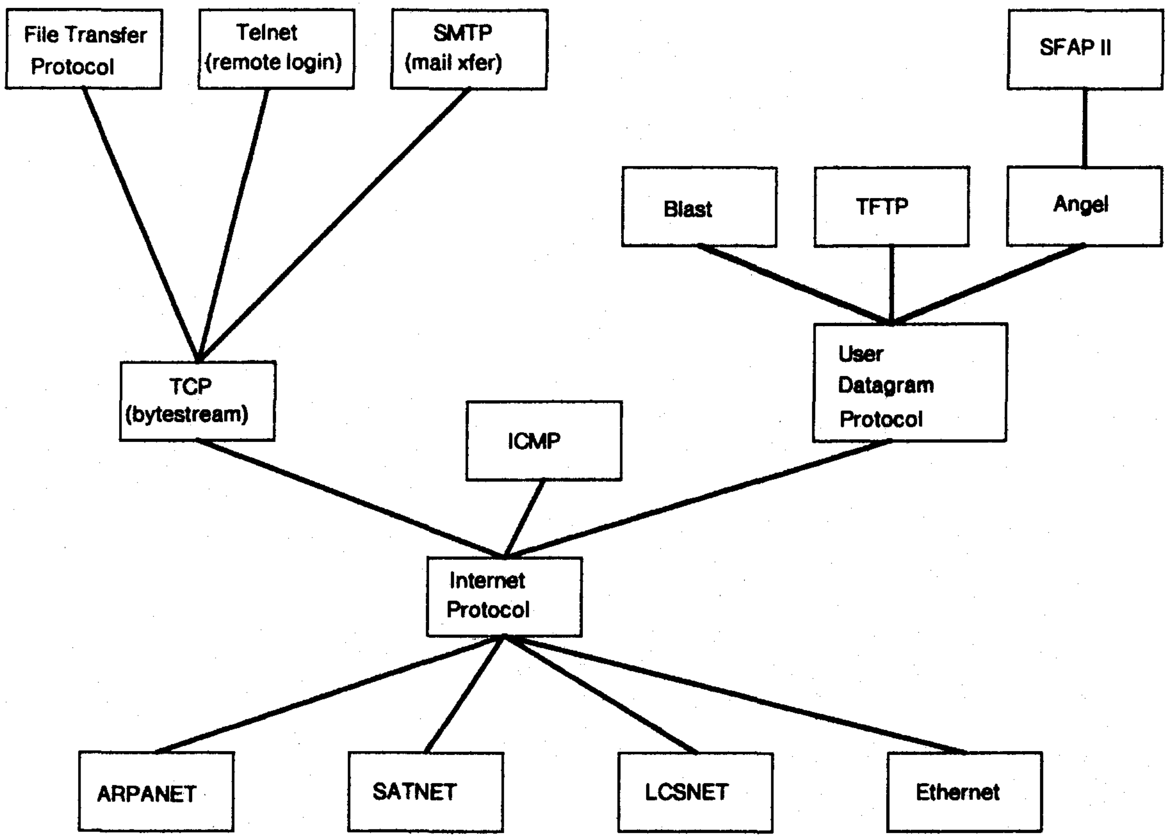


Figure 1-1:Part of the Internet Protocol Architecture

Protocol architectures typically contain many different protocol designs, to accommodate the many different services that are to be accessed by applications which make use of the architecture's implementation. The implementation of even a single protocol is characteristically a difficult task, because protocols usually require the use of the more complex facilities which are provided by operating systems, such as lower-level I/O mechanisms and asynchronous scheduling; when an entire protocol architecture is implemented as a single, unstructured unit, the implementation task becomes formidable. The protocol architecture should thus suggest a modularization that may be used in the implementation process.

The particular kind of modularity that has come to be used in most protocol architectures is the technique of protocol *layering*, whereby modularity is achieved by imposing a lattice structure on the different protocols in the architecture: each protocol must only interface with those protocols immediately 'above' and 'below' it in the lattice. As an example of this sort of design, part of the design ordering for the Internet protocol architecture [23] is contained in figure 1-1.

Layering of software, in the general case, is the natural result of the two goals that must be met by any operating system's interface to a hardware device:

- *Hardware Multiplexing*: the need for many different applications (and different instances of the same application) to gain access to the hardware without interfering with each other.

- *Software Modularity*: the desire to allow software to share the code that implements commonly used operations on the hardware.

Consider a disk drive connected to a computer system. One approach to dealing with the drive would be to enclose all applications that use it in a single subsystem, internal to which are implemented all the conventions for dealing with the disk and the data that is stored on it. In the situation where the number of different applications using the disk is limited (for example, in a database system), there are advantages to this approach.

In the more common case of a "general purpose" computer system — one designed to serve as a general tool with a limited preconception of the applications that will run on it — the number of different applications which run on the system is usually large. The requirements of such a computer system are always in flux, so new applications are frequently created, and old ones frequently changed. In this situation the problems associated with creating and maintaining a single subsystem which embodies all the applications which access the system's disk hardware make this approach to dealing with the hardware prohibitive.

General purpose computer systems traditionally use a different approach to dealing with disk hardware. The operating system does not try to understand all the bits that are stored on the disk. Instead, the disk is partitioned into those parts that are meaningful to the operating system — the disk sector and track addresses, the list of free blocks, and a few disk blocks which contain information internal to the file system — and those parts that contain data that is meaningful only to the applications that use the disk. To allow the applications to use the disk without interfering with each other, the operating system provides operations that store data to the disk and retrieve data from it, and mediates these actions so as to control the degree to which applications can sense each other's existence.

Once it has become necessary for the operating system to mediate all access to the disk, it becomes reasonable to solve the other problem of interfacing to a disk drive — that of centralizing the implementation of the most commonly used disk actions — by implementing these actions inside of the operating system. Thus, operating systems provide operations on files and directories instead of disk blocks and free lists. Since the need for hardware multiplexing and software modularity of the disk software go hand-in-hand on most computer systems, the concept of establishing layers of software around the disk hardware follows naturally — it allows the system to provide both functions in a single mechanism.

A similar argument applies to the network hardware of a computer system. In a common situation, the functions of controlling the network interface hardware and allowing it to be shared by different network entities are embodied in a 'hardware driver' for the network interface. A portion of each packet is reserved for the driver, to enable it to multiplex incoming packets correctly.

Once the needs of the system have caused the hardware driver to exist, it is possible to perform the needed function of extending the services provided by the hardware within the driver. For example, it is possible for the driver to decrease the probability that a packet is corrupted in transmission by including a checksum of the packet in its reserved portion, and by agreeing with other hardware drivers that any packet that arrives with an invalid checksum is to be retransmitted until it is received correctly. Similarly, a more complex driver might provide sequenced, unduplicated, reliable delivery of packets, in order to modularize the function of correcting the most common failures of packet-switched communication networks for all the clients of the network hardware at once. Another advantage of the existence of a level of mediation on all network actions is that it abstracts the details of the driver from the rest of the network software. If the driver ever needs to be changed in a way which does not affect its interface with the other protocols in the system (the "higher-level" protocols), the code to be changed is completely contained within the driver.

The above examples illustrate how the requirements which must be met by most computer systems which provide an interface to shared hardware lead them naturally into a layered structure. In the network example, we have the beginnings of a layered protocol architecture; we might begin to speak of a *driver-to-driver protocol*, which we have inserted *under* all the other protocols in the system, and which is independent of them.

There is a danger in layering. Some protocols might not need the added features which are provided in the layers that lie beneath them. Since the lower-level protocol 'hides' the network interface, they will receive these features anyway. At best, this will slow down these protocols. At worst, it will prevent them from functioning at all.[4]

Luckily, hardware multiplexing comes to the rescue. Since the hardware driver is already capable of hiding the existence of two different entities from each other, it is possible for it to define a second driver-to-driver protocol for these protocols, which does not implement the higher level functions.

We may extend this layering technique further by dividing the higher-level protocols themselves into different protocol layers. Where we wish to share the code for a protocol function among several entities we define a layer that performs this function, and reserve a portion of each packet that is transmitted for a *protocol header* that is reserved for the use of the new layer. At the same time, we introduce enough multiplexing into the next lower layer so that every protocol does not have to make use of the functions that are provided by any particular layer below it. When every protocol in a protocol architecture is layered, we arrive at the *layered protocol architecture* with its lattice structure.

For a protocol architecture to be useful, it must be possible to produce, at reasonable cost, an implementation of it which does not unduly burden the resources of the computer system on which it must be run; i.e., the implementation must be efficient. A criticism of layered protocol architectures is that they are inefficient when implemented in the manner which is suggested by their layered design, and hard to implement in a manner which is efficient [18].

In this thesis we will see that a layered protocol design, when carried into the implementation, does indeed hamper the production of an efficient protocol implementation. We contend that this is so because some of the problems that protocols must solve do not readily modularize into layers, and existing layered architectures ignore this fact. Since the advantages of a layered design are many [6], we do not advocate the abandonment of layering in protocol

[4]For example, a protocol for performing a data-collection function from remote monitoring stations might operate by having each monitoring station transmit periodic updates to its state [2]. The central characteristic of many such schemes is that newly received data from each station obsoletes all its previous messages. In this case, a driver protocol which simulates a reliable FIFO data stream on a packet switched network might prevent such a scheme from working correctly, because a FIFO stream presumes that older information is more important. If an entity making use of a FIFO stream based data-collection protocol ever gets behind in its processing of the remote monitoring station's data, the FIFO nature of the underlying communications requires that the entity catch up with the monitoring station before it can receive the latest data sent by the station. If the entity is normally very close to being saturated by the normal amount of data that it receives from the monitoring station, it may never catch up. This problem has also been noted in the implementation of protocols for the real-time transfer of digitized speech [5].

design. Rather, we shall present an extension to conventional protocol layering, called *soft layering*, which provides a mechanism for different modular structures to coexist in a layered implementation.

1.2 The Correctness and Efficiency of Software

The previous section has made implicit use of two issues which are integrated deeply into the process of producing working computer software: *correctness* and *efficiency*. Correctness is the degree to which a program meets its specifications. Efficiency is the degree to which a correct program avoids overburdening scarce resources while it is running. This thesis is greatly concerned with the interaction between the correctness and efficiency of network software.

The issues of correctness and efficiency are usually applied quite differently. A correctness criterion is applied uniformly over all implementations, while an efficiency criterion is applied differently for different 'correct' implementations. For example, if two compilers for a (properly specified) programming language are *correct*, a program which compiles using one will compile using the other, and the resulting object-code modules will do the same thing. However, even if we assume that both compiler implementations perform 'efficiently,' it is likely that the amount of CPU time taken by each compiler to compile the same program will be different, perhaps radically so. To see how this can be, consider the possibility that one compiler was designed to run on a machine with a small address space, and the other on a machine with no such limitation. In the case of the first compiler, the primary concern is whether or not the program makes effective use of the memory available to it; in the case of the second compiler, the amount of CPU time consumed per compilation would be more important. We are comfortable using different criteria to judge the efficiency of different implementations.

A non-uniform view of program efficiency can fail when different implementations of the same specification must interact with one another. In the compiler example, above, the two compilers interact in a manner which stresses only their relative correctness. It is possible to imagine a scenario in which the two compilers interact in a manner which stresses their relative efficiency.

Imagine that the two compilers are written in the same computer language, and that both use identical intermediate representations of the source code. It is decided to turn the two compilers into a 'distributed compilation service,' where parts of the service reside on different machines, connected via a communications network. When a client of the service submits a program consisting of a number of modules for compilation, the service divides the tasks of parsing and optimizing of the modules between the two compilers, each of which runs on its native machine (to cope with object code incompatibilities, code generation is always done on the local machine). In order for the compilers to be able to resolve and type-check references to global variables, a mechanism is provided for each compiler to access the other's symbol table.

Both compilers implement symbol table abstractions that are equivalent (i.e., the operations on the symbol table are effectively the same), but were designed with different concerns for efficiency. In the first compiler, which runs on a machine of small address space, symbol lookup is very slow, but the memory occupied by the symbol table is small. In the other compiler, symbol lookup is fast, but the table is allowed to become very large. It is likely that if we were to meter the amount of time spent by each compiler on local symbol table operations, we would find that for a typical compilation there was no real difference: where symbol table operations are slow, the rest of the code is designed to carefully avoid unnecessary symbol lookups.

In the distributed compilation service, the two compiler implementations interact in a way which is affected by the relative efficiency of their symbol table abstractions. We expect that, ignoring network delays, the compiler which normally uses the 'fast' symbol table implementation will run slower than either of the original two, because the code in it that uses the symbol table abstraction *assumes* that symbol lookups are efficient, and does not try to avoid them — even though this assumption is no longer valid for external symbol lookups.

The example illustrates what is common knowledge to those who build complex software, that for software to interact efficiently with external modules, it is necessary to augment the specifications of the external modules with a 'model' of the efficiency considerations that mattered most when the modules were implemented. When software modules which assume different models of each other's concerns for efficiency are forced to interact, they will tend to do so in an inefficient manner.

In this thesis, we deal with the design and implementation of communications protocols. Software that implements communications protocols is characterized by a constant need for different correct implementations to interact efficiently with one another. It is our contention that in the network environment, it is essential to impose a single 'model of efficiency' which is used by all implementations to ensure that they will interact efficiently.

1.3 Scope of the Thesis

There are many different kinds of protocols, and many different kinds of networks. While we believe that the issues of this thesis apply over a broad selection of these, we have found it necessary to restrict the range of hardware and protocols with which this thesis is concerned to simplify the discussion, and to make it easier to see the issues at hand.

This thesis concerns itself only with reliable communications between a pair of cooperating entities. By reliable communications, we mean that data is transferred from one entity to the other and acknowledged, so that the first entity can be certain that the data was both received *and* acted upon. The reader should be aware of the fact that it is not sufficient for the sending entity to trust underlying mechanisms to deliver the data to its partner, regardless of their reliability: there must still be an *end-to-end* acknowledgment between the two cooperating highest-level entities to ensure that the receiving entity has not only received the data, but also handled it correctly.[5] The ISO Reference Model for Open Systems Interconnection [11] provides a particularly clear terminology for dealing with the distinction between reliable communications (i.e., with end-to-end acknowledgement) and reliable delivery (i.e., without end-to-end acknowledgement), and we will adopt this terminology for the rest of the thesis. A data *transfer* from an entity A to an entity B means that A has sent the data to B *and* received an acknowledgement for it. On the other hand, a *transmission* of data from A to B means that no acknowledgment was sent: A is willing to rely on the underlying mechanisms to deliver the data to B, and does not care to know for certain if B receives the data or not.[6]

Reliable communications, as presented here, also implies the existence of a *connection.* A connection is a caching of information which is used to make repeated transmissions more efficient. We are not concerned herein with the class of network protocols that do not make use of a connection. These include the Internet name-server protocol [23] and some data-acquisition protocols [2].

[5]This principle, which has been called the *end-to-end argument* will recur frequently in this thesis. The end-to-end argument states (in part) that, in a layered software system, if an end (highest-level) entity wishes to ensure that a requested action at its level takes place, it is not sufficient for it to rely on the correct performance of lower-level entities; it must perform its own confirmation of the action *in addition to* any that are performed at a lower layer.

If the reader is not convinced of the end-to-end argument, he is referred to [28] for a more complete description of it and its implications for network software.

[6]It is important to distinguish between A *not caring* whether B receives the data or not, and A *not caring to know for certain* whether this is so. If A does not care whether B receives the data that it sends, there is no need for A to send it. The situation we have described is one in which A is satisfied with the knowledge that B will receive and act on the data with some particular probability, which is a characteristic of both B and the underlying delivery mechanisms.

There are many concerns in computer communications other than those of reliability. We avoid discussion of flow control, congestion control, routing, and error recovery in this thesis.

We also restrict ourselves to consideration of two-way communications. Issues such as reliable broadcast or multi-cast, or a more random connectivity of more than two distributed processes are avoided here. Partly, we avoid N-way communications as a means of limiting the size of the thesis. It is also the case the we do not feel that the discussions presented in this document would benefit from the added complexity of more than two processes. We believe that the simpler example makes the point better.

It is implicit in all that follows that the underlying hardware is that of a packet-switched network. This includes the range of such networks from long-haul networks such as the Arpanet [20] to local-area networks such as the Ethernet [8] or ring nets [3]. It is the author's feeling that the arguments of this thesis could be described in terms of either packet-switched or circuit-switched networks. In the interests of brevity, only one of the two will be used here. The choice of packet-switched networks stems from their increasing popularity in data communications.

1.4 Background and Related Work

The idea of packet switching dates back to the mid-1960's. The first large-scale packet-switched network was the Arpanet [9]. It was in the context of the Arpanet that the first 'family' of layered protocols (what is herein called a 'protocol architecture') was introduced. This collection of protocols has been in service, with periodic enhancements, from the late 1960's to the present. Its principal application protocols provide remote login, file transfer, and electronic mail services.

Another network, Tymnet, was developed in the early 1970's [34]. While both Tymnet and Arpanet provided remote login as their primary service, Tymnet's protocols differed from those of the Arpanet in the amount of information that the lower layers were able to use concerning the nature of the higher layers of protocol. The use of higher level information in the lower layers has enabled Tymnet to control network congestion in a more straightforward way.

One criticism of the design of Tymnet is that it is not modular, since information is allowed to flow across layer boundaries in Tymnet which would be required to reside within a single layer in most existing protocol architectures. This argument is not justified, since Tymnet does exhibit modularity, albeit in a manner somewhat different from most other networks. In many ways, the added flow of state information between the layers of Tymnet is akin to the technique of 'soft layering' which is presented in chapter six. However, it is our feeling that soft layering allows a far more controlled flow of higher-level information downwards than is the case in Tymnet, and so is better at preserving the benefits of protocol layering.

The two protocol architectures developed by the Xerox corporation, named Pup [1] and *NS* [19] also contain elements which are reminiscent of soft layering. Specifically, in the protocols of the Pup and NS architectures, it is common for the lower level protocols to have fields in their protocol headers which are maintained solely for the use of higher-level protocols. This is also characteristic of lower level protocols in a soft-layered architecture; however, in a soft-layered protocol, both the lower *and* higher level protocols assign meaning to the fields, using the protocol specification of each as a means of coordinating the way in which each field is used. Thus, in some sense, soft layering takes the 'field sharing' approach of NS and Pup to its logical conclusion.

One of the most notable events of the last few years in protocol architectures has been the advent of the ISO Reference Model for Open Systems Interconnection [11]. The ISO model is not a protocol architecture in the sense of the others discussed above, but is rather an abstract 'schema' which describes the structure of protocol architectures. The ISO model attempts to define exactly seven layers for protocol architectures to use, along with the function and general characteristics of each.

More recently, the Arpanet has adopted a new set of protocols, entitled the Internet protocols [23]; the Internet protocols extend the familiar functions of the Arpanet's earlier protocol architecture to include networks connected to the Arpanet through gateways.

The development of the Internet protocol architecture is of particular pertinence to this thesis because it was during the 'rush' to understand and implement the new protocols of the architecture that the ideas of the thesis were developed. For example, the asynchrony problem, which is the subject of chapter three, was exhibited by a number of early TCP implementations. The protocols developed as part of this research were designed as part of the Internet architecture.

Strongly related to the topics of this thesis is the research into various new ways of implementing TCP that was carried out in the last two years or so by Dave Clark, Noel Chiappa, Larry Allen, Elizabeth Martin and Michael Greenwald. Most of the results of this research are contained in a collection of papers by Dave Clark which have been entitled the 'Internet Protocol Implementation Guide' [10]. This thesis should correctly be viewed as an extension of the ideas in this research to cover layered protocols in a more abstract sense.

1.5 Overview of Thesis

In the next chapter, we examine the technique of protocol layering in depth. The chapter also introduces some terminology that is used throughout the thesis.

In chapter three we examine a particular way in which layered protocol implementations fail to be efficient. The problem on which we concentrate, called the *asynchrony problem*, is caused by an inability of the lower layers in the protocol hierarchy to understand what the higher layers are doing. An additional problem, called the *timer problem* is also discussed.

In chapter four, we examine two different protocols, each of which avoids the asynchrony problem in a different way. This discussion introduces the notion of avoiding the asynchrony problem by increasing the flow of information between the layers of a protocol architecture.

Chapter five returns to the subject of chapter three and introduces a new terminology which makes it easier to understand the way in which the asynchrony problem occurs. In chapter six, this terminology is used to develop the mechanism of *soft layering*, which formalizes the flow of state information in the protocol specification. Finally, in chapter seven, an example of soft-layering is presented. Chapter eight summarizes the thesis.

Chapter Two

The Nature of Layered Protocols

The last chapter introduced the technique of protocol layering. In this chapter, we examine layered protocols in greater depth, to lay down the framework upon which the rest of the thesis will be built. To this end, we develop a definition of a protocol layer for future use, and examine the dependency relation between the layers in a layered protocol architecture. Finally, we present some terminology for distinguishing between the different layers in a layered architecture.

2.1 The Nature of the Layers

In the last chapter, we built up the technique of protocol layering. Now we wish to examine the question of what the technique is, from a more abstract point of view. A useful manner in which to proceed is to examine some different definitions of layering, so as to try and gain insight into the technique. Thus we seek a definition of what a layer is. There are three possibilities that we wish to consider:

1. A Layer is a protocol header, which envelops higher level protocols, and is enveloped by lower level protocols.

2. A Layer is an abstraction of a data communications channel, which uses a lower level abstraction in its specification, and is used by higher level abstractions.

3. A Layer is a software module which transforms one level of network service into a more complex level of service. The starting level of service is received from lower level protocols, and the final, more complex service, is offered to higher level protocols.

The first definition is equivalent to describing the progress of a packet which enters a computer system. Each layer acts as a wrapper for all the layers above it, so the packet must, in general, be parsed wrapper by wrapper. This approach has the advantage that it assigns an ordering to the layers which is both unambiguous and easy to determine by looking at a packet as it is transmitted over the network. The disadvantage of this definition is that it says nothing about the nature of layering: why the layers are there. It can be used as an analytical tool to understand particular implementations of network software, but does not help the designer of new protocols.

The second definition is our characterization of the strategy which is embodied in the ISO reference model (although the term "abstraction" does not appear therein). The definition is a good one, in that it deals directly with the nature of the layers, and makes clear the hierarchical structure of the design.

Our problem with this definition is that it is geared to a hierarchy in which all the parts rise uniformly. A protocol at layer N is restricted from making use of both a protocol at layer N-2, and one at layer N-1 at the same time. The reader might at first think that this is a necessary restriction if the N-1 layer is to be able to maintain its integrity; allowing a higher layer to poke down into the layers beneath layer N-1 would allow the higher layer to change the state of N-1's connection without its knowledge. We have already pointed out that multiplexing may be used at layer N-2 to prevent this from happening. Indeed, the Xerox NS protocols provide just such a facility [19].

An example of how this sort of behavior is useful is a file transfer protocol. Many such protocols (e.g., the Internet FTP protocol [20]) open a *control* connection, using a reliable stream protocol, in order to set up the environment for a second connection, on which the high speed transfer of the file takes place using a different protocol. On most operating systems, files are stored in blocks of data, and are accessed most quickly block by block (rather than as a byte stream, for example). It is simpler and more efficient for a file transfer protocol to use a (possibly unsequenced) datagram interface to the network. It would be reasonable for a file transfer protocol to use a datagram service for the high speed data connection. It might well be the case that the reliable stream protocol used for the control connection makes use of the same datagram protocol; this is of no consequence to the file transfer.

Since there is no need to avoid these and other[7] useful protocol combinations which are disallowed in the ISO model, we will not use the second definition here. To prevent confusion, we replace the notion of an abstraction of a communications channel, with that of a *network service*, and use the third definition, above.

[7]For example, protocols which use each other. The Internet protocol, which performs internetwork routing in the Internet protocol architecture, relies on the Internet Control Message Protocol (ICMP) to determine to which gateway or host to send every packet. In a gateway, ICMP uses the Internet protocol to communicate with other gateways, in order to coordinate routing information. Yet ICMP is not a part of the Internet protocol; it is possible for Internet to make use of other routing information, such as might be derived from newer or experimental routing protocols. The Internet protocol even protects itself from an ICMP error by eliminating packets which have been routed in a loop.

2.2 Dependencies Among the Layers

Any layered structure implies that there is some sort of dependency relation between the different layers. It is important for the subsequent development of our argument to examine the nature of the dependency relation that we assume for layered protocols.

It is certainly the case that for any module to provide its specified service, it must receive the correct services that it expects from all the modules beneath it. Note that this is not to say that it must receive error-free service from all the modules it uses. Rather it must correctly receive those services that it is not prepared or able to replicate on its own. For example, it is unlikely that a transport-level protocol would wish to cope with an error in routing at a lower level which causes its messages to arrive at machines which are randomly scattered throughout the network;[8] on the other hand, it is common for transport-level protocols to cope with lower-level errors such as lost packets.

One common sort of dependency that is exhibited by software is *failure dependency* — the higher level software is not willing to cope with a failure in lower level software.[9] In conventional, non-layered software systems, a software bug or a random hardware failure in any part of the system will usually cause the system as a whole to fail. In layered operating systems, such as Multics [27], crashes at some level can affect only higher levels.

Protocols are a special case, since there is no need for a failure in any layer to necessarily cause a failure in any other layer. For example, a failure in the terminal handling layer of a remote virtual terminal protocol need not affect the successful operation of either the underlying stream protocol or the higher level application software which is performing some computation. Of course, no further stream data can be transmitted to or from the application, so it will usually have to cease to function.[10] Note that this failure may come after a considerable delay (e.g., if the application didn't need any input or generate any terminal output for a long while), and might never come. Protocol implementations often allow a large degree of failure independence between at least some of the different layers.

[8]It is interesting to note that some mail-transfer protocols could conceivably cope with a version of this situation [23].

[9] We use the term 'failure' to mean the crash of a program, or the occurrence of an error which the program in question either could not or decided not to handle. This terminology is based on the termination model of program failures, as exemplified by the CLU language [16].

[10]There are common cases where this is not so; for example, it is common to deal with a failure of a hardware network interface by re-routing traffic through another connected interface. Another example is that an application layer protocol which attempts to transfer a file to a printer-spooler might cope with an inability to contact the spooler by simply choosing another printer/spooler combination.

2.3 More- and Less-Trusted Layers

Another way of describing the dependency relation between layers is to speak of layers placing *trust* in other layers. A layer is described as being more- or less-trusted depending on the number of layers that share and depend on it — that is, on the severity of a failure in the layer. Thus, the degree to which a layer is trusted is a measure of the amount of multiplexing that the layer is actually performing in a system (as opposed to the amount of multiplexing that a layer is capable of performing). Traditionally, in operating systems and in networks, the layers which are closer to the hardware (i.e., the lower layers) are the most multiplexed, and thus the most trusted.

It is bad software practice, in general, for a layer to place trust in (i.e., to rely on the correctness of) a layer which is less trusted than itself. Doing so would elevate the severity of a failure in the less-trusted software, which was likely constructed on the assumption that there was no danger in allowing a failure to occur in some circumstances; if the less-trusted software is relied upon by more-trusted software, the effect of what was intended to be a 'soft' failure on the part of the less-trusted software could propagate to the other less-trusted levels, by denying them the multiplexed service on which they were depending. In the most extreme case, the effect of a failure in a less-trusted software module could be to jeopardize the integrity of the entire system. For example, in the unlikely event that the security kernel of an operating system executes a user-supplied subroutine while still in the hardware-supervisory mode, a failure in the user's subroutine would cause a crash of the kernel, which would ultimately cause all processes on the system to fail.

It is unfortunately the case that the 'higher' layers — those that are the least trusted in most systems — are the ones which are ultimately trying to accomplish real tasks such as editing, compiling or performing a file transfer. All the lower, more-trusted layers are really intended to help the higher layers do their jobs. As we shall see in later chapters, it is often the case that information available to the higher layers would be of great value to the lower layers, were they able to depend on it.

It is our contention that, in the case of layered network protocols, it is vital to the efficiency of the lower layers that they make use of (i.e., in some sense, *trust* in) the information that is available to the higher layers. It is our further contention that it is possible to do this for networks in such a way as to protect the integrity of the lower, more-trusted, layers. The integrity of the lower layers is protected by 'sidestepping' the multiplexing of the lower layer. Higher level information is used by the lower level software in such a way as to benefit or harm only the particular higher level entity that supplied it. If the higher level information is wrong, only the entity that supplied it is harmed.

2.4 Further Classifying the Layers

We find it useful to classify the different layers into three kinds: Application layers, Transport layers and Network layers.

The application layer is distinguished as the top layer in the system. By this we mean that it is the highest layered part of the code that actually deals with the network. It forms the interface between non-network based structures — those of conventional systems — and network-based structures — protocols. Application protocols are typically concerned with concrete actions: transferring a file, manipulating a bitmap, calling a remote procedure.

The bottom layer of the protocol hierarchy is referred to as the *network* layer. It forms the interface between the software of the communications system and the hardware. We feel that it is useful to distinguish the point of transition from software to hardware because it gives a convenient "bottom level" to the protocol hierarchy. It tells us what the implementor starts off with, so we can understand the difficulty in achieving a particular degree of service.

It is hard to characterize the network layer of a protocol hierarchy. Network hardware differs widely, from unreliable packet-based networks such as the Ethernet [8], to networks which provide for the reliable delivery of sequenced bytes of data, such as Tymnet [34]. As mentioned in chapter one, we assume in this thesis that the network layer has the characteristics of a packet-switched network which transmits packets of data with a high probability (but not certainty) that they will correctly arrive at their intended destination.

What is left are all the layers between the network and application layers. These are referred to as the *transport* layers. They provide a progressively refined network service, and multiplex the network layer among different applications, so that each application need use only those transport layers that are helpful to its task at hand.

Transport protocols are difficult to design and implement efficiently because they are generally not aware of the concrete actions which are occurring at the layers above them. The problems exhibited by layered protocol implementations, which form the subject of the next chapter, occur in the transport layers.

Chapter Three

Problems with Layered Protocols

In the previous chapter we discussed layered protocols. We were concerned primarily with the arguments in favor of protocol layering. In this chapter we round out the view of layered protocols by examining some efficiency problems that are present only in layered implementations of protocols.

The presence of added layers of software between the application and the hardware implies that layered protocols come only at a cost. Each protocol layer maintains its own state information. Keeping this information in order consumes processing time and storage, and the need to transmit bookkeeping information to the cooperating foreign entity increases slightly the size of each packet transmitted. Other costs include an increase in the number of procedure calls and the number of levels of indirection needed to access data structures. We would thus expect layered protocols to run more slowly and at a higher cost to the computer system than non-layered ones.

The costs we have mentioned are minor ones. Studies have indicated that the cost of sending a single packet is dominated by factors which are independent of the size of the packet [10], so the small added size of the various protocol headers will not slow down communications considerably. Similarly, added levels of procedure calls and indirection represent only a small multiplicative decrease in efficiency. The tradeoff of an increase in software modularity for a linear decrease in run-time efficiency has become increasingly attractive as the cost of digital hardware and high speed communications equipment has decreased with time.

Experience with layered protocol implementations has indicated that layering is the cause of a considerably more severe decrease in efficiency than could be attributed to the costs mentioned above. In the worst case, the number of packets sent over the network increases exponentially with the number of protocol layers. In this chapter we consider the cause of this decrease in efficiency — which we entitle the *asynchrony problem* — in detail.

We examine some of the techniques which existing layered protocol implementations use to avoid the asynchrony problem. All these techniques have one thing in common: they result in a protocol implementation which does not resemble the structure implied by its specification. To produce an implementation of a protocol which runs acceptably, it is necessary to rely on information other than that which is available in the protocol specification. Sometimes, this information has been documented (for example, [10]); most often it must be learned by trial and error over a long period of time. Since, as we stated in section 1.2, we believe that protocol efficiency is a global rather than local concern, we consider that previous sentence really says that the protocol specification no longer indicates how to implement the protocol.

3.1 Failures in Distributed Systems

A failure, as the term has been used in this thesis, is an exceptional condition which either was not or could not be planned for when a program was implemented. If, when the program is run, it encounters this condition, it has no choice but to terminate execution: it must fail. A failure may result in a controlled software abort or in a complete "crash" of the process executing the program in question. The advent of type-safe languages [16] has blurred the distinction between these two, and we will consider them equivalent.

Failures are a part of any software project. Some failures arise from the truly random failings of hardware: cosmic rays, mechanical connections which are bent by heat, decaying electrical characteristics of the hardware components, and so on. Other failures represent software conditions which were considered at implementation time to be rare enough occurrences that it was not worth the effort of dealing with them. When considering the external behavior of a large software system these failures appear to be random as well.

Consider the case of two processes communicating with each other over a communications network. As long as both processes are alive, each may verify that the other is well by sending it a message and waiting for the response. If one process fails, we may assume that it could not notify its cooperating partner of this fact (even if it could, the message could still be lost during transmission). The remaining process will continue to operate properly, until it reaches a point in its execution at which it requires information from its now-deceased cooperating partner. Since no such information will ever arrive, the process will hang forever on network input. Although the process has not failed in any sense which is apparent to the local system, the global situation is such that it will never again perform any useful function. In effect, it has also failed.

When a process is running, it reserves system resources for its own use, such as a time-slice of the processor, memory, and I/O channels. If it either terminates successfully or fails, the system is able to reclaim these resources so that they may be reused. The danger of the situation described above is that, while the process has effectively failed, neither it nor the operating system is capable of verifying the failure; the system resources it has reserved cannot be reclaimed.

An analogous situation is that of a program with a bug which causes it to loop forever. It is impossible for the operating system to tell the difference between a program which is looping *forever* and one which is simply looping for a long time because it has much to do. A common solution to this problem is for the system to limit the amount of time that a program can run before it is assumed to be in an infinite loop, and is killed by the system. Similarly, in networks, we solve the problem of the distributed failure by introducing the notion of time into the implementation of network software. Before waiting for network input, a process makes a "worst case" estimate of how long it will take for the expected input to arrive, and sets a timer for this amount of time. If the timer expires, the process assumes that a failure has occurred in its remote partner, and aborts itself. We refer to a timer with this function as a *death timer*.

3.1.1 Death Timers

The length of a death timer is ideally the sum of:

- the time for a request for information (if any) to reach the foreign partner.
- the time for the foreign partner to compute the data that is expected, and send it to the requesting process.
- the time for the expected data to be transmitted between the foreign partner and the requesting process.

It is clearly not possible for a local process to know any of these times except under very restrictive circumstances. Thus a process must endeavor to estimate the delays both in the network and the foreign partner. There are a variety of existing algorithms which allow a process to estimate these delays (for example, the technique described in the TCP protocol specification [33]) which are not of interest to us here.

There exists the possibility that the estimate for the worst case delay will be long enough so as to be dangerous; a failure would cause resources to be tied up for long enough to disable the computer system, under normal loading conditions. This is usually the case when the worst case delay is several orders of magnitude greater than the expected delay.

Consider a protocol for transmitting electronic mail to foreign hosts which makes use of a reliable stream protocol — the same reliable stream protocol that is used for remote login to the host. Because of system restrictions, the number of virtual streams that may coexist on the system at any time is limited to some fixed number. Normally, when mail is sent to a foreign process, the process will store the mail on disk, confirming the transmission in a second or less. In the worst case, however, we can envision a host that prints the message on a slow printing terminal. Mail to this host may take as long as an hour to complete, so we set the death timer for the mail process at one hour. Now assume that some foreign mailer has a bug, such that it accepts the initial transmission of mail, but then fails without confirming that the message was completely delivered. Because the local mail process' death timer is very long, a virtual stream will be reserved for one hour after each attempt to mail to this buggy mailer. It is easy to see how the local mailer process may suddenly "lock out" remote login requests for an hour or more.

Because of the problems caused by the great variance in worst case times, the length of the death timer must also reflect the investment in system resources that it protects. This includes such factors as the global demand for the resources that are locked up by the connection, and the cost of aborting the connection when it did not actually have to be aborted.[11] In a layered protocol implementation, each layer reserves some system resources. Since every layer sends and receives messages over the network, every layer could potentially hang on input, with all the other layers hanging on it. Thus, to protect the resources that it has allocated, every layer must maintain its own death timer.

It is worth pointing out that death timers are used in a manner which is different from the way in which most timers are used. Usually, a process sets a timer to notify it about a needed real time delay. When a timer is set, the process that set it blocks until it expires. Thus, timers are manipulated rarely (because the time the process is blocked is typically great compared to the time spent initializing the timers), and are rarely canceled. Death timers do not follow this model. A network entity sets a death timer as a safety measure. Death timers are frequently being set and canceled, with the hope that they will not expire before they are canceled.

3.1.2 Optimization Timers

In a layered structure, each layer tries to provide an improved network service for the next higher layer. Often this improvement requires that the layer compensate for random errors or delays in the network. To do this it must find out about the loss of a packet quickly, so that it can retransmit the packet. Specifically, it must detect any transmission errors it wishes to correct before any death timers expire.

Death timers, as we have seen above, are typically set for worst case timing values. When an error is correctable, it need not be detected only in the worst case. Such errors may be

[11]The cost of aborting the connection when it did not need to be aborted is usually simply the cost of re-establishing the connection; it might also include the cost of backing out of other tasks that were tied to the activity being performed over the network.

detected more quickly by setting a timer for some value which is close to the *expected* time for a response to arrive. Such a timer is known as an *optimization timer*, since it improves the practical situation, but has no effect upon an ideal network in which no errors occur.

Optimization timers are used as follows. When a layer sends a message, it sets an optimization timer. If the expected response arrives before the timer expires, the timer is canceled. If the timer expires, the layer performs some corrective action, such as retransmitting the message. Since the optimization timer is set faster than the worst case delay, it will occasionally expire when there was no transmission error. Protocols which make use of optimization timers are designed to be resistant to the possibility that the timer expires when no error has occurred. A common technique is to use the optimization timer as no more than a *hint* [15].

3.1.3 The Timer Problem

In the preceding sections, we have shown that there is a need for two kinds of timers in layered protocol implementations. Death timers are used to avoid a deadlock situation whereby a process hangs on network input forever, locking up system resources that are needed by other processes. Optimization timers are used by some layers as a means of providing greater network reliability than would otherwise be available.

The added cost of death and optimization timers does not slow down protocol implementations to any considerable extent, but does increase the complexity of protocol implementations. A layer which wishes to wait on some single event must now wait on two or more events: the expected event, a death timer, and possibly an optimization timer. It is interesting to note that many existing operating systems make it awkward for a process to wait on more than one event.

We recall that in a layered protocol architecture, there are death timers at least at every layer of the structure, which exist so that the layer will be able to relinquish the resources that it has reserved for itself, should any other layers die. Many of the timers that are set at any time in a layered architecture are useless, because other timers, which will go off before them, are more significant. In the case of death timers, it is almost always the case that a failure in *any* layer will eventually cause all layers to fail.[12] Thus all but the shortest of the death timers in a layered structure might more reasonably be truncated to the value of the shortest timer.

[12] We ignore the occasional situation in which a lower layer may be replaced transparently by a different layer which performs the same function. This may occur, for example, when two different networks, with two different protocol architectures, are connected to the same computer.

Optimization timers are prone to the same sort of redundancy. If the expiration of an optimization timer causes some layer to retransmit a message, all lower layers are able to transmit a message of their own "for free" by piggybacking it upon the higher level message. Any optimization timers which had not yet expired at these lower layers might as easily not have been set, since the new packet which was prompted by an earlier timer has obsoleted their intended function.

Optimization timers also increase the asynchrony of the layers in a layered protocol implementation, by introducing events into the lower layers of a hierarchy which are not reflected by any event at the higher layers. This lack of coordination among the layers will be central to our presentation of the asynchrony problem in the next section.

The problem of the overabundance of timers in a layered protocol implementation is known as the *timer problem*. In chapter six we will indicate how this problem may be alleviated by getting rid of redundant timers, using soft layering.

3.2 The Asynchrony Problem

Consider the layered structure in figure 3-1. Two hosts, A and B, are communicating using a layered protocol hierarchy. There are three protocols in the hierarchy, whose actual identities are not important here; we refer to them as the application layer, the transport layer and the network layer, as indicated in the figure.

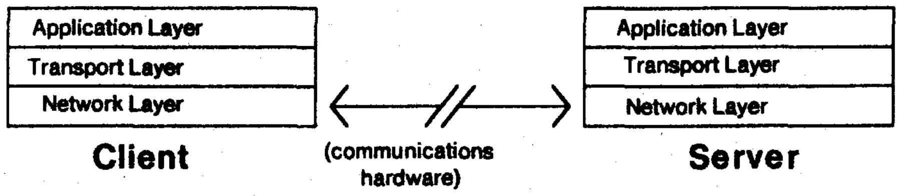


Figure 3-1: An example of a layered hierarchy.

Imagine a scenario, represented in figure 3-2, in which A's application layer protocol wishes to send a single byte of data to its cooperating application layer at B, and expects to receive a single byte of data in response. This scenario is rather common in layered protocols; it is exemplified by a remote login protocol performing remote echo [22].

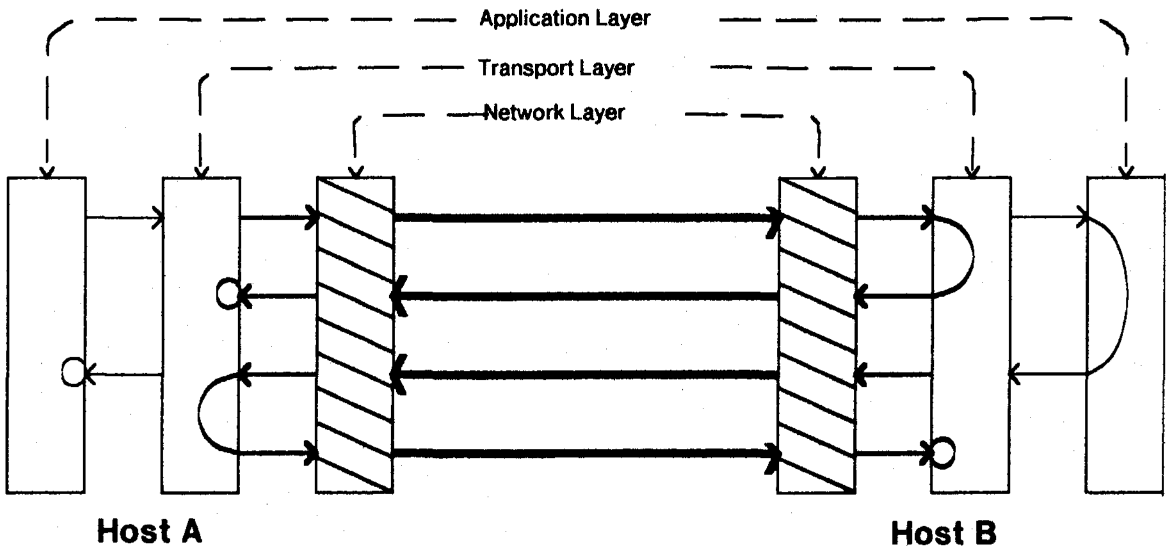


Figure 3-2: The application protocol sends a single byte of data to its cooperating partner

The local application layer hands its initial byte of data to the local transport layer. Let us assume that the particular transport layer in use must maintain a running dialogue with its cooperating partner (at host B) in order to keep the states of the two cooperating entities in agreement (for example, the two transport layers might need to coordinate what higher-level data had been successfully transferred from A to B, what data was yet to come, how much data could be transmitted at a time, and so on). A's transport layer allocates some position for the data in its internal bookkeeping system (perhaps a sequence number), so that it will be able to discuss the progress of the byte with B's transport layer. The transport layer then surrounds the application layer's byte with a protocol header and hands it to the network layer. Ignoring what happens in the network layer (and if all goes well), B's transport layer is presently prodded by B's network layer, and handed the same data.

As B's transport layer processes the transport protocol header, it notices that there has been some change to the state of the transport layer at A. To be able to update its own state to reflect this change, it must indicate to A's transport layer that it has received the state update. Thus it formats a protocol header indicating that it has received this latest state information, and sends it off to A's transport layer. Finally, it updates its own state, and hands the original byte of data to B's application layer.

The above is now repeated in reverse. B's application layer hands a byte of data down to B's transport layer, which sends the byte across to A via the network layer. As before, A's transport layer must update its state to the new situation, and must coordinate this change in state with B's transport layer, so an additional packet is sent over the network from A to B. Finally, the responding byte of data is handed to A's application layer.

Assuming that no packets were lost by the transmission medium[13]only two packets were really needed to transmit one byte of data from A's application layer to B's application layer and to return B's byte of data: one packet to send the initial byte of application level data, and one to send its response. The actual number of packets sent was four, however, since two packets were needed at the transport level to coordinate state information of interest only to the transport layers. If we use the number of packets transmitted as a meter of efficiency, the transport layer protocol has caused a twofold decrease in the efficiency of the application-level operation.

Let us examine the point at which the trouble started, i.e., the point at which the transport layer decided to send the first of its two 'wasted' packets. By considering the overall situation, we can see that the most efficient thing for the transport layer to have done when it received an update to its state was to do nothing but hand the application-level data to the application layer. If it had done this, it would have been able to "piggyback" its confirmation of the state update with the application layer's responding byte.

This, sadly, is not a technique that works for all situations and all possible application layers.[14] From the point of view of the transport layer, the application layer might as easily never return any data, or might only send data after a long delay. In fact, since A's transport layer has set a death timer, an application layer that returns data after a delay which is longer than the value of the death timer belonging to A's transport layer appears to the transport layer as if it never returns any data.

Furthermore, if A's transport layer is multiplexed, it may be dangerous for it to wait for the application layer to send its application-level response — the application layer is a less trusted layer than the transport layer. For example, if B's application layer fails and its transport layer

[13]This is not to say that the transmission medium is reliable; we infer only that no packets were lost in this instance.

[14]The reader who is unfamiliar with layered protocols is reminded that a lower layer has no way to know which higher layer is using it. This is sound software design for any hierarchical system, since it allows for the design of new higher level modules without modification to either the specification or implementation of existing lower level modules.

hangs waiting for the application layer, service to other application layers which make use of the same transport layer might be degraded or interrupted. Because these application layers have set death timers, an interruption of service could cause them to fail.

Even if the transport layer is not multiplexed, A's transport layer might have set an optimization timer. If this is allowed to expire (which is rather likely given even a small delay in B's transport layer) A's transport layer will retransmit the original message, so any savings that B's transport layer might have hoped by gain by piggybacking its response are nullified.

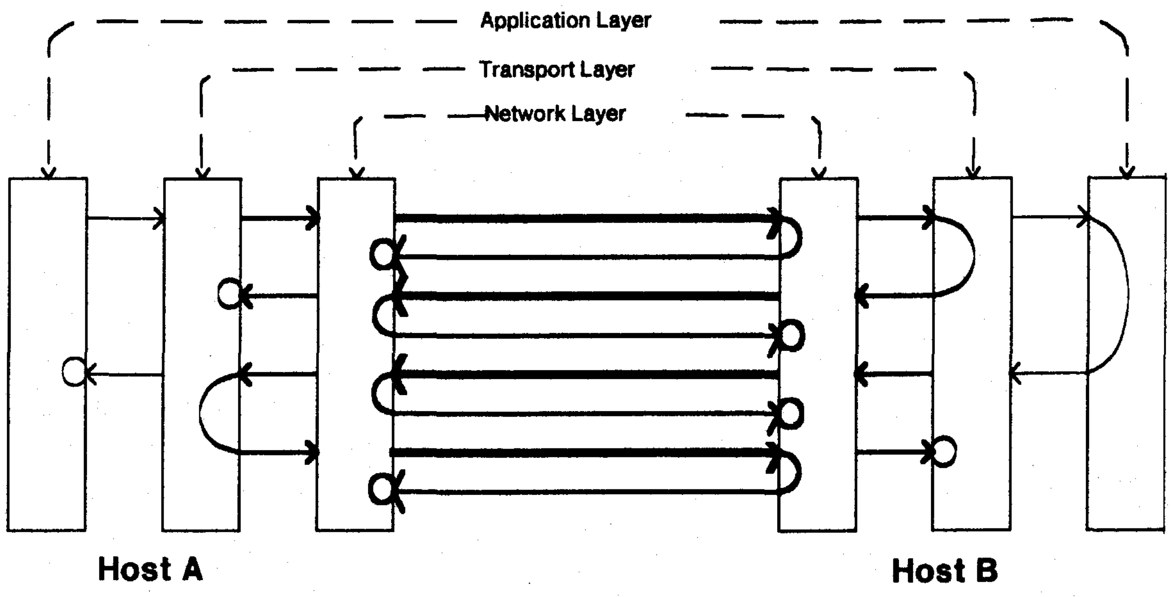


Figure 3-3: The Scenario of Figure 3-2 with the Network layer also Performing Bookkeeping

Certainly, this problem can cause protocols to be less efficient, but two extra packets do not "make or break" a communications facility. The severity of the problem is better seen by considering the possibility that, in the above example, the network layer protocol also requires a running transfer of state information.[15] For each of the four packets handed from the transport to the network layer above, the network layer adds one of its own. The total number of packets transmitted would then be eight. This more complicated version of the scenario is illustrated in figure 3-3.

[15] An example of a network layer that is of this complexity is the X.25 protocol [35].

The number of packets sent over the network can be seen to be exponential in the number of layers of protocol. To be precise, the worst case number of packets transmitted to accomplish a byte-for-byte exchange at the application layer (where 2 packets needed to be sent) is 2^M, where M is the number of layers in the protocol hierarchy that need to coordinate state information between the cooperating peer entities.

Not all of the layers in a typical protocol hierarchy need to coordinate state information by confirming every state update. For example, a layer which uses a checksum of higher-level data to catch and discard packets which were corrupted during transmission need only agree once with its partner on the particular checksum algorithm to be used. No subsequent coordination of state is necessary. In the expected case, of M layers in a protocol hierarchy, only some smaller number, N, of them might actually need to transfer state information on a packet-for-packet basis. For a two packet exchange at the application layer, the number of packets sent by this "expected" hierarchy would be 2^N: still an exponential.

In some existing protocol hierarchies, the number N is small enough that the effects of the asynchrony problem are not prohibitively expensive, although they are noticeable (a twofold decrease in the efficiency of a process in an operating system kernel is hard to miss). Unless the exponential decrease in efficiency due to the asynchrony problem can be avoided, protocol hierarchies will be severly limited in the degree to which they can accommodate new and more complex protocols as the need for them arises.

The situation in layered protocols whereby the number of packets sent for a simple higher-level operation increases exponentially with some function of the number of layers in the hierarchy is known as the *asynchrony problem*.

3.3 The Asynchrony Problem and Protocol Efficiency

We have seen how the asynchrony problem affects the number of packets transmitted; how does this relate to the efficiency of the protocol implementation? Two of the most common metrics of protocol efficiency are throughput, the number of bits transferred over the connection per second, and system load, the number of CPU seconds consumed by the protocol software.

The throughput of a protocol connection is the rate of data transfer that was actually achieved over the connection, regardless of the hardware or software that intermediated in the activity. In layered protocols, throughput must be measured relative to a given layer. The throughput of a large data transfer at a given layer is computed as the number of bits transferred *at that* *layer*, divided by the total elapsed time for the transfer (which is roughly the same at all layers[16]).

The throughput relative to the application layer is the throughput for the entire connection, which would be the only number of interest in a non-layered protocol implementation. At each lower layer, the amount of data transferred includes all the data sent at the application layer, plus the extra data sent as control information by the intervening layers between the layer of interest and the application layer. Since the amount of time for the transfer remains the same, the relative throughput increases at lower layers. The relative throughput at the network layer is the maximum throughput that can be had from the particular network interface and network in use. In what follows, we assume that the application layer is capable of sending data at a rate greater than the relative throughput at the network layer.

When the lower layers make optimal use of piggybacking, i.e., when *every* lower-level message is piggybacked on some higher-level message, the relative throughput between each layer changes only insofar as is necessary to accommodate the small increase in packet size at each layer which is caused by the protocol headers of the previous levels. Assuming that all the protocol headers are roughly the same size, the change in throughput would be a small linear decrease from the network layer, which sees the highest throughput (since it transfers all the protocol headers, as well as the application's data) to the application layer, which sees the lowest (since it transfers only its own protocol header). This situation is illustrated in figure 3-4.

In a real-world situation, it is not possible for all lower-level messages to be piggybacked on higher-level responses. Since the network interface presents a constant bandwidth to the base of the system, the time spent transmitting these lower-level packets is time during which all higher levels are 'robbed' of the chance to send a packet of their own. The change in relative throughput between each layer and the one below it is no longer constant, since the change depends on the number of extra messages that the lower level sent on its own. The loss in throughput between the application layer and the network layer is greater than the optimal case, above. This situation is illustrated in figure 3-5.

In the situation presented by the asynchrony problem, each layer sends one message which is

[16]The difference in the length of the connection at different layers corresponds to the time needed for higher layers to set up their state *before* establishing the lower level connection. This time is usually insignificant compared to the total connection time. We ignore situations where the setup time for the connection takes on the same order of time as the time the connection is used. Applications with this property should probably be using connectionless transfer techniques.

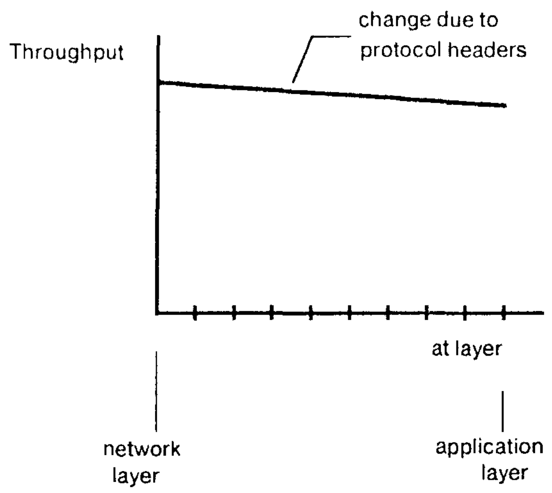


Figure 3-4: A Graph of the "Best Case" Throughput Relative to Each Layer

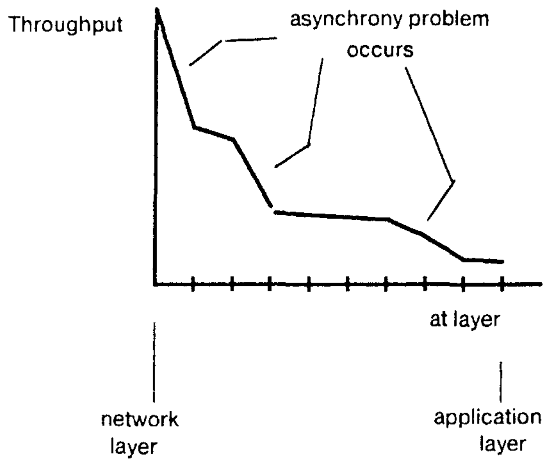


Figure 3-5: A Graph of the "Expected" Throughput Relative to Each Layer

unique to its layer (i.e., is not piggybacked) for each message handed to it by the next higher layer. Thus, the relative throughput decreases exponentially from the network layer's throughput, with an exponential base of two. This is illustrated in figure 3-6. It is clear from the graph in the figure that for any appreciable number of layers, the throughput seen by the application layer is minuscule.

Numbers make this discussion clearer. If the network layer can send 14 packets of 4096 bits each per second (which corresponds approximately to a 56 kilobit throughput), an application layer separated by six layers (all of which maintain state information) would be able to achieve no more than 875 bits per second throughput, assuming that all the packets sent are the same size. The throughput seen by the application layer is thus less than that of a fast terminal line. Even if the network layer is capable of sending at one megabit per second (such as might be the case for a local area network, ignoring operating system delays), the same application layer could still achieve no more than 15 kilobits per second throughput.

In the more usual case, most packets (and especially the "spurious" packets which are characteristic of the asynchrony problem) are much smaller than 4096 bits. Nevertheless, experience has shown that the most significant time delays in sending and receiving packets are independent of the length of the packet being transmitted. For example, metering of network software has indicated that the longest single delay encountered in the transmission and receipt of a packet is the time needed for the operating system's process scheduler to transfer control to the network process. Similarly, the time needed for a computer to prime a network interface's DMA hardware is usually longer than the time consumed by the DMA transfer itself. In practical terms, the network software of most operating systems is most accurately characterized by the maximum number of packets that may be transmitted per second, independent of the lengths of the packets. We would not expect that the results of the above example would change very much if we were to relax the restriction that all packets transmitted are of the same size.

A protocol implementation that does not avoid the asynchrony problem will end up consuming most of the network bandwidth available to it by sending messages at lower layers that do not achieve any function of use to the application (since the information in them could have been sent "for free" if piggybacking had occurred). If we presume that there is some fixed cost in CPU time for sending a packet (such as process scheduling time, and the time spent parsing the protocol headers), it is easy to show that this application will also consume much more CPU time than would be necessary were the asynchrony problem not occurring. It is thus necessary for any *real world* implementation of network software to either avoid the asynchrony problem completely, or to minimize the number of layers that communicate state information with their remote partners.

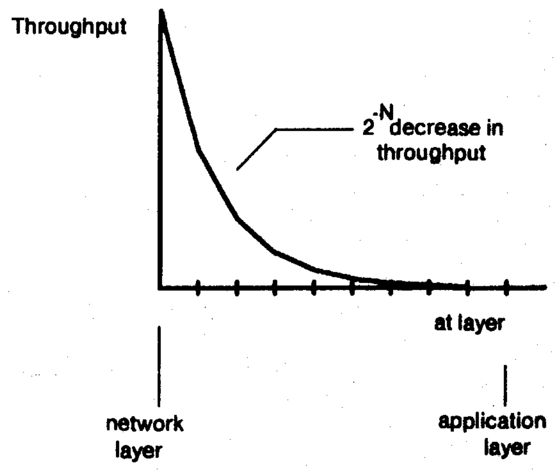


Figure 3-6:A Graph of the "Worst Case" Throughput Relative to Each Layer

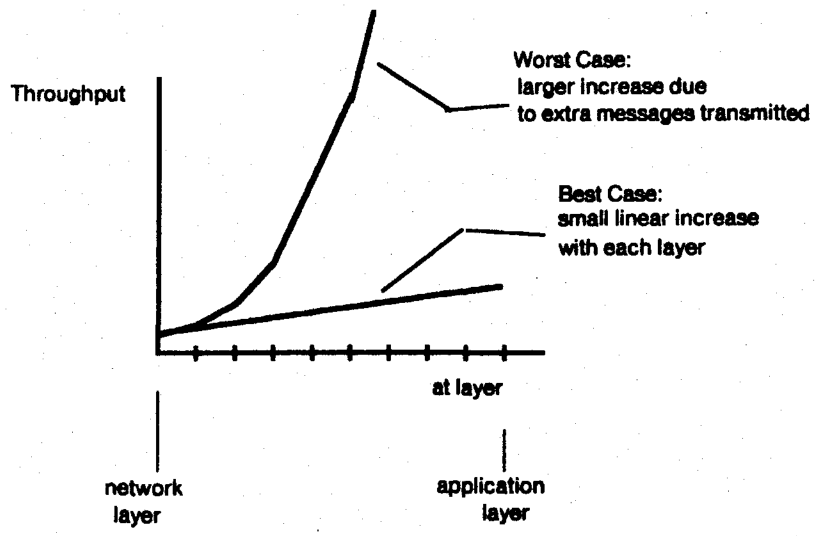


Figure 3-7: Graph of CPU Time Due to Each Protocol Layer

Finally, we note that in many data communications networks, billing for network services is based on a charge per packet. On a network of this sort, the monetary cost of the network service is directly coupled to the effects of the asynchrony problem, and will rise exponentially if these effects are not controlled.

3.4 Avoiding the Asynchrony Problem: Existing Techniques

The last section indicated why it is imperative for protocol implementations to avoid the asynchrony problem. In this section, we examine some techniques by which existing protocol implementations avoid it.

The simplest way around the asynchrony problem is to restrict the number of layers which maintain quickly changing state information to one or two. This does not avoid the asynchrony problem, but rather restricts the implementation to a portion of the exponential curve which is shallow enough to be bearable. The defects of this approach are clear: it leaves no room for future expansion of the protocol hierarchy, and there are very few computer systems which can afford even a twofold decrease in the efficiency of a commonly used subsystem.

Another common technique is to do away with the layered structure of the design when implementing a layered protocol. It then becomes feasible to implement strategies such as (for the example in section 3.2) sending the transport-level response to a message only when it is certain that the application will not send its own response soon enough. Unfortunately, much of the advantage of using layered protocols is lost; a change to either the implementation or specification of any layer will require a careful examination of all layers.

More exotic strategies involve setting timers in the transport layer to try to predict the behavior of the application layer without depending on it's behavior. The scenario of section 3.2, extended to use predictive timers, would proceed as follows: when B's transport layer receives an incoming message, it sets a timer for a short interval, say 50 ms., passes the application data on, and waits. If the application level sends data within 50 ms., the asynchrony problem is avoided. Otherwise, when the timer expires, the transport layer can send its state update and is no worse off than a 50 ms. delay.

The problem with this strategy is that its probability of increasing protocol efficiency is not uniform over all application layers and all time. For example, if the particular application layer in use *never* sends any responding data, or even if it always sends responding data after exactly 51 ms., the timers cause more harm than good. One could envision a modification to the above scheme whereby the timer is adjusted over time to fit the characteristics of the particular application layer, i.e., by cascading a predictive scheme with another predictive scheme. We are not aware of any such scheme, and it would in all likelihood be a complex one.

3.5 The Real Problem

All of the techniques described in the last section have one characteristic in common. They are all outside of the protocol specification. We have shown how the asynchrony problem introduces an exponential decrease in the efficiency of layered protocols, which implies that it must be avoided in any layered protocol implementation that is to be useful. It follows that in any real world situation, a layered protocol specification does not contain sufficient information for an implementor to produce a usable — in the terminology of chapter one, a 'correct' and 'efficient' — implementation of it. This is the problem which this thesis seeks to solve.

In the next chapter, we see some evidence that the asynchrony problem can be avoided by increasing the flow of information downwards, from the application layer to the layers below it. This leads, in chapter six, to a general technique for managing this flow of information, by embedding extra information in the protocol specification. An added advantage of the technique is that it presents a means of allowing the protocol specification to formalize all the information necessary to produce a correct and efficient implementation of the protocol.

Chapter Four

Propagating Information Downwards

From the discussion of the previous chapter we see that the asynchrony problem leads to a serious loss of efficiency in layered protocol implementations. We conclude that any protocol implementation which is to provide a useful service must have some means of controlling or avoiding the asynchrony problem. In this chapter, we examine two very simple file transfer protocols, each of which avoids to the asynchrony problem by sharing information between the application and transport layers which is normally kept separate. The two protocols are the Trivial File Transfer Protocol (TFTP [29]) and the Blast protocol.[17]

The analysis of TFTP and Blast that is presented here is somewhat artificial since both TFTP and Blast are non-layered protocols; each provides both transport and application layer functions in a single specification. For the purpose of the discussion in this chapter, we separate each of the protocols into an application layer and a transport layer in a manner which is consistent with the definitions of application and transport layers given in chapter two. The 'artificial' protocols that we present below are in their own right a useful example of a layered design which by its nature avoids the asynchrony problem. In practise, it is not uncommon to modularize the implementation of each protocol in precisely the way we will divide their design.

We do not find it necessary to be excessively detailed or complete in our description of the two protocols (for example, we will restrict ourselves to file transfer from the client's process to the server's, even though both directions are supported in the two protocols). A more complete and accurate description of the protocols is not necessary for the discussion of the chapter.

[17]Both TFTP and Blast had their origins as experimental protocols at MIT. TFTP has since been named as a standard protocol in the Internet protocol architecture, whereas Blast, developed in 1982 by Dave Reed, remains an experimental protocol.

4.1 The Application Layers

The application layer of a protocol performs a translation between the structures of the network and the structures of the local file system. In the case of TFTP and Blast, the application layers translate between the representation of a file that is sent over the network and the representation of a file that is used by the local file system.

In both protocols, application-level communications take place over a connection which is established between a client process and a cooperating server process by the transport layer. The connection lasts for the duration of one file transfer. Unlike more complex file transfer protocols, TFTP and Blast do not support either directory queries or any form of user authentication or identification.

The application layer's connection is established when the client's application layer sends an initial message to the server's application layer. The client's initial message specifies the name of the file to be transferred. In Blast, the initial message also contains the length of the file. In TFTP, the length is replaced by a *transfer mode*, which is not of interest here.[18]

If the server does not wish to accept the connection, it may abort it by sending an error message back to the client. An error message always aborts the connection, and is the only error reporting mechanism available in the two protocols.

Assuming that the server accepts the connection, the client then sends the entire contents of the file to be transferred. The file is broken down into 512-byte "blocks" of data for transmission, each of which is sent as a single message (and usually is inserted into a single packet). In TFTP, the end of file is marked by a block containing less than 512 bytes. In Blast, the length of the file has been transmitted to the receiver.

As should now be clear, the application layers of TFTP and Blast are essentially the same. Since our concern here is primarily with the transport layers, we will not bother to make a distinction between the application layers of TFTP and Blast in our discussion, except where such a distinction is needed.

[18]The transfer mode specifies a transformation of the file that is to be performed between reading the file from disk and writing it to the network. The most common transformation involves converting local file system newline conventions to a network standard one. This is the so-called Netascii convention, described in [22].

4.2 The TFTP Transport Layer

TFTP's transport layer makes use of the User Datagram Protocol [21], which provides a verified but unordered and unreliable[19] datagram service between two cooperating processes. The transport layer must therefore be resistant to any loss of, reordering, or duplication of transmitted packets.

TFTP was designed with the goal of having a simple design and supporting simple implementations. This is reflected in its transport layer. The transport connection is initiated when the client's transport layer sends an initial message to the server's transport layer. Once the initial message is sent, an optimization timer is set. If it expires, TFTP's transport layer assumes that its initial message was lost, and the message is retransmitted. This sequence is repeated until either the server's response is received or or the client decides to give up and terminate the connection (i.e., TFTP implements its death timer by counting some number of successive expirations of its optimization timer).

When the server receives (and accepts) the initial message it sends back a message to acknowledge it, called an ACK message. The ACK message is a 'go ahead' signal to both the client's transport and application layers, indicating that the file itself may now be sent.

As mentioned above, the application layer splits the file into blocks of 512 bytes. The client's transport layer operates by sending the first block of the file as a single datagram, receiving an ACK for it, and passing the ACK to the application layer (which produces the next block of the file), sending the next block, receiving an ACK for it, and so on. No more than a single data block may be sent between consecutive ACK messages. If an ACK fails to arrive for a block, the datagram containing the block is retransmitted until either the ACK arrives or the client gives up and aborts the connection. Messages containing data blocks and ACK messages both contain the number of the block of the file to which they refer, so that delayed duplicate packets may be discarded.

Note that there is a symmetry between client and server in TFTP. Each ACK message is an acknowledgment that the previous data message has been received, and each data message is an acknowledgement that the previous ACK message has been received.

The last block of the file may be recognized by the server because it is the only block of the

[19]We use the terms 'verified' and 'unreliable' in the sense of the ISO Reference Model [11]. A protocol which ensures 'veracity' is one which ensures that the bits that are received are the same bits that were sent. A protocol which is 'reliable' ensures that the bits actually get to where they are supposed to go.

file which contains less than 512 bytes of data in it. The server's transport layer notes when this block has been handed to the application layer so that it may close the connection as soon as it sends the ACK.[20] The client declares the transfer successful as soon as it receives an ACK for the last block number of the file.

4.2.1 TFTP and the Asynchrony Problem

The asynchrony problem could conceivably come into play in TFTP whenever the transport layer receives a datagram. The datagram contains a block of data, and the foreign transport layer is waiting for the ACK for this block: should it send the ACK now or wait until the application layer has processed the block, so that the packet containing the ACK message has meaning to the foreign application layer as well?

In TFTP, the transport layer knows that the application layer will always have more data to send. If the datagram does not contain the last block of the file (or its ACK), then there is either another block of data to be sent, or an application level acknowledgment for this block. If it does contain the last block of the file or its ACK, then there is either an ACK to be sent, or there will never be any more data sent on this connection. Thus it is always safe to wait for the application layer.

The asynchrony problem can not occur in TFTP because the application layer has been constrained to meet the needs to the transport layer: whenever the transport layer must send an acknowledgment, the application layer must send one too.

4.3 The Blast Transport Layer

A Blast transport connection begins in a manner analogous to that described for TFTP. The client's transport layer sends the server's transport layer an initial message and receives one in reply. An optimization timer is used to control retransmissions of the client's initial message, and a death timer is derived from successive expirations of the optimization timer. The client may not begin to send the file to be transferred until the initial message is acknowledged.

Once the server's application layer has accepted the connection, and the server's transport layer has transferred the acceptance to the client, the similarity with TFTP ends. The client's

[20] Actually, it waits for a short while so that it may retransmit the ACK if the first one got lost in transmission.

transport layer proceeds to transmit all the blocks of the file, in sequence, to the server's transport layer. The server's transport layer keeps a map of all the blocks in the file that it has received. When it receives a block, it hands it to the application layer, which writes the block into the correct place in the file. It is usual for the blocks of the file to be transmitted in order, but this is not a requirement of the protocol. No optimization timers are ever set during the transmission of a file, and each block to be transmitted is transmitted but once.

When all the blocks of the file have been sent, the client's transport layer sends an end-of-file notification to the server's transport layer. The server's transport layer responds with a datagram containing a map of the blocks of the file that it has received. The reliability of the end-of-file/block-map exchange is handled in the same way as the initial message and its response, by controlling a series of retransmissions of the end-of-file notification with an optimization timer.

Once the client has received the server's map of blocks, a new pass over the file takes place in which the client sends only those blocks of the file that were not received by the server on the first pass. After this, another end-of-file notification is sent, and responded to with an updated map of the file. This repeats until the client has received a copy of the server's map that indicates that the entire file has been received. Both server and client then terminate the connection.

Messages containing data blocks are identified by their position in the file, so that they may be stored in the correct place. The server may use its map of the blocks it has already seen to detect and discard duplicate packets, or may rely on the idempotency of the disk-write operation.[21]

Lost packets are detected as gaps in the server's block-map, and are retransmitted on subsequent passes through the file. In fact, a server on a host whose file system takes more time for random than for sequential file access may choose to discard any packet which arrives out of order, with the hope that the block will arrive in order on the next pass through the file.

[21] An operation is *idempotent* if the effect of performing it any number of times greater than once is the same as if it had been performed exactly once.

4.3.1 Blast and the Asynchrony Problem

In Blast, as in TFTP, the application layer must send an acknowledgement exactly when the transport layer must send an acknowledgement. Thus Blast's transport layer may avoid the asynchrony problem in the same way as TFTP's, by always assuming that an application-level acknowledgment will be forthcoming shortly.

4.4 Considering Efficiency

Both TFTP and Blast avoid the asynchrony problem by allowing layers to make use of information about the nature of the layers adjacent to them. In Blast the transport layer is constrained to work in a way which is similar to the application layer. The characteristics of the task the application layer is performing are those of disk files: the ability to perform random rather than sequential access, and the idempotency of reading and writing a disk block. For example, Blast places no restriction on the number or order of data blocks that are sent, since there is no intrinsic best order for the application layer to use. In many file systems, a file is not stored on the disk in contiguous sectors. It is possible for a Blast implementation that uses this kind of file system to increase its speed by transmitting the file in the order that it is stored on disk, rather than by the logical order of the data in the file.

The idempotency of disk reads can be used by a Blast implementation to restrict the amount of internal buffering that it must maintain. From a description of the protocol, it would appear that the client's transport layer must save a copy of every message that it has sent on each pass through the file (which means that it must save a copy of the entire file on the first pass), since the foreign transport layer can request that any message be retransmitted. In practise, the transport layer implementation can make use of the application layer's ability to re-derive the original data quickly (by rereading it from the disk) to avoid storing arbitrary numbers of messages for retransmission. Rather, when the transport layer must retransmit a given message, it hands a request to the application layer to reread, reformat and resend the appropriate blocks of the file. Since the application layer can translate the block number of a message to a position in the disk file that is being transferred, it is not unduly difficult for it to respond to such a request.

In TFTP the interactions between the transport and application layers are the reverse of those in Blast: the application layer is constrained to make use of the transport layer, rather than the other way around. The characteristics that TFTP's application layer inherits from its transport layer cause it to work more slowly. Since the transport layer does not presume that the application layer's data can be regenerated, it can not rely on the application layer to recreate messages for retransmission. To avoid using up excessive memory in buffers containing messages that might need to be retransmitted, it constrains the number of messages that the application may have outstanding at any one time. In most transport protocols, this limit is less restrictive, but still there. For example, in TCP, the number of bytes of buffering that must be maintained by the protocol can be set to an arbitrary, but fixed, number. In practise, this number is usually set to several thousand bytes. Since the transport layer is not cognizant of the characteristics of the application layer, it is unable to benefit from them.

Blast sends many fewer packets than TFTP; in this sense, it is the more efficient of the two. In TFTP (if there are no lost packets), one ACK message occurs per data message; in Blast (under the same conditions), there is one ACK message for the entire file. This impacts the efficiency of the two protocols in a number of ways. A TFTP protocol implementation requires a separate process wakeup for each message; a Blast implementation can decrease the effective cost of operating system overhead in a large computer system by "batching up" messages so that several may be sent per process wakeup. TFTP requires that no more than a single block of the file be in transit at a time, so a TFTP implementation is unable to use the network as a buffering mechanism. Blast, when presented with a high speed, long-delay network (such as a satellite network), can buffer as much as the entire file in the transmission medium. Finally, a Blast implementation can make more effective use of the characteristics of the disk hardware by reading the file to be transferred in arbitrary order. While a TFTP implementation could perform a similar optimization, the amount of buffering required might make it difficult for it to do so. For example, when the two cooperating processes are matched in speed and are connected by a network which rarely loses packets, and whose transit time is small (so that the network round-trip time is dominated by delays inside of the hosts), Blast can be expected to achieve more than double the throughput than TFTP, since TFTP must wait double the host-to-host transit time between blocks.

TFTP is less efficient an Blast because it solves the problems of the transport layer by constraining the application layer, when it is the application layer's problem (that of transferring the file) whose accomplishment is paramount. The characteristics that the application layer 'inherits' from the transport layer protocols are artificial insofar as they do not reflect the needs of the global task.

4.5 TFTP, Blast, and Layering

Both TFTP and Blast share information across layered boundaries in a way which avoids the asynchrony problem. There are two conclusions to be drawn from this fact. One is that it is possible to avoid the asynchrony problem by increasing the flow of information between the layers. The other is that an increased flow of information 'downwards' from the application layer to the transport layers is preferable to an increased flow 'upwards' for the sort of protocol issues which are of concern in this thesis.[22]

There is some harm in the approaches of TFTP and Blast. An increase in the amount of information that is shared across layered boundaries decreases the modularity of each of the layers. A change in the application layer of Blast could result in the need for a change in Blast's transport layer; the same is true for TFTP's transport and application layers.

The reader might have noticed that Blast's application layer would run on TFTP's transport layer, but not the reverse. The implication of this is that an increase in the flow of information downwards is more harmful to the modularity of a layered structure than an increase in the flow upwards, since the application layer can always operate correctly under a more restrictive set of conditions, albeit less efficiently. We reject this argument, since we consider an inefficient implementation to be no more useful than an incorrect one.

One problem that appears in Blast but not in TFTP involves the degree to which lower layers place trust in higher layers. Specifically, Blast's transport layer relies on its application layer to help it reconstruct any messages that must be retransmitted. In section 2.3 we outlined the danger in this, that a failure in the application layer could propagate down to the transport layer, and hence up to other application layer protocols, leading to a more severe failure than was expected. In this particular example however, a failure in the application layer can not cause undue harm to the system. Assuming that the transport layer is properly designed, the only effect of a failure in the application layer would be a failure of the transport layer *to perform the service that the application layer has requested.* In effect, the trust that the transport layer places in the application layer is a 'two-way contract,' stating that the transport layer will agree to provide the desired service *as long as* the application layer agrees to help. This method of allowing lower layers to place trust in higher layers will form an essential part of the arguments of chapter six.

Because of the decrease in modularity that stems from the increase in information flow between the layers, it is necessary to restrict carefully both the nature and amount of

[22] In other situations, such as congestion control, the reverse is true (see section 8.1.2).

information which flows across layer boundaries. The development of a method for doing this is the subject of chapter six. In the next chapter we seek to further generalize the results presented in this chapter.

Chapter Five

Problems with Layered Protocols (Revisited)

In chapter three an example of a layered protocol architecture is described in which the efficiency of an application layer protocol was impaired because of poor interactions between the layers. We name the source of the inefficiency seen in the example the 'asynchrony problem.' In this chapter we return to the discussion of the asynchrony problem begun in chapter three. Through the introduction of new terminology for describing the state of network entities, we find that it is possible to characterize both the asynchrony problem and its solutions in a more abstract way than is done in chapter three. This leads, in the next chapter, to the development of soft layering. Soft layering formalizes a class of solutions to the asynchrony problem which have the property that they do not cause the protocol implementation to abandon the layered structure of the protocol design.

5.1 Happiness and Unhappiness

In this section we define terminology for describing the state of a network connection in a more succinct way. The terminology is know as *happiness* terminology.

A network entity is said to be *happy*, if it has received confirmation that every action it has requested from its remote partner has been completed. It is said to be *unhappy* (or upset) if there is some action for which no such confirmation has been received. The act of causing a remote partner which is unhappy to become happy is referred to as *appeasing* it.

Different entities request different sorts of actions, and are thus appeased in different ways. A transport layer might typically be concerned with the delivery of the a body of data in a manner which preserves both the order in which the data was transmitted and its exact contents. This kind of transport layer would become unhappy when it transmitted data out over the network, and would be appeased by the receipt of an acknowledgment from its cooperating transport layer entity which indicates that the data had been successfully passed on to the remote application layer. Most application layers become unhappy when they request some action of a more complex sort: file transfer, disk access, terminal display, and so on. Their appeasement would indicate that the requested action had been accomplished.

A useful illustration of happiness and unhappiness in communications is that of a human user, say George, typing at a terminal which is connected to a computer. George wishes to type the command:

delete myfile

As George hits each of the keys needed to type the message, he is watching the screen to make sure that the correct character is echoed. Each keystroke is a request for the transfer of an ascii code to the computer. With each keystroke the part of George that is acting as a transport layer protocol becomes unhappy, and is appeased by the appearance of the echo for the keystroke.

When the command line is completed, George becomes unhappy once again, but in a different way. He has requested a concrete action from the computer, which has not yet been confirmed. In effect, the part of George that is acting as an application layer protocol is unhappy, while the part that acts as a transport layer is happy, since it does not expect any data. When the computer displays

"myfile" deleted.

on George's terminal, his 'application layer' is appeased.

5.2 Unhappiness and the Asynchrony Problem

We can now resume the discussion of the asynchrony problem that was begun in chapter three, using happiness terminology. In the example from chapter three, the application layer on host A wishes to send a single byte of data to the application layer on host B, and to receive a byte of data in return. The discussion of section 3.2 shows that the number of packets actually sent over the network in this case is an exponential function of the number of layers of protocol. In particular, when the application layer wished to send two messages, four messages were sent by the transport layer, and eight by the network layer.

Let us review the example again. The application layer at A wishes to send a single byte of data to its cooperating application layer at host B. A's application layer hands the byte down to its transport layer. Since the purpose of sending the byte of application-level data was to evoke a byte of application data in response, the act of sending the "requesting" byte of data causes A's application layer to become unhappy.

A's transport layer now receives the byte, allocates some name for it in its scheme of bookkeeping, and sends off a transport-level message containing A's application-level byte. In doing so, the transport layer becomes unhappy, since by transmitting the message containing application-level data it has implicitly requested an action of the cooperating transport layer at B, namely, to confirm that the change to the state of A's transport layer has been reliably transferred to B's transport layer.

Ignoring the details of the network layer, the transport-level message transmitted by A's transport layer is presently handed to B's transport layer. Upon processing the message, B's transport layer determines that the message contains a state update, which implies that A's transport layer is unhappy. It is the job of B's transport layer to appease its cooperating partner; furthermore, it is now in a position to do so, since it has updated its internal state to match that of A's transport layer: B's transport layer knows that the event for which A's transport layer is waiting has taken place. Thus, B's transport layer formats a transport-level message designed to appease A's transport layer, and sends it to its cooperating partner. Then it hands the byte of application-level data to B's application layer.

B's application layer examines the byte of data, and determines that it has caused A's application layer to become unhappy. The nature of the appeasement which is required by A's application layer is the responding byte of application-level data, so B's application layer sends the correct byte to its cooperating entity, via its transport layer.

B's application layer is now happy. B's transport layer sends a message containing the application layer's byte of data, but in doing so, updates its state and so becomes unhappy. When A's transport layer receives the message, as in chapter three, it must appease its cooperating entity by sending a fourth transport layer message confirming the receipt of the previous message.

There are a number of things that are visible in the above presentation that were harder to see when the same scenario was presented in chapter three. Consider the beginning of the scenario. The application layer at A sends a byte of data, which causes the transport layer at A to send a message. Both have requested remote actions, so both become unhappy *as a result of the same event*. It is significant that neither A's application layer nor its transport layer is aware that the other is unhappy. From the point of view of the transport layer, the application layer could expect a byte of data in return, or could not. Similarly, the application layer expects a service from the transport layer, but does not care (within the limits of the transport protocol specification) how the transport layer provides that service. In our case, the transport layer coordinates state information with its foreign peer entity, but it could equally be the case that it instead relies upon the reliability of the network layer, and is always happy.

Although the application and transport layers were both made unhappy by the same event, the nature of the appeasement that each expects is different. The application layer is expecting a byte of application-level data, and the transport layer is expecting a transport-level message. It is thus possible — and even natural — for the two layers to be appeased at different times, even though they are really waiting for the same event to occur.

The asynchrony problem is a side effect of the different way in which different layers view the same event. The asynchrony problem first occurs when B's transport layer receives A's incoming transport-level message. To B's transport layer, the application-level data bears no relation to the transport-level data; all it knows is that both data items happened to have arrived in the same packet. To be able to avoid the asynchrony problem B's transport layer needs to know whether or not the appeasement that it is sending to A's transport layer is linked to some later appeasement that B's application layer will have to send to its cooperating entity. That is to say, B's transport layer needs to know if the event that triggered the unhappiness in A's transport layer — the event that it is about to appease — also caused unhappiness in A's application layer. This information, as is illustrated by the example above, is precisely the global information that the technique of protocol layering has obscured.

The importance of the last paragraph is that it serves to characterize the particular information that would allow a protocol implementation to avoid the asynchrony problem. Specifically, *all solutions of the asynchrony problem involve providing a means for the transport layer to be informed of the happiness or unhappiness of the application layer*.

There are two general techniques to allow the transport layer to determine the happiness or unhappiness of the layer above it: the *design* technique, and the *parameterized* technique. In the design technique, the application and transport layers are designed in such a way as to guarantee that happiness-unhappiness transitions happen at particular, coordinated times. In the simplest example of this, happiness-unhappiness transitions always happen at the same time for all layers. In the parameterized technique, the higher layers pass down enough information to the lower layers to indicate when they are and are not unhappy.

In the rest of this chapter, we shall examine the design technique for avoiding the asynchrony problem. The parameterized technique is exemplified by soft layering, which is the subject of the next chapter.

5.3 TFTP and Blast: Designing Away the Asynchrony Problem

TFTP and Blast are two examples of protocols which design-away the asynchrony problem. Each does so in a different way.

In TFTP, the application layer is modified to become unhappy whenever it is expedient for the transport layer for it to do so. In TFTP's case, this means that both layers are kept unhappy throughout the transfer; a transition from unhappiness to happiness, then back to happiness, occurs with every message.

In Blast, the transport layer becomes unhappy only when the application layer is unhappy, i.e., after each pass through the entire file. When no packets are lost this happens only twice per transfer: once at the beginning, to make sure that the receiver is ready, and once at the end, to get an acknowledgment for all the transmitted data.

Each of these two techniques has its problems. TFTP, because it is always unhappy, sends one ACK message per data message. Thus it can not achieve high throughput except between lightly loaded machines connected together with fast local-area networks. Blast has no flow control during the transmission of a file, so it is only appropriate when the bandwidth attempted by the sender is no greater than that of the slowest network component in the path between client and server. There is no mechanism in most networks which allows the client to determine whether its sending speed is, in fact, matched by the communications path between itself and the server; in practise Blast is most useful in situations where the cost of sending a packet is very small, and the bandwidth of the network is sufficiently greater than that of the interfaces connecting machines to it that network congestion is not a problem.

Blast also performs poorly when faced with network failures that last for a long time, such as gateway crashes. Since the protocol contains no facility for marking the progress of a connection except between entire transmissions of the file, it is possible for a Blast implementation to transmit an entire file, expending enormous system (and network) resources, only to discover that a gateway crash has caused all but the first block of the file to fail to reach its destination.

The problems outlined above do not prevent TFTP and Blast from working and from being useful for a wide variety of applications. It should be noted, for example, that TFTP has been implemented and is in daily service on a number of machines, ranging from large mainframes to small personal computers. It is used for general file transfer, mail transfer, access to remote printers, and downloading of boot code into gateways. Over a period of six days, one large mainframe computer system at MIT initiated over six hundred TFTP transfers.

While it is true that TFTP and Blast can function usefully, the discussion of this section does indicate that neither of these two protocols will ever replace the more general transport-layer protocols such as X.25 [35][23] or TCP [33]. Neither TFTP's nor Blast's approach to solving the asynchrony problem could be made to work for a more generally applied protocol.

For example, consider the Transmission Control Protocol of the Internet protocol architecture. TCP is a reliable stream protocol that is used by a wide range of application protocols including protocols to perform remote login, file and mail transfer, and remote database access. For TCP to be useful for such a variety of applications protocols, it must restrict the characteristics of the application on which it relies to the very small common denominator of all its intended applications. Thus, very few characteristics of the applications for which TCP is used can be used to improve TCP's efficiency. TCP is used to connect widely varying systems connected to widely varying hardware, so it cannot use the more radical transmission schemes such as those employed by Blast. Still, it must achieve relatively high bandwidth at relatively low cost, so the halting but safe approach used by TFTP would never do.

TCP uses a windowing scheme so that (among other things) it is not necessary to require either TCP or higher level protocol implementations to have infinitely large buffer space. To work correctly TCP must become unhappy when its windows close. This has nothing to do with any other layer of protocol; to try to synchronize other layers' happiness to TCP's would make other protocols awkward. In effect, we would be requiring the application to do much of the work that the transport protocol was designed to do.

How can the asynchrony problem be prevented in a protocol with requirements so general as those of TCP? In the next section we present soft layering, which is solves the asynchrony problem in a way which is compatible with more generalized protocol structures.

[23]Note that X.25 contains several layers, not all of which are transport layers. The above statement refers to those parts of X.25 which are transport layers, for example, the X.25 packet level

Chapter Six

A New Solution to the Problem: Soft Layering

In the last chapter we examined the asynchrony problem in greater detail. Applying 'happiness' terminology, we were able to see that the asynchrony problem occurs as a side effect of the lack of coordination of the happiness-unhappiness transitions that is characteristic of existing layered protocols. We also described one technique for avoiding the asynchrony problem, called the 'design' technique. In this technique, the design of each layer in a protocol architecture is done in such a way as to guarantee that all happiness-unhappiness transitions occur at well defined times which are visible to all the protocol layers. While the design technique works, we note that it is not applicable to a wide class of protocols and applications.

In this section we examine a 'parameterized' technique for avoiding the asynchrony problem. In a parameterized technique, the design of the protocol architecture does not constrain the timing of the happiness-unhappiness transitions at the component layers. Instead, facilities are provided which allow layers to pass parameters to other layers which indicate when transitions in happiness occur. The asynchrony problem is avoided by allowing higher layers to make explicitly clear to lower layers when it is and is not safe for them to wait before appeasing their remote partners.

Parameterized techniques for avoiding the asynchrony problem have the property that they increase the flow of state information between the layers of protocol. This is a departure from conventional protocol design. Layered protocols are typically designed so as to keep the state information of the layers hidden from one another. One might say that normal practise is to is to establish *hard* boundaries between the state of one layer and that of the next.

As we describe in section 4.5, there is potential harm in an increase of information flow across layered boundaries: such an increase causes a decrease in the modularity of both the design of a protocol and its implementations. The advantages of a layered structure are clear, and we would not wish to do away entirely with layering. It would be a severe loss to the modularity of protocol design and implementation if the production of an efficient protocol implementation necessitated a violation of layering, even if the violation is restricted to a particular application protocol.

Rather than abolish layered boundaries, we merely *soften* them; each layer is given information about the state of the layer above it, but this information is accessible only through a properly specified interface. Behind the interface, the implementation (and much of the specification) of the higher level protocol is able to maintain its independence of all other layers. We hope in this way to improve the effectiveness of layered protocols without losing most of their advantages. We coin the term *Soft Layering* to refer to this technique, as a reminder of the softened, but still existent, division between the layers that is characteristic of soft-layered protocols.

6.1 Soft Layering

A protocol specification has two parts. The bulk of the specification describes the format of the data that may be transmitted, the way in which connections are established, and the correct way to maintain communications. The rest of the specification describes the environment in which the protocol is to be applied. In a layered protocol, this is generally a description of the layer directly beneath the one being defined. Often the description becomes very specific, to the point of requiring one protocol to be built on top of a particular existing protocol. For example, TCP [33] requires that the Internet protocol — and none other — reside beneath it.

Other protocols are not so particular. They give a more general description, a *model* of the protocol that lies below. The Internet protocol is an example. It requires of the next lower layer only the ability to send and receive packets to other hosts on the same network. It places limited restriction on the reliability of the network, the size of its headers, the nature of its hardware, and so on. The Internet protocol has been successfully implemented on networks varying over the entire range of hardware from local area networks such as the Ethernet [8] to satellite networks such as the Satnet [12]. It would even be possible to implement Internet using TCP as a base. The model that a protocol specification uses to describe the protocol that runs directly beneath it describes the minimal characteristics of the protocol's operating environment. We call this model the *environment model* of the protocol.

In a soft layered protocol, we add to the protocol specification a third component, a model of the layer above the layer being specified. As with the environment model, this model may be very general or very specific. The difference is that it is not required of higher level protocols that they fit the model. When higher level protocols *do* fit, the service they receive is more efficient. We name this model the *usage model* of the protocol. The usage model is a means of introducing the notion of protocol efficiency into the protocol specification.

From the point of view of simple network connectivity ("Can A speak to B?"), a protocol's usage model is no more than a hint. When efficiency considerations are taken into account — as we feel they must be — it becomes a stronger restriction. A protocol need not even try to be efficient when it is used in ways that do not fit its usage model. It may well be the case that a higher level protocol must fit into the lower level's usage model to obtain service of acceptable efficiency.

We recognize that the addition of a usage model to the protocol specification causes the protocol specification to be less general. It is our feeling that this generality never really existed. The usage model merely formalizes in the specification of a protocol much of the compendium of subjective suspicion and empirical evidence that is currently used by the protocol designer to determine whether a given lower level protocol is suitable for use.[24]

While the motivation for adding a usage model to a protocol specification is to make its implementation more straightforward and efficient, the presence of the usage model can also help in the design of a protocol architecture. For example, when a reliable delivery protocol specifies in its usage model that it is to be used as a tool for implementing reliable communications at a higher level, it is not necessary to implement a three-way handshake when closing a connection. When one higher level entity has indicated that it has finished communicating, the lower level connection may be aborted without further action, since it takes at least two higher level entities to support reliable communications.

6.2 How to Make Soft Layering Work

The reader will probably have realized that soft layering does not present a foolproof solution to the asynchrony problem. It is rather a mechanism for unifying the correctness and efficiency of protocols in the protocol specification, which presents the clever designer or programmer with a means of solving the problem. In fact, the mechanism provided by soft layering is also useful for solving the "timer problem" of chapter three. This section attempts to nail down somewhat the sort of usage model that is most effective in dealing with the problems of layered protocol implementations.

To allow the implementor to avoid the asynchrony problem, the usage model must provide the protocol being specified with access to the transitions in the happiness-state of the next

[24]For example, Internet Implementor's Guide, which explains (in the majority) what sort of efficiencies are important in a TCP implementation, is longer than the TCP specification.

higher level entity. This is almost always equivalent to revealing the higher-level divisions in the higher-level entity's transmitted data; transitions in the happiness-state of an entity must occur at a point where some higher level division of data is reached. The proof of this statement is straightforward:

> An entity becomes unhappy when it requests a particular action. To request the action, it must send data over the network. The point at which it becomes unhappy is thus the point at which it completes the request, i.e., at a higher level data boundary. Similarly, an entity is appeased when confirmation of an action is received over the network. The point of appeasement is when the higher level confirmation has been completely received, i.e., at a higher level data boundary.

Not all higher level data boundaries cause happiness transitions. A useful way for a usage model to specify where happiness transitions occur is for it to "offer" the higher level protocol a model of higher level data in which happiness transitions *do* occur at each higher level boundary. This is known as the *offered* data model.

The presence of an offered data model also helps in the specification of protocols. When a protocol is designed to run directly above another in a layered structure, much of its specification can take the form of a mapping from its own data model to the offered model. This task is considerably simplified by the empirical observation that, in most protocols, a happiness transition occurs at each higher level data boundary.

In some situations, it is not sufficient for a transport layer to know that the application layer is unhappy, and will eventually send a response: it must also have some indication of the amount of time that the response will take to arrive. A problem occurs when a transport layer that requires appeasement (in the example of section 3.2, A's transport layer) has set an optimization timer. If the foreign transport layer delays sending appeasement because it knows that the application layer is unhappy, the optimization timer might expire, causing a retransmission of the initial transport-level message. If this situation occurs repeatedly, it is possible for the transport protocol to experience a modified version of the asynchrony problem which exhibits exponential behavior with a larger coefficient! In the extreme case, the amounts of time in which the application and transport layers expect appeasement could vary so widely as to cause the transport layer's death timer to expire while its cooperating entity is holding its appeasement pending a message from the application layer. An important component of the usage model is an indication not only of when the higher-level protocol becomes unhappy and is appeased, but also of the time frame in which the appeasement takes place. The inclusion of this information in the usage model leads to a solution of the timer problem of section 3.1.3.

When it is intended that a lower level protocol is to be used to support a class of protocols all of which exhibit a common characteristic that would not normally be visible to the lower level protocol, it is useful to include this characteristic in the lower level protocol's usage model. It is often the case that when the lower level protocol knows not only the service that it is providing, but also the way in which this service is used, it is able to provide the service more efficiently. The justification of this statement involves the relative happiness of the lower level protocols and the higher level protocols that use them. There are reasons for a lower level protocol to become unhappy without regard to the happiness of any higher level protocol (for example, when a closed window is encountered). Conversely, any time a higher level protocol is unhappy, the protocol layers beneath it must also become unhappy, to avoid the asynchrony problem. However, a lower level protocol should avoid becoming unhappy when it has neither higher nor lower level reason to do so.

An example is in order. If a lower level protocol is providing a reliable delivery service, it will typically become unhappy whenever any data is transmitted. There are two reasons for it to behave in this way:

1. to modify the characteristics of the network (using optimization timers, as does TFTP's transport layer) in order to provide a more sophisticated service to a higher level protocol that uses it.

2. to ensure that a higher level protocol that uses it sees the reliable delivery of its data *regardless of when it decides to become unhappy*.

When the happiness or unhappiness of the higher level protocol is known to the lower level protocol, the second of these reasons does not apply. In many situations, the characteristics of the network do not need to be modified in a way which affects the happiness state of the lower level protocol (we will present a general-purpose reliable delivery service which exhibits this characteristic in the next chapter); in these situations, the lower level protocol has no reason to become unhappy unless the higher level protocol is also unhappy. If the lower level protocol does not become unhappy until the higher level protocol does, it will tend to be able to piggyback its appeasement on top of the higher level appeasement. The number of packets sent is thus kept to a minimum.

To summarize, the usage model of a protocol specification contains the following components:

- An *offered data model*, a model of the higher level divisions in data, in which a happiness transition occurs at each data division. This serves to transmit information about happiness transitions.

- A description of the higher level timing requirements that are assumed to exist, so that the protocol implementation can judge whether the higher level timing characteristics are compatible with its own or not.

- A description of any important characteristics common to all higher level services which are derived from the service being provided, so that the protocol implementation can avoid becoming unhappy when there is no reason for it to do so.

6.3 Solving the Timer Problem

Recalling the discussion of section 3.1, we note that protocol implementations are prone to a particularly insidious sort of deadlock, in which a network entity waits forever for a message from a cooperating remote entity that has failed. To avoid this deadlock every protocol entity is required to have a death timer. In a layered architecture, each layer needs to implement its own death timer for it to be able to avoid the possibility of waiting forever when another layered protocol entity in the local host fails. We noted that all but one of the death timers in a given implementation of a layered protocol architecture are unnecessary, since it is usually the case that no useful work can be accomplished after the first death timer expires. That all layers still need to implement their own death timers is an artifact of the perceived need of each layered protocol entity to avoid trusting in the death timers of another layer.

We also note that to improve the performance characteristics of the network, layers set timers called optimization timers. Many of the optimization timers which exist at any time in an implementation of a layered protocol architecture are also unnecessary, since other optimization timers at higher levels, which expire before they do, duplicate the effect that they were to have.

Since most of the death timers and many of the optimization timers in an implementation of a layered protocol architecture are unnecessary, and since their existence complicates the implementation of each layer, it is desirable to find a way to detect and eliminate the timers that are not necessary.

As a side effect of providing timing information in the usage model of the protocol specification of a lower level protocol, the timing requirements of the higher level protocol are made visible at the lower level, so it is possible for the lower level protocol to provide a *timer service* to the protocols that use it. The lower level protocol provides a timer service to higher level protocols by setting timers for them and allowing their actions to affect the timers in predictable ways. The timers are manipulated by the higher level protocol, but are managed by the lower level protocol. In this way, the lower level protocol is given access to both its own timers and to those of the protocol above it. It can thus determine which timers will ever have a useful effect and which will not, and need not bother to actually implement those timers that will never matter. If each layer in a protocol architecture offers timing services to the layer above it, all timers are ultimately concentrated in the network layer, and only those timers that will have a useful effect will actually be allowed to consume system resources. This solves the timer problem.

It is not hard to see that the timer service helps protocols to deal with the overabundance of optimization timers. In some sense, a reliable stream protocol which performs retransmissions to recover from lost packets provides just this sort of service to protocols that use it, except that it not only provides the optimization timer, but also interprets its use in a rigid way.

When a timer service is used to provide death timers, the situation is more complicated. The problem with such a timer service is that it introduces a failure dependency between the layers of the protocol architecture, as described in section 2.2, which makes it possible for the deadlock of section 3.1 to occur. If a lower layer is entrusted with a higher layer's death timer and fails, the higher layer will soon hang waiting for the lower layer. Since its death timer — which was inside of the lower layer — no longer exists, it will hang forever.

In many applications it is acceptable to presume that the lower layers of protocol are more-trusted than the higher layers, and that it is thus appropriate for the higher layers to depend on them. The reason for this is that there is usually only a single lower layer which is appropriate for many higher layers. For example, on a computer system with but a single network interface, the link-level protocol for that interface is the most trusted layer since a failure in it would deprive the entire system of service. In this situation, it is reasonable to assume that the higher layers will fail if there is a failure in the lower layers on which they depend. If a more-trusted lower layer protocol fails, the operating system may avoid the deadlock described above by *assuming* that all the layers above the layer that failed will soon deadlock, and can cause them to be aborted immediately so that it can reclaim their resources. It is safe in many applications for such a lower level protocol to provide a timer service which includes death timers.

6.4 A Methodology for Designing Protocols

The addition of a usage model to the protocol specification is also useful in that it begins, in a rudimentary way, to develop a methodology for designing and implementing protocols. There are two components to the methodology: data abstraction and happiness.

Data abstraction is a useful technique because the invariants that apply to data structures and are maintained by a data abstraction play a major part in the design and implementation of software. 'Happiness' is useful to a protocol implementation in a similar way; whereas data abstraction is the concept that describes the interactions of programs with data, happiness is the concept that describes the interactions of programs with other programs. A large part of any protocol implementation is concerned with the rules which determine which data may be sent at what times. These rules are really a set of "happiness rules" which indicate how the entity becomes happy and unhappy and how it is appeased. Most of the code which implements a protocol is there to keep to the implementation in accordance with the "happiness rules" of the protocol.

The duality between protocol happiness and data abstraction goes further. The code that maintains the happiness state of a protocol entity is tightly coupled in similar fashion to the code that implements a data abstraction. The code which maintains happiness has a somewhat more difficult task than is usually associated with data abstractions, since the code which appeases an entity runs on a different network site than the code which is appeased. An inevitable tradeoff occurs between the need to trust a remote partner sufficiently to accomplish what is intended, and the desire to prevent an error in the partner (or on the way to him) from causing too much harm.

6.5 Summary

In this chapter we have developed a technique called *Soft Layering*, whereby a protocol specification is extended to include a model of the protocol that uses it, known as the *usage model*.

To be most effective a usage model should include:

1. A description of any higher level service which is derived from the service being provided.

2. An *offered data model*, a model of the higher level divisions in data, in which a happiness transition occurs at each data division.

3. A description of the higher level timing requirements that are assumed to exist.

In the next chapter we try to make the discussion of this chapter more concrete by examining a particular pair of soft layered protocols. The first, called *Angel*, is a protocol that provides a reliable, sequenced, datagram delivery service. The second, which resides on top of Angel, is called *SFAP*. SFAP, which stands for *"Simple File Access Protocol,"* provides a means of getting at a file which is stored on a remote file system without any predefined order of access.

Chapter Seven

An Example of Soft Layering

The preceding chapters have developed all of the fundamental ideas of this thesis. In this chapter, we present an example which illustrates some of the ideas outlined earlier. The example is in the form of a pair of soft layered protocols. The particular protocols presented are not perfect. They are intended only as a tool to help the reader understand the abstract presentations of earlier chapters from a more concrete point of view.

The two protocols that are described in this chapter are *Angel*, a reliable datagram transport-level protocol, and *SFAP* (the **S**imple **F**ile **A**ccess **P**rotocol), an application-level protocol for accessing files stored on remote file systems. To simplify the machinery of implementing and testing the two protocols, Angel has been designed to work on top of the User Datagram Protocol [21].[25] Angel's lower interface is thus *not* soft layered. For this reason, our discussion will center on the interface between Angel and SFAP.

The specifications of the two protocols are included as appendices to this thesis. The zealous reader may desire to read the specifications at this point and then return to the rest of this chapter, although it is not necessary to do so in order to understand what follows.

7.1 The Angel Protocol

Angel is a transport protocol which presents higher level protocols with a view of the network as a reliable stream of datagrams. Other protocols exist which provide a similar view of the network; among these are the Arpanet Host-to-Host protocol [20], the Chaos packet protocol [17], and the Xerox *NS* Sequenced Packet Protocol [19]. Angel differs from these in that it provides a facility for keeping track of the higher level acknowledgements to data, and makes use of its access to this higher-level information in its design: in other words, it is a soft layered protocol.

[25]The User Datagram Protocol (UDP) is in turn built upon the Internet protocol [24]. UDP implements an unreliable, checksummed datagram service with one extra level of multiplexing. UDP's multiplexing is visible to higher level protocols as an address-space of multiplexed network ports, known as the *sockets*.

7.1.1 Angel's Usage Model

Angel's usage model describes a higher level protocol called the *letter-level* protocol. The letter-level protocol is a fictitious protocol which has the characteristics that Angel expects are common to all the higher level protocols that use it.

Two *letter-level entities* communicate by building a letter-level connection between themselves. This is done by establishing an Angel connection between their corresponding Angel level entities, and then sending tagged and ordered blocks of data, known as *letters*, across the connection in either direction.

A letter-level connection is a communications channel between the letter-level entity which is requesting a service — the client — and the letter-level entity which provides the service — the server. The identity of the host and service must be provided by the letter-level in the form of a host address and (numeric) service identifier; Angel does not specify how these are derived. A common technique is to cause the service identifier of the server to be specified in a letter-level protocol specification.

The letter-level client is distinguished only in that it opens a connection by actively sending a letter, while the letter-level server listens to the network for an indefinite amount of time and "accepts" a connection when it receives a letter from some client which requests its particular service. Once the letter-level connection is opened, the distinction between the letter-level server and client disappears.

Angel presumes that letter-level entities are using it to perform reliable communications. This assumption makes Angel different from protocols such as TCP [33]. For example, Angel does not provide a handshaking scheme to help letter-level entities coordinate the closing of a connection, since, as mentioned in section 6.1, reliable communications has no meaning once one of the two letter-level entities has gone away. An Angel connection is terminated when either letter-level entity indicates that the letter-level connection has terminated; if the letter-level entities must cooperate to properly terminate the connection, the proper place for this cooperation to take place is at the letter-level.

The assumption of reliable communications also affects Angel's view of the way in which letter-level entities communicate. If letter-level communications are to be reliable, then the letter-level entities must exchange letter-level acknowledgements [28]. Thus, every letter can be assumed to be ultimately acknowledged by some later letter. Angel provides the bookkeeping needed by the letter-level to keep track of the responses to letters.

Angel does not presume that letter-level responses are sent in any particular order. It is acceptable for responses to arrive in a different order from the requests to which they respond. For example, a letter-level "remote-job" protocol might wish to acknowledge the completion of jobs in the order in which they complete, regardless of the order in which they were begun; this sort of behavior is acceptable to the Angel protocol.

Angel's bookkeeping is designed around the model of a one-letter request and a one-letter response. However, the letter-level is free to assign whatever semantics it wishes to Angel's bookkeeping scheme. For example, since the order in which letters are sent is preserved by Angel, it is possible for the letter-level to interpret the response to the *last* of ten letters as a response to all ten.

The timing characteristics of the letter-level protocol are expressed in terms of letter-responses. When a letter-level entity sends a letter which requires a response, it specifies the amount of time it is willing to wait for the response. If the response does not arrive within this interval, Angel notifies the letter-level and aborts the connection.[26] Angel also provides the letter-level with operations to modify or cancel any particular response timer.

The reader may find an brief example of how Angel's letter-response facility is used to be helpful. Consider a file transfer protocol built on top of Angel. During the course of a connection, the file-transfer client wishes to read a file. To do so, it sends the file-transfer server a message containing the name of the file. The message is presented to Angel as a letter with an indication that a response is expected within some reasonable amount of time (say ten seconds).

The most meaningful response to a request to read a file is the file itself; however, Angel is willing only to keep track of a one-letter response to a one-letter request. The file transfer protocol solves this problem by using the convention that the response to a file transfer request is the *last* block of the file, where each block is small enough to fit into a single letter.

Angel has already been instructed to make sure that the response to the file transfer request arrives within ten seconds. It is clearly impossible for the file-transfer client to predict in advance how long any particular file transfer will take. Even if it knows how long the file is, the difference between the worst case time for a transfer (where each packet takes just as long as the client is willing to wait) and the best case time (where the entire file arrives within the time allotted for *any* progress to be made) would be so great for a long file that it would not be reasonable to set timers for that length of time.

[26] An alternate design for Angel would be simply to inform the letter-level that the response did not arrive, and allow it to make the decision of whether or not to abort the connection.

Instead, the file-transfer client interprets incoming letters — each containing a single block of the file — as a partial acknowledgement of the entire file, and informs Angel that it is content with the progress of the transfer by extending the letter-response timer that Angel is maintaining for it. The job of keeping track of acknowledgements is shared between Angel and the file transfer protocol.

7.1.2 The Details of Angel

Now that we have presented Angel's usage model, we may present some of the details of the way in which Angel realizes the services that are described in it. In what follows, we refer to the messages that are transmitted between Angel-level entities as *datagrams*, in keeping with the terminology of the User Datagram Protocol. The discussion of this section ignores the parts of Angel which would be the same in any reliable datagram service. These details are available to the reader in appendix A.

To keep track of letters and their responses, Angel assigns identifiers to letters from an ever-increasing (modulus 2^{16}) space of integers. These are known as the letter-identifiers. When a letter is sent or received, Angel informs the letter-level of the letter-identifier that was assigned to it. The letter identifier is used by the letter-level when it sends a response to a letter, to indicate to Angel to which letter a response is to apply. Letter-identifiers also give Angel a way of automatically resetting letter response-timers when the response to the letter has arrived.

There are several other items that are conceptually part of a letter, but which are manipulated by Angel on behalf of the letter-level. These include a letter-type field, a flag indicating whether the letter is a response, and (when it is) the identifier of the letter to which it is a response. With the exception of the letter-identifier, an implementation might choose to let the letter-level fill in any of these fields directly, by exposing the letter header to the letter-level. The letter identifier must be filled in by Angel in order to ensure that letter-identifiers are allocated from a non-decreasing sequence. That the letter header does not fit solely into either Angel's or the letter-level's protocol header is a consequence of soft layering.

A letter is constrained to fit into a single datagram, which in turn fits into a single packet. It would have been possible to design Angel in such a way as to obscure packet boundaries, as is done in many stream protocols. It was decided that a reliable datagram protocol was not the place in the protocol hierarchy at which to do this, for reasons which do not bear on the discussion of this chapter.

Angel performs its protocol multiplexing using the User Datagram Protocol's *socket* facility [21], combined with an initial handshake. When an Angel client wishes to connect to a service, a local User Datagram socket number is chosen in such a way as to minimize the possibility of a conflict either with another Angel connection (although this is usually checked) or some service identifier which is not implemented on the local machine.[27] The client's initial letter is sent from this randomly chosen socket to the User Datagram socket named by the service identifier at the foreign machine. Angel's service identifiers are mapped one-to-one onto User Datagram socket numbers.

At the Angel server's side, the letter-level has instructed Angel to listen on the service's User Datagram port. When an incoming datagram arrives, Angel extracts the letter which is contained within it and gives the letter-level a chance to accept or reject the connection; a rejected connection is returned with a special *exception* packet. If the letter-level decides to accept the connection, the Angel level chooses a second randomly chosen User Datagram socket number and returns to the letter-level a connection between the Angel-level client's and Angel-level server's randomly chosen socket numbers. Angel continues to listen for initial letters on the service socket.

During the course of the connection Angel maintains letter-response timers on behalf of the letter-level. The happiness or unhappiness of the letter-level is thus directly accessible to Angel (as a consequence of the way in which unhappiness is defined, on page 50). This information is used by Angel to avoid the asynchrony problem. As much as is possible, Angel forces itself to be happy whenever the letter-level is happy:[28] it never retransmits any datagrams when the letter-level is happy, even if it does not know if they were or were not received by the foreign angel-level.

When the letter-level expresses its unhappiness to the Angel-level by sending a letter which requires a response, Angel sets an 'unhappy' flag in the datagram it uses to transmit the letter, and sets an optimization timer. If the optimization timer expires, the Angel level retransmits all the datagrams for which it has not received an Angel-level acknowledgement. The letter-level's response timer is used to place a limit on the number of times the optimization timer may expire — this is an example of how it is possible to combine the death timers of the various layers of a soft-layered protocol architecture.

When the foreign Angel-level receives a datagram with the unhappiness bit set, it hands the

[27]In practise, either a random number or the low 16 bits of a millisecond clock have been observed to be appropriate for this application.

[28]The exception to this rule occurs as a result of Angel's windowing mechanism, described below.

letter in the datagram to the letter-level, and sets an optimization timer for a short period of time. If this timer expires before the letter-level sends any data, the Angel-level sends a datagram containing no letter-level data to appease its cooperating partner. This kind of optimization timer is known as a 'dally' timer.

The justification for the Angel-level's lax approach to implementing reliable delivery is worth noting. It is presumed that the underlying network is relatively reliable. Packets may be lost, but not most of the time. Thus, when Angel sends a datagram which contains a letter, it may be reasonably sure that the datagram will arrive safely at its destination. To do its job, it is not sufficient for Angel to be only reasonably sure that the letter was transferred; Angel must receive an acknowledgement for the letter. However, if the letter-level is happy, i.e., if it is not expecting any letters in response to those it has sent, Angel may presume that the letter-level does not yet care whether Angel has performed its job or not. Until the letter-level becomes unhappy, i.e., until it sends a letter which requires a response, Angel need not bother to retransmit any of the datagrams it has sent.

Once the letter level is known to be unhappy, the Angel-level sets optimization timers in a manner which is similar to other reliable delivery protocols. The difference is that an Angel-level sets optimization timers *only* when the letter-level is known to be unhappy — when there is a reasonable chance that the optimization timers will expire only when a packet is actually lost during transmission.

Angel, as any lower level protocol, can be foiled by a higher level protocol which exhibits characteristics which are completely different from those which are expected; in Angel, those characteristics are more clearly defined. For example, in section 3.4 a technique for setting optimization timers was examined which was very similar to the Angel-level's 'dally' timers. There we noted that the problem with the technique was that there was no reason to believe that the optimization timer would help rather than hinder the performance of the protocol. In the case of Angel, a dally timer is set only when there is reason to believe that it will help. Angel was designed with simplicity in mind; it is possible to envision a modification to Angel in which appropriate values for optimization timers are passed from one Angel-level entity to its cooperating partner. The important characteristic of Angel, as a soft layered protocol, is that its structure would naturally support such a modification.

To summarize the above, Angel allows the letter-level to coerce it into a happy state until such a time as the letter-level is itself unhappy. Since Angel only becomes unhappy when the letter-level does, the techniques that Angel uses to allow a letter-level response to be piggybacked on top of Angel's response are guaranteed a high chance of success for all higher-level protocols that fit into Angel's usage model.

When an Angel-level entity holds back on sending an appeasement message until the higher level appeasement is sent, it is placing trust in a layer which is higher than itself; should the letter-level *never* send a letter which requires a response, the Angel-level will not do its job correctly. However, this deficiency of the Angel layer only affects the particular Angel connection that is providing a service to the letter-level entity that failed to send the expected letter-level response; of those resources that are shared among all Angel connections, only the most inexpensive ones (for example, UDP socket numbers) are endangered by the trust that Angel places in the letter-level. This is the same sort of reasoning used in connection with Blast in section 4.5: Angel sidesteps the dangers of placing trust in a less-trusted layer that come from protocol multiplexing.

The Angel-level implements a flow control scheme based on a window of packets. Angel's flow control is analogous to that of the Xerox NS Sequenced Packet Protocol [19]. Each side of the Angel connection extends to its partner a window which enables the partner to send a fixed number of packets. As the partner sends packets, the window gradually closes, until no more packets may be sent. In Angel, a closed window causes both the entity that extended the window and the entity that closed it to become unhappy, so that the datagram that extended the window is retransmitted until the window is reopened. Because the cause of the unhappiness is independent of the higher level protocol, no daily timer is set. It should be noted that this significantly impairs Angel's ability to forestall the transmission of data between mismatched Angel-levels. This function is left to the letter-level.

An Angel connection is closed whenever either of the letter-level entities decides to close the letter-level connection. A single packet is unreliably sent to the other side of the connection as a courtesy, but it is presumed that the letter-level entities have previously negotiated the termination of the connection. If not, the next letter-level response timer will certainly expire, and the connection will be aborted anyway.

7.2 The Simple File Access Protocol

SFAP, the Simple File Access Protocol, is a protocol which provides access to files stored on the file systems of remote computers. File access is distinguished from the more common file transfer in that a file may be accessed in random order, and less than the entire file may be accessed.

The sort of access to a file provided by SFAP is not unlike that provided by the Leaf protocols of the Woodstock file server [31]. The primary difference between them is that SFAP is built on top of a file system, while the Leaf protocols are integrated into the file system structure. For example, the structure of a file name handed to SFAP is subject to the particular file name matching conventions of the file system on the SFAP server's machine.

The reader will note that there are two protocol specifications for SFAP appended to this thesis. The first, in appendix B, to which we shall refer as SFAP I, was designed early in the course of the author's research, and is built directly on the User Datagram Protocol. It is not soft-layered. The second, in appendix C, which is of interest to us here, is a soft-layered protocol with almost exactly the same capabilities as SFAP I. It is aptly named SFAP II. Both specifications are presented in order to illustrate the way in which soft-layering simplifies the protocol specification.

In the rest of this chapter, "SFAP" will refer to SFAP II, unless otherwise stated.

7.2.1 SFAP's Usage Model

Any protocol which deals with files must perform at least a minimal mapping between the conventions of different file systems. When a protocol implements remote file access, this mapping becomes more involved, since both the structure of files and the operations on them must be mapped across file system boundaries.

SFAP views a file as an array of eight-bit bytes (sometimes called *octets*). The bytes are arrayed into *blocks* containing 512 bytes each, except for the last block, which is guaranteed to contain less than 512 bytes. Conceptually associated with the file, but actually created and maintained by the server, is an additional block, known as the *descriptor block*, which contains information about the file, such as its length, the time it was last written, and so on.

There are five operations that may be performed on files: Open, Describe (read the descriptor block), Read (a single block), Write (a single block), and Close. All operations are ordered; if two write requests are sent in sequence, it is guaranteed that the first write will have been performed successfully by the time the second is attempted. It is possible to pipeline requests: subsequent requests may be enqueued without waiting for outstanding requests to be acknowledged. It is not necessary to receive confirmation of a write request, since the success of any subsequent operations implies this confirmation.

7.2.2 Realizing the Protocol

In keeping with the methodology described in chapter six, they way in which SFAP realizes the operations of its usage model is best described in terms of a mapping between SFAP's usage model and Angel's.

Each SFAP-level message is realized as a single *letter*. The letter type field is used to determine the operation in question. Angel's letter-response facility is used to indicate when a response is needed for a particular series of requests. This is primarily interpreted as a request to flush the pipeline when the end of a burst of requests arrives. For write requests, it has the additional effect of causing an explicit confirmation of the write to be sent.

Because of the pipelined nature of the protocol, errors require some extra effort. When the server encounters an error, it sends an error message (which is a single letter) to the client, with response required. It then ignores all subsequent requests from the client until the response to the error message arrives. The client uses the arrival of an error message as an indication that all outstanding letter timers must be canceled.

Connections are closed by a Close request, which may only originate in the client. A close which originates in the server is interpreted as an error.

7.3 Angel, SFAP, and Soft Layering

The discussion of the Angel and SFAP has concentrated on pointing out those areas in which responsibilities could be shared between the protocols in a way which improves both protocols. It is important to note that all of the "tricks" which are used in Angel and SFAP are entirely contained within the design of the protocol. Concepts which are of vital importance to the feasibility of the protocol are not left to either auxiliary documents or to the whims of individual implementors.

It is useful to note the importance of the usage model in all this. Angel is able to offer a timer service to SFAP because it has built into it an understanding of SFAP's behavior. Similarly, the design of SFAP is greatly simplified by the constraint that it fit into Angel's model.

What if SFAP did not fit into Angel's usage model? If SFAP *never* sent a message which required a response, or if it required a response but neglected to tell Angel about it, Angel would assume that SFAP had no need of a response, and might never cause any letters to be delivered successfully! In fact, it is possible to use Angel without fitting into its usage model, if you use it the "right way." For example, if the letter-level *always* tells Angel that it expects a response, and immediately sends responses (which consist of letters of zero length) when it sees any letter, communications will take place correctly. In this case, as a perusal of the Angel specification makes evident, Angel degenerates into a conventionally layered protocol; in fact, it looks very much like a design for a poor implementation of TCP.

If the position is taken, as it is here, that a protocol must be efficient to be usable, then it is certainly the case that Angel is not usable for letter-level protocols which do not fit into its usage model. This makes it appear as if soft layering comes only at the cost of a great loss of modularity.

In practise, whether a usage model is or is not presumed in the protocol specification, it is presumed by the eventual implementation. What is worse, different implementations may well end up with radically different "implied usage models," which may interact in complex ways. An example of this sort of behavior has been noticed between different implementations of the reliable stream protocol, TCP.

TCP is used as a transport layer protocol for both remote login and file transfer applications. The windowing strategies of these applications are very different. In a remote login protocol, it is desirable to open the window whenever possible, since the client receives its data one byte at a time from a human user typing at a console; any opening in the window can be put to good use. In a file transfer application, both the client and server would prefer to receive data in large chunks, since this is more appropriate to the disk hardware. If the client is trying to send large chunks of data, and the (TCP) server is presuming — since it does not know for what it is being used — that the client receives its data byte by byte, a resulting confusion between the server and the client occurs, which has been termed the "Silly Window Syndrome" [10], whereby the client's window can fragment to the point where most packets contain no more than one or two bytes of data.

It is our feeling that the restriction on the generality of a protocol which is caused by the introduction of a usage model into its specification is no more than the formalization of restrictions that have always been present.

Chapter Eight

Conclusion

In this thesis we have examined the technique of protocol layering and found it lacking in a number of ways. We concentrated on one particular situation, that of a layered protocol architecture which implements reliable communications between cooperating application-level entities. In this context we saw that the maintenance of a layered structure in the protocol implementation could cause it to be so inefficient as to be unusable. After a good deal of analysis, we introduced an extension to protocol layering which provides a mechanism whereby the shortcomings of the layered structure can be fixed.

We began our discussion of layered protocols, in chapter two, with a development of the concept of protocol layering, and an outline of the advantages of the scheme. There we saw that, because there are typically many different network entities inside of a computer system, but only one (or perhaps two) hardware interfaces to the network, it is a requirement of the network software in the system that it allow all of these entities to share the network hardware. Further, because the task of implementing all the network applications in a host is a major one, there is a strong desire to modularize the structure of the network software in a computer system in such a way as to make it possible for the different network applications to share at least part of the code that implements them.

Protocol layering provides an elegant means of satisfying these two criteria in but a single mechanism. In a layered protocol architecture, each layer of the architecture provides to the layers above it a more sophisticated set of services than is provided by the network hardware. The nature of the 'refined' service that is provided by each layer is such that the service fits into the requirements of many of the network applications. Some applications will wish to make use of this refined service directly, while others will make use of it indirectly through the device of added layers of protocol (each of which refines the service further). Protocol multiplexing can also be provided in each layer of the protocol architecture to allow applications to use a layer's services directly without interfering with transport layers which wish to further refine the layer's services. As a side effect of the introduction of protocol multiplexing, the requirement of being able to share the network software among many network entities is also met.

Protocol layering is, then, a powerful technique for achieving modularity of the design and implementation of network software. It has been central to the discussion of the preceding chapters that the advantages presented by protocol layering are sufficient that it would not serve to abandon a layered structure entirely.

At the same time, as the discussion of the thesis progressed to chapter three, we began to examine some severe shortcomings of layered protocol implementations. Because of the uncertainty which is inherent in all network communications, we noted that a network entity must always maintain a *death timer*, which provides it with a mechanism that it can use to avoid waiting forever for a cooperating remote entity which has failed. In a layered protocol implementation, we saw that the same problem occurs at every level of protocol, so that it is necessary for the implementation of each layered entity to provide its own death timer. Since, at a given time, the cooperation of all the layers in use is required for any useful communications to take place, all but the shortest of these death timers are really unnecessary. All but one of the death timers in a layered structure are a parasitic side effect of the introduction of protocol layering. We also showed how the common practise of setting shorter, *optimization* timers for the purpose of providing different transmission characteristics than are available in the network layer is also prone to a similar sort of wasted resources. The characteristic of layered protocols that caused their implementations to set many timers, most of which are useless at any given instant, was entitled the "timer problem."

A more severe problem than the timer problem was also investigated in chapter three. Entitled the *asynchrony problem*, it stems from the need for network entities to reliably coordinate state information with their cooperating remote entities. In a layered context, this coordination of state occurs independently at each layer of protocol, because of the perceived inability of layered protocols to coordinate this activity without violating their modularity. The asynchrony problem results in an increase in the number of packets sent over the network to perform a given function at the highest level of protocol which is exponential in the number of layers in the protocol architecture (in the worst case). Since, as we demonstrated in section 3.3, this can be expected to result in an exponential *decrease* in the relative throughput seen at the application level protocol, it is a requirement of any protocol implementation that is to provide a useful service that it avoid the asynchrony problem.

In chapter three we also examined some of the techniques which are currently used by protocol implementations to avoid the asynchrony problem. These range from the total abandonment of a layered structure to a series of predictive 'tricks' which work well for some higher level protocols, some of the time. The inherent harm in these techniques is that all are independent of the protocol specification. Thus, to implement a usable version of a protocol, it ceases to be sufficient to simply implement its specification as written. If a protocol specification does not say how to implement the protocol, then its value is considerably diminished. Furthermore, the 'tricks' needed to implement a protocol efficiently are generally not codified, and are often specific to particular protocols or operating systems. This makes clear the attractiveness of any solution to the asynchrony problem that works within the context of protocol layering and is integrated into the protocol specification. In chapter six we described just such a solution, called *soft layering*.

In a soft layered protocol, the protocol specification is augmented to include a 'usage model' for the protocol: a model of the way in which the protocol expects higher level protocols to use it. Higher level protocols which conform to the usage model may expect to receive an efficient service from the protocol being specified. Other higher level protocols will still be able to make use of the service defined in the protocol specification, but the service they are provided will not be efficient. Soft layering provides a mechanism whereby the meaning of protocol efficiency — which is *always* a part of the protocol implementation — may be formalized in the protocol's specification. This ensures that all implementations of the protocol provide the same service from the point of view of both correctness and efficiency.

In chapter seven we presented an example of a layered protocol architecture, consisting of two layered protocols. We showed how soft layering not only prevents the asynchrony problem, but also makes the design and implementation of higher-level protocols more straightforward.

8.1 Suggestions for Further Research

In chapters one and two, we discussed in the abstract a number of issues of concern to protocol design and implementation: the unity of correctness and efficiency concerns in protocol implementations; the modularization of software into layers; and the difficulties of allowing layers to place trust in one another. Now that the particulars of this thesis have been presented, it is instructive to return to each of these topics, and to describe the way in which the ideas presented here bear upon them. We will also indicate areas which are open to further research.

8.1.1 The Correctness and Efficiency of Software

We have spent a great deal of time discussing protocol efficiency in this thesis. We began with the general idea that network software is an important component of a class of software for which the concerns of correctness can not be divorced entirely from those of efficiency. We gave a graphic illustration of why this was so in section 3.2, where we saw that the lack of attention to efficiency in the production of correct layered protocol implementations caused them to degrade exponentially in efficiency with each added layer of protocol.

Our approach to the problem has been to develop soft layering, a mechanism which allows efficiency considerations to be included in the protocol specification by the addition of a *usage model* of the protocol being specified. The usage model is not a metric by which the efficiency of a protocol implementation can be measured. It is rather a description of the interactions between the protocol and the layers above it, which restricts protocol implementations on different machines sufficiently that they can interact efficiently with each other.

Even in soft layered protocols, there are many efficiency concerns which remain local to the computer system in question. This is as it should be, since many of the efficiency concerns that affect protocols do, in fact, differ when the protocol is implemented on different machine architectures, operating systems, and/or computer administrations. It is the intent of soft layering to provide a universal notion of efficiency only insofar as protocol interactions require that such a notion exist.

8.1.2 Layering and Modularity

In section 2.3 we indicated that it is commonplace in layered structures to consider some layers 'more-trusted' and others 'less-trusted.' In layered protocol architectures, it is most common to consider the lower layers — those closest to the hardware — to be the most trusted layers. In many computer systems this follows naturally from the fact that there is but one hardware interface to the network, so that if the lowest layer fails, all other layers can rightfully be expected to fail as well.

It is a violation of conventional layered software structures for a layer to place trust in a layer that is less trusted than itself. This is because of the greater potential that a failure in a less-trusted layer will lead to globally disastrous results if the failure is allowed to propagate to more-trusted layers.

In chapters four and five, we saw how it is specifically at the application layers — traditionally the least-trusted layers of a protocol architecture — that the information which is needed to implement protocols efficiently exists. Because of the layered structure of the protocol architecture, it is not possible for conventionally-layered protocols to make use of this information.

One way of looking at this situation is that layering imposes a particular sort of modularity on protocols. In some situations (for example, protocol multiplexing) this sort of modularity is appropriate to the task at hand, and works well. In this thesis, we have demonstrated that the task of efficiently providing reliable communications at the applications level does *not* fit into the modularity exhibited by conventional layering.

In soft layering, we present a mechanism for allowing a layer to place trust in a less-trusted layer without endangering its or the system's integrity. What we have done is to provide a mechanism for a different modular structure to coexist with that of conventional layering.

We have been concerned only with the issue of implementing reliable communications among network entities. Among the issues we have ignored are some which fit equally poorly into conventional protocol layering, and are left to further research. These include the issue of congestion control within a host's network software.

Network congestion is a well known problem, whereby the uneven loading of the network by host computers causes packets to be lost even when there are sufficient resources in the network as a whole to handle the traffic. An analogous situation can occur in layered protocols. Consider the case of three application protocols which use the same transport layer protocol to perform their function. Each application layer wishes to send data at the same time. When the transport layer protocol was implemented, constraints were placed on the amount of buffering that could be accommodated at any one time (for example, the transport layer might reside in a memory-tight operating system kernel). The transport layer has enough buffering to handle any one application layer which exercises the network resources to the fullest. When it is exercised by three application layers simultaneously, it cannot afford to buffer all the data to be sent. In many layered protocols (with notable exceptions, such as X.25 [35]), there is no mechanism whereby the transport layer can inhibit the higher layers from handing it data faster than it can send it, even for short periods of time. The usual approach in the implementation of transport layers is to discard higher-level data in such a way that the higher level protocol's own error-recovery mechanisms can recover and resend it. Unfortunately, the transport layer has no way of knowing which higher-level messages are the most important; it may choose to discard a higher-level message which would have caused the higher level to stop sending packets, and instead choose to keep ten identical retransmissions of some less important message. The poor choices that are made by the transport layer concerning which messages to discard and which to keep result in behavior at the higher-level protocols that is less efficient than would be the case if they could simply have refrained from using up the transport-level resources.

There are a number of ways in which existing computer systems prevent this sort of problem from occurring. Some are able to implement vast amounts of buffering at the network level, with the result that it is possible for hundreds of packets to be sitting on queues waiting to be transmitted, most of which are retransmissions of the same packet. Other hosts make use of the process scheduler to suspend processes which would cause congestion at lower level protocols. This invariably results in congestion elsewhere in the network, because packets may still be sent *to* the offending process. Often the congestion is merely shifted into the operating system's network interface, which becomes congested with packets for the suspended process. As above, it is likely that many of these packets will be retransmissions of earlier ones.

An interesting scheme for avoiding in-host congestion is one recently proposed by D. Clark at MIT, called the *Upcall* scheme. In it, when an application-layer entity wishes to send data, it makes a reservation with the lower layer to do so. The reservation ultimately reaches the network layer (which presumably has access to the network's own congestion control facilities). When conditions at all layers in the protocol are such that it is safe for the application layer to send its message, an *upcall* proceeds from the network layer, through all the layers of protocol, which informs the application layer that it is time for it to send data. No network activity is allowed until the application layer acts on the upcall.

The disadvantage of this scheme is that it has as its basis a violation of the principles of trusting that are normally presumed in a layered structure. It is possible that this technique, when coupled with soft layering, will be able to make use of the same technique described in section 4.5 to avoid the dangers of violating the normal 'trusting rule' of layered protocols.

8.2 Meta-Conclusion

This thesis has developed a number of ideas about network protocols to a level which has not, to the author's knowledge, previously been done. It is useful to summarize what we consider to be the most important of these.

- Problems with Layered Protocols: We presented an abstract description of two problems that are exhibited by layered protocols, which stem from their layered structure. They are:

    - The Timer Problem: that timers proliferate at all layers of a layered protocol architecture, even though most of the timers that exist at any one time are redundant.

    - The Asynchrony Problem: that, in a protocol architecture where every layer must coordinate state information with its cooperating entity, the number of packets that traverse the network to enable the highest-layered entities to exchange one message each, is 2^N, where N is the number of layers in the protocol architecture.

- Happiness: We have presented a new terminology which is particularly appropriate to discussion of the state of a network entity. It is our belief that the concept of 'happiness' is generally useful in the process of designing and implementing protocols, in a manner analogous to the way in which data abstraction is useful in the process of designing and implementing other kinds of software.

- Soft Layering: In soft layering, we present a mechanism for introducing into a layered protocol architecture other forms of modularity which are more appropriate to some of the tasks at hand. Soft layering is implemented entirely within the context of the protocol specification, so it allows the efficiency considerations of interacting network software to be coordinated without losing the benefit of a centralized and complete specification.

Appendix A

The Angel Protocol

A.1 Introduction

Angel is a protocol which facilitates reliable communications between two cooperating, higher level entities. It is built on top of the User Datagram Protocol [21], and hence the Internet Protocol [24], but runs as Internet protocol number *101*[29], so that its packets are kept separate from other UDP packets on the network. Angel provides reliable, ordered delivery of typed blocks of data known as *messages* from one higher-level entity to another.

Reliable communications requires more than the delivery of the appropriate data in the right order. As described in [28], it is not possible for a lower level of protocol to implement the reliable transfer of data on behalf of a higher level. However, it is often possible for a lower level of protocol to use the information available at a higher level to make this transfer more efficient. This is the basis of the argument for a 'softer' division between protocol layers. Angel imposes a model of communications upon the level above it so that it has a better idea of how the higher level operates and can better serve it. While the model used is very general, it may cause angel to be unsuited for use with some higher level protocols. This is considered an acceptable restriction.

A.2 Angelic Communications

Angels are among the earliest messengers in history. Angel is named after these divine messengers because the unit of transfer it provides to higher level clients is known as a *message*. An angel message is a block of data which has a type and a (numeric) name. The type of a message, described by a tag appended to it, determines its interpretation. A protocol specification, for example, might be described in terms of which types of messages are defined in a protocol, and under what circumstances messages of a given type should or could legally be transmitted.

[29] this number is assigned for experimental purposes only.

When protocols are layered, a message from a higher level protocol is wrapped inside of one or several messages at a lower level. One way of speaking of this is as is done in the ISO reference architecture, by level number: message-1, message-2, and so on [11]. Since angel is only concerned with a limited number of levels, we will avoid such terminology and will instead refer to messages at different levels by different names.

Angel is concerned with two levels of messages. For its own purposes, angel passes messages back and forth between angel software on cooperating hosts. These messages are used to coordinate the transfer of higher level data. They are referred to in this document as *datagrams*.

A higher level protocol which makes use of angel sends messages. When these messages are handed to Angel, they are wrapped in a datagram "envelope" for transmission; they are referred to here has *letters* and the higher level client is itself referred to as the *letter-level*. Some of the information in the letter is understandable to Angel and is used to make the transfer of letters more efficient. Other parts of a letter are of interest only to the letter-level.

Angel maintains a *connection* between two letter-level entities. For the purposes of routing letters appropriately, a connection identifier, or *port* is maintained for each of the two communicating letter level entities. This identifier is for angel's internal use and should be of no interest to a letter-level protocol.

A.3 Higher level Interface

Angel assumes that letter level entities can be divided into *clients* and *servers*. A letter server provides a protocol *service*. It passively listens to the network, awaiting letters which request this service. Should the server choose to honor such a request, it handles it using a different angel connection, so that it may accept other requests. For each protocol service the letter-level protocol defines a *service identifier* which names it. Angel provides the following interface for server operations:[30]

```
    ProvideService( serviceID, letterHandlerPredicate )
            returns( AngelServerConnection )
```

ProvideService is called with the identifier for the service to be provided and a pointer to a "handler" predicate which decides whether or not to accept the connection. It returns a data structure which contains the angel-level state of the server. The server will then typically wait for a request to arrive on the network. This is accomplished by using the interface:

[30] All the examples presented here are synchronous procedure calls. It would, of course, be possible (and more normal) to use asynchronous communications between angel and a letter level protocol implementation, but the notation is here simplified to the synchronous case.

```
WaitForConnection( AngelServerConnection )
        returns( AngelConnection )
```

When a request to connect to a particular service arrives at the angel level, the handler predicate for that service is called with the letter's type and contents. If the handler returns false, the request is aborted by the angel level. If the handler returns true, a new connection is created, and becomes the returned value from the appropriate call to WaitForConnection. When possible, the letter requesting the service may be passed to this connection. If this is not possible, the letter-level must provide its own means of communicating this data (or of ensuring that no data need be communicated).

When a letter-level client wishes to access a remote service, it formats the letter-level request for service, determines the host and service identifiers, and calls:

```
AccessService( hostID, serviceID, RequestingLetter )
      returns( AngelConnection )
      except-in-error: signals( Refused, NoSuchService, NoResponse )
```

Assuming that the service is accepted, the call will return a data structure describing the angel connection state. There are possibilities for error termination. If the remote server's handler predicate returns false, the *Refused* exception is raised. If the angel level software on the specified host knows of no such service, *NoSuchService* is the result. Any other error will appear to the angel level as if its packets were not getting to the specified host; this would be the case, for example, if the angel protocol were not implemented on that host. In this case, *NoResponse* is signalled.

Once a connection has been established, the letter level may send and receive letters on it. A letter is sent using the SendLetter procedure:

```
SendLetter( AngelConnection, letterData, letterType,
                              [responseTimer], [responseTo] )
        returns( letterID )
        except-in-error signals( ConnectionDead )
```

The first three arguments are clear; they identify the connection to be used, the data to be sent and the type of the letter. The other arguments are not always included, and require more explanation.

Angel differs from other reliable delivery systems in that it does not even guarantee reliable delivery of data unless a response is expected (The reader will note that to deliver packet ***N*** reliably, all packets preceding ***N*** must also have been delivered reliably, so every packet need not expect a response). If a response is expected for the letter being sent, the *responseTimer* argument indicates how much time may elapse before the response is forthcoming. A letter which is a response to an earlier letter is sent by calling *SendLetter* with the last argument, *responseTo*, indicating the letterID of the letter to which a response is given.

If *responseTimer* is included in the call to **SendLetter**, angel will set a timer for the interval specified by *responseTimer*. If the timer goes off, angel assumes that a fatal error has occurred, aborts the connection, and notifies the connection's letter-level owner.

SendLetter returns once the letter has been sent out over the network with the identifier that was assigned to the letter. The letter identifier may be saved by the letter-level to distinguish the letter's eventual response.

Under some circumstances, SendLetter will need to block the calling process (if the window is closed, as explained later on). It is permissible for an Angel implementation to provide an interface where SendLetter returns in error in this case, so as to avoid blocking a process when there are other things for the letter level to do. Angel was designed to allow a blocking interface to SendLetter to be used in the majority of applications. If the connection has died or dies while SendLetter is blocking the calling process, the *ConnectionDead* exception is raised.

It is sometimes the case that partial results of a computation have been received at the letter-level, indicating that a requested transaction is underway, but has not yet completed. To prevent the timer associated with the letter that requested the transaction from destroying a healthy connection of this sort, angel allows the timer for any particular letter response to be reset:

```
ResetLetterTimer( AngelConnection, LetterID, newTimerValue )
```

An incoming letter is received by calling ReceiveLetter:

```
ReceiveLetter( AngelConnection )
     returns( letter, letterType, [responseTo] )
     except-in-error signals( ResponseDidn'tArrive, ConnectionDead )
```

In this interface, ReceiveLetter will block the calling process until a letter arrives (the comment about blocking, above, applies here as well. When a letter arrives, its data and type are returned. If the letter is a response to some previous letter, angel will clear the timer associated with that letter and return the LetterID of the letter as *responseTo*. Letters are handed to the letter-level by ReceiveLetter in the order they are sent using SendLetter.

If ReceiveLetter raises either of its two exceptions, the angel connection has died. If the reason is because an expected responding letter never arrived, the *ResponseDidn'tArrive* exception is raised. Otherwise the *ConnectionDead* exception is raised.

It is assumed that letter-level protocols are interested only in performing reliable communications between cooperating processes. If one process is no longer interested in the connection, there is no point in maintaining it. A connection is thus terminated when either of the two communicating letter-level processes calls TerminateConnection. A string and an error code may be sent explaining the reason for the termination:

```
    TerminateConnection( AngelConnection, ReasonCode, ReasonString )
```

A.4 Implementation

The previous sections introduced Angel and described the interface that it presents to the letter level. In this section, we discuss the implementation of that interface.

Angel-level software maintains a connection between two letter-level entities by passing messages called *datagrams* back and forth. Some datagrams will have letters wrapped in them for transmission to a letter level entity. Others will contain only Angel level data.

Datagrams, as letters, are messages. The length of a datagram is restricted in that a datagram must fit into a single packet; since letters are wrapped within datagrams, this restriction is passed on to the letter level. Some means should be provided for negotiating the maximum packet size between the angel and letter levels.

Every datagram has a type which determines how it will be processed by the receiving angel-level software and indicates whether it contains data to be passed to the letter level. Datagrams have names, known as *sequence numbers*. Sequence numbers are allocated sequentially from a sixteen bit counter. A datagram which contains a letter causes the counter to be incremented, but a datagram which contains only angel-level data (an *angel-only datagram*) does not. Thus, the probability of confusion between two letter-bearing datagrams which have the same sequence number is small. Sequence numbers are used to sequence letter-level data, and double as the letter identifiers which are handed to the letter level.

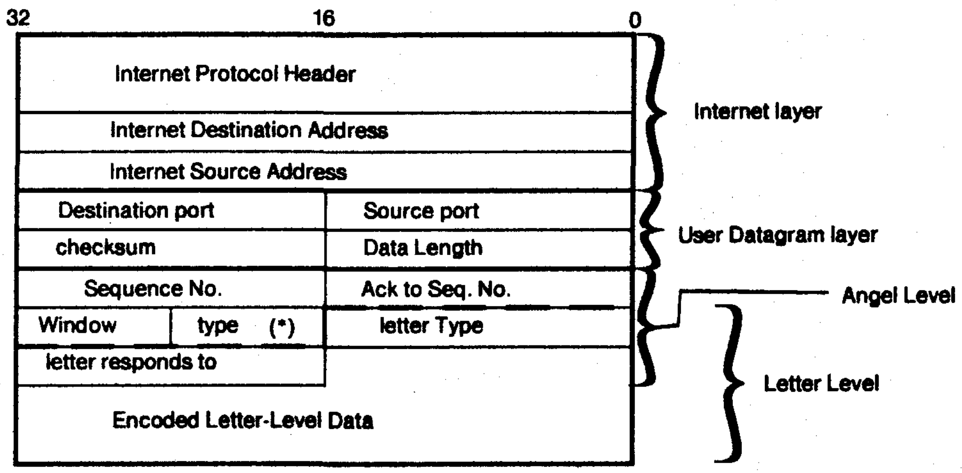
The flow of letter-level data is restricted using a window. Each side of the angel connection specifies to the other the number of datagrams that it can send until the window is closed. As datagrams are received, the window closes, and may hit zero. If it does, no new sequence numbers may be allocated, so it is only permissible to send angel-only datagrams. The handling of the windows is described in section A.4.3.

Each side of the connection indicates to the other side the sequence number of the highest numbered datagram that has been completely processed by angel level software. For the data in a datagram to be completely processed, all datagrams with a lower[31] sequence number must have been previously processed. If a datagram containing letter-level data is lost in transmission, the fact will be detected by its sender since it and all subsequent datagrams will not be acknowledged as having been fully processed. The semantics of recovering lost datagrams is discussed in section A.4.3.

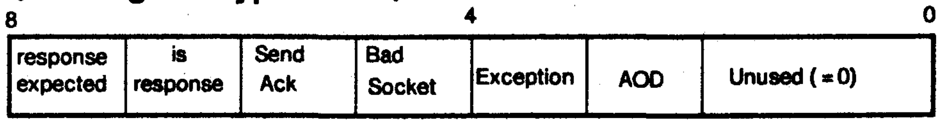
A.4.1 Datagram Format

The format of a datagram is given in figure A-1. It contains a prelude, consisting of the Internet and UDP protocol headers. It also has a header of its own, containing a number of fields. In a soft-layered protocol such as Angel, it is not completely clear where the angel header ends and the letter-level header begins. The diagram indicates with a dotted line where a datagram ends when there is no letter-level data.

Figure A-1: Format of an Angel Datagram



(* datagram type fields)



The fields in the datagram header, with the exception of the type field, are described below. The type field is described in section A.4.2.

Internet Protocol Header
    The header of the DoD standard Internet Protocol [24].

[31] Note that comparisons of sequence numbers should be done in modulus 2^{16} arithmetic

Internet Source/Destination Address
: These are the Internet addresses of the hosts on which the two cooperating entities are running. They are standard, 32 bit, global addresses. "Source" refers to the packet's sender.

Source/Destination Port
: The UDP port numbers of the source and destination entities on the hosts involved. During the course of a connection, these are temporary identifiers (TID's) that are unique on each host, and together identify the connection (see section A.4.3). "Source" refers to the packet's sender.

checksum
: The UDP [21] checksum of the packet's contents. This allows for detection of data that has been corrupted during transmission.

Data Length
: The UDP length of the packet, in octets. This value includes the length of the angel header.

Sequence Number
: [16 bits] The sequence number of this packet, in the connection of the packet's sender.

Acknowledge to Sequence Number (AtSN)
: [16 bits] The sequence number of the last packet received by the packet's sender. If this number is N, then all packets with sequence numbers less than N may also be assumed to have been received (note that "less than" should be interpreted in 2^{16} modulus arithmetic). When the datagram contains a letter, the value of this field is also the letter identifier of the letter.

Window [8 bits] This specifies the sequence number of the highest numbered datagram that this datagram's recipient may send to this datagram's sender. It is implemented, to limit the size of the protocol header, as a non-negative (zero or positive) offset from the "AtSN" field. Once the window is offered, it may not be withdrawn; the receiver of a packet should accept a window value only if the offered window is larger than any window previously received.

Type [8 bits] The type of the datagram. This is described in section A.4.2.

When the datagram does not contain any letter level data, it may end after the datagram type field (although space for the letter-level fields may be included, as warranted by ease of implementation). When letter-level data is encoded inside a datagram, the following fields must also be present.

Letter Type
: [16 bits] The type of the letter that is encoded inside of this datagram.

Response To
: [16 bits] This field is only valid when the IsResponse bit is set in the datagram type field (see A.4.2). It specifies the letter identifier of the letter to which the letter encoded in this datagram is a response.

Encoded Letter Level Data
: The body of the encoded letter or letter fragment. This data is not interpreted by Angel. The length of this field is the value of the User Datagram Protocol's *Data Length* field, less ten (10).

A.4.2 The Datagram Type Field

The type field of a datagram determines how the datagram will be processed by the foreign angel level. The type is itself divided into a number of sub-fields, each one bit long. The following is a list of the sub-fields, with a brief explanation of each:

ResponseExpected
: This bit indicates whether a response is expected for the letter contained within this datagram. It must be zero whenever the AOD bit is set.

IsResponse
: This bit determines whether the *Response To* field contains any data. It must be zero whenever the AOD bit is set.

SendAck When this bit is set, the datagram should be acknowledged by another datagram.

BadSocket
: This datagram is a **BadSocket** message, as will be explained in section A.4.5.

AOD Angel Only Datagram: The datagram is purely for the maintenance of the connection, and contains no letter. Such datagrams are used to extend windows when no letter-level data is available to be sent. When combined with the "SendAck" bit, an Angel-level echo service is provided.

Exception

The datagram is an exception datagram, which is provided as a courtesy when the foreign Angel level has aborted the connection. Exceptions are discussed in section A.4.5.

A.4.3 Maintaining a Connection — Happiness

Angel differs from other reliable delivery protocols in that it provides reliable delivery only as a means of implementing reliable communications at a higher level. To discuss this difference more clearly, we introduce some terminology.

We say that protocol software is *happy* when it has received an acknowledgment from its partner in communications indicating that all the data it has sent has been acted upon. Conversely, if some data has been sent, but not acknowledged, the protocol software is said to be unhappy or*upset*. Most protocols which provide a reliable delivery service become upset whenever they send any data, and become happy (if ever) whenever they have determined that all the data that has been sent has been delivered. In angel, happiness is assumed unless the letter-level explicitly indicates that the angel level should be upset by sending a letter for which a response is expected. In addition, an angel level becomes upset whenever the window offered to it closes completely, such that it may not send any more letter-level data. When one side of an angel connection is upset, this condition is passed to the other side of the connection as packets are sent so that the foreign angel-level may help to "appease" it.

An angel-level which has become upset sets an *activity timer*. If this timer goes off, any packets which have been transmitted but not acknowledged are retransmitted. This process may be repeated a number of times, until either a letter timer goes off (which causes the connection to be aborted) or the angel-level has determined that the connection is no longer active and aborts the connection.

When an incoming datagram contains a letter for which a response is expected (i.e., its ResponseExpected bit is set), it can be presumed that the foreign angel-level is upset. Usually, it will not be necessary to send an angel only datagram to appease it, since the local letter-level can reasonably be expected to send the responding letter after only a short interval. To be sure, the angel level sets a *dally* timer for a short period of time. If this timer goes off before any letter-level data is sent, an angel-only datagram is sent, which acknowledges all the datagrams that have been received to date. If any letter level data is sent before the dally timer expires, the timer is cleared, since the acknowledgment for the delivery of the datagram in question will be "piggybacked" on the letter being sent.

When an incoming datagram indicates that the foreign angel-level has a zero-length window, an angel-only datagram should be sent immediately to open the window, since it is not reasonable to expect that any letter-level data will be sent. It is acceptable to send an angel-only datagram for the purpose of opening the offered window before it is completely closed. Care should be taken to only do this when the window will soon be closed, so that the number of packets transmitted for angel's internal purposes is limited.[32]

A.4.4 Initial Connection Protocol

An angel connection begins when a letter-level client sends a letter to a letter-level server using the procedure "AccessService." The sequence number of the first datagram sent should be one (1), and the "acknowledge to sequence number" field should be zero (0). The angel-client should assume an initial window of one (1) has been offered by the angel server.

The angel level at the client's end the (*angel-client*) begins by selecting a UDP port number at random with which to identify its side of the connection. It then wraps the initial letter in a datagram and sends it to the foreign letter level. Since the User Datagram Protocol does not distinguish between service identifiers and temporary port identifiers (TID's), an angel service is identified by the UDP port number allocated to the angel-server.

When the Angel-server receives the initial datagram from the angel-client, it calls the letter-server's handler predicate with the contents of the initial letter. If the predicate fails to hold, an exception datagram is sent as a courtesy to the angel-client (see A.4.5). Otherwise, the angel-server chooses a TID at random to be used for the duration of the connection, creates a new connection structure which has been updated to the initial datagram's header information, and notifies the letter-level so that the connection may begin.

The connection is now established. The angel-server may have to send an angel-only datagram indicating that the initial letter was delivered to the server; this is handled as is a zero-window condition.

Henceforth, the letter-client and letter-server may send letters to each other at any time. At the angel level, letters are immediately encoded into datagrams and are sent as soon as the windowing mechanism permits. Angel-only datagrams are not affected by the windowing mechanism, and so may be used to open windows when no letter-level data needs to be sent.

[32]When a letter which is expecting a response arrives and indicates that the window is closed as well, it is reasonable to set a dally timer before sending an angel-only datagram as if only the former condition held.

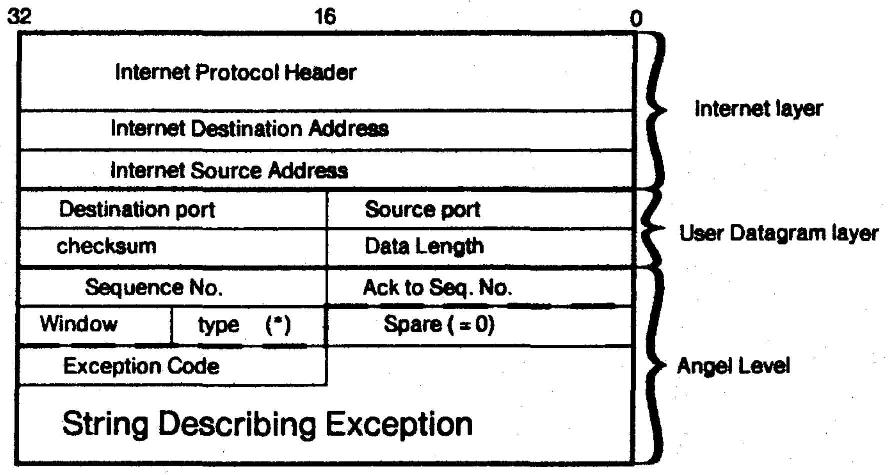
A.4.5 Exception Handling

Exception datagrams are used between angel-level software to transmit data on exceptional conditions when they occur. The most common sort of exception is when a letter-level entity decides to terminate a connection.

Exception datagrams are provided to allow foreign angel level software to determine more easily at which level the exception occurred, and to obviate the waste of system resources associated with determining that a connection is dead through a long time out. They are a courtesy provided by one angel implementation to another; the effect on the foreign connection will be the same regardless of whether or not an exception datagram that has been transmitted is successfully received.

There are two types of exception datagrams: *Exception* datagrams and *BadSocket* datagrams. In the former case, an existing connection has encountered and unrecoverable error, and the angel connection must be terminated. An Exception datagram is a datagram with the *Exception* bit set. In a datagram of this sort, any data following the angel header is assumed to be a netascii string describing the exception. A machine-understandable code describing the error is placed in what would normally be the "ResponseTo" field of the datagram. The format of an exception datagram is illustrated in figure A-2, along with a list of possible codes for the machine-understandable Error Number field.

Figure A-2: Format of an Angel Exception Datagram



*) Only exception bit may be set.

The following list describes the error codes that are defined for angel Exception datagrams. The exception code is placed in the position in the datagram that would be used for the "Response To Letter" field in a letter-bearing datagram.

Code	*Description*
0	No code exists for this error, see the string for a more complete description
1	Connection terminated by letter-level
2	System resources are used up
3	System is going down
4	The service that you are accessing has been terminated (this refers to a system-wide shutdown of a service, as opposed to the termination of a particular instance of that service).

A *BadSocket* datagram is sent by the angel level software when an angel datagram is received with a foreign-local TID pair that does not correspond to any existing angel socket. A BadSocket datagram should never cause another BadSocket datagram to be sent. The procedure for sending a datagram of this sort is as follows: the foreign and local internet addresses and the foreign and local UDP port numbers are switched; the BadSocket bit in the Angel header is set; the packet is transmitted.

Appendix B

SFAP: A Simple File Access Protocol

The following is a reprint of an internal document of the Computer Systems and Communications Group at the MIT Laboratory for Computer Science.

M.I.T. Laboratory for Computer Science
Computer Systems and Communications Group

Network Implementation Note No.29
July 11, 1980

SFAP: A Simple File Access Protocol

Geoffrey Cooper

B.1 Introduction

This document defines SFAP, a *Simple File Access Protocol.* This is a protocol for performing semi-random (i.e., block by block) access to files stored on other machines in a computer network. Since it is built on top of the User Datagram Protocol—which, in turn, employs the Internet Protocol—it may also be used to access files across network boundaries.

SFAP is most similar to TFTP, the *Trivial File Transfer Protocol.* It shares TFTP's aim of being easy to implement. SFAP does not try to solve the problem of simultaneous reads and writes to files; there is no facility for providing a clearly defined result in this case, or even a notification of the problem. As in TFTP, eight bit bytes of data are passed.

SFAP is more complicated than TFTP, since it allows freer access to remote files, and is thus open to several worst-case failures that can not occur in TFTP. In particular, messages must be numbered to make it possible to detect duplicated packets.

Currently there is only one mode of file access: binary mode. This provides for the transfer of raw eight bit bytes of data. It is assumed that the host requesting file access is familiar with the binary formats and file naming conventions of the owner of the file. It is felt that when this is not the case, non-sequential access into a file is not particularly useful, and is hard to implement; TFTP is a better solution.

B.2 Overview

SFAP views a file as an array of bytes, arranged into *blocks* of 512 bytes each. The last block of a file will always have less than 512 bytes. Thus, it may be of zero length if the file's length is an integral multiple of 512. Every file has at least one block. If the file is of zero length, SFAP views it as consisting of one block of zero length.

SFAP is used by a user on one host to communicate with a server on another host, to access files that are stored on that host. Every message in SFAP is numbered with a 'message number' (MNO). The current MNO is stored on both hosts as part of the state of the access. Messages with numbers lower than the current number are discarded. Thus, SFAP extends UDP to ordered messages, but does not provide a facility for recovering lost messages. Flow control is of the simplest sort; once a request has been sent, no further messages may be sent until the last outstanding one is received. In the case of lost requests or acknowledges, the requesting host will time out, and can re-transmit the request.

Access to a file using SFAP begins with an **open** request. This contains the overhead of initiating the access: the message number with which to start the connection, the name of the file to be accessed, and the access mode. Authentication, access privilege checking, and lock-checking, although currently not required or supported by SFAP, would also be performed at this time.[33] Once the **open** request has been acknowledged, access to the file is provided through the **read** and **write** requests. These specify the the number of the block to be read or written, and, in the case of write, give the data to be used. These requests are acknowledged (along with the data that was read, for a read request) when the requested action has been taken. If a request or an acknowledge is lost, the requesting party will time out of waiting for the acknowledge, and can retransmit the request. This will work, since two sequential writes or reads of the same data have the same effect as a single write or read.

Interdispersed with the read and write requests, may be a **describe** request. This request is acknowledged with useful information about the file being accessed. Subsequent describe requests, while permitted, will return the same information.

Finally, a **close** request is sent upon completion of access. This is a courtesy, since it might not be received, in which case the server would timeout and unilaterally close the connection. The close request is not acknowledged. Upon sending of a close request, the user may terminate.

B.3 Relation to Other Protocols

The relationship between this and other protocols is the same as that between TFTP and others; the reader is referred to [29].

[33]Use of access privilege is partially supported by the "describe" request, in that it is possible to query what privileges have been granted. No attempt is made to define how the privilege is granted or refused, however.

B.4 Retransmission Protocol

If a request or an acknowledge is lost, the problem will first be noticed by the user, who will time out waiting for the acknowledge. Both cases are noticed in the same way, so it is impossible for the user to tell whether the action was completed (the acknowledge was lost) or not (the request was lost).

To get around this we specify that both user and server store the last message that they transmitted. At the user end, this allows a message to be retransmitted if it is lost. Furthermore, when a message is received, its MNO is compared to that of the last message. If it is the same or less, the new message is assumed to be the result of a network packet duplication, and is discarded.

At the server end, when a new message is received, its MNO is compared to that of the last message. If they are the same, the server assumes that the new message is a retransmission of the previous one, and retransmits the last acknowledge. If the MNO of the newly received message is less that than that of the last acknowledge transmitted, the message is assumed to be a result of network packet duplication, and is discarded.

Most of the actions performed by an SFAP server will have the same effect if they are done twice in a row as once (e.g., reads and writes). It is no more than a savings of the server's time to save the last acknowledge transmitted for requests for such actions. This is not the case for the action of opening a file. Here the protocol will only work if the retransmitted open is *not* re-interpreted. The reader is referred to section B.5.1 for a full discussion of the problem.

B.5 Initial Connection Protocol

For a host, A, to access a file which resides on another host, B, a connection must first be established. To do this, A must send an open request to B. This consists of the message number with which to start the connection, the required status of the file, the name of the file to be accessed, and a variable number (which may be zero) of variable length fields. Each field is begins with a byte describing its contents, a byte giving its length, and then a number of data bytes. The entire list is terminated by a zero byte.

The required status field of the open request specifies whether the file is to be created or modified. The possible values for this are (assume the "netascii" values of the characters):

'C'	**create**. The file must not exist.

'M'	**modify.** The file must exist.
'D'	**destructive create.** If the file does not exist, it is created. If it does exist, it is truncated to zero length.

The 'destructive create' file status is the only way to shorten a file using SFAP.

The variable length fields at the end of the open message are used to specify optional arguments. Only one kind of field is currently defined. It is identified by a byte containing the value of the netascii character 'M', and contains a string (currently always "binary") specifying the mode to be used. If no mode field is present in the open message, the mode defaults to "binary". Other fields may be supported by servers on certain hosts, and are ignored if not serviced. The open acknowledge message contains a list of those fields that were looked at by the server, so that users may request any options without needing prior assurance that they are supported by the server being addressed.

The UDP header of the packet contains the known value, 70_{10}, in its **destination** field. The source field contains a AID (Access Identifier) that A has created. This is a number that is chosen at random, such that the probability of a number repeating within a short period of time is low. An AID of 70_{10} is illegal. Figure B-1-a summarizes the **open** packet.

When B receives the open request, he must verify that the named file exists, and may be accessed by A (how he determines A's privilege is B's problem). Then B creates an **open acknowledge** packet. The UDP header destination field of this packet contains A's AID. The source field contains a new AID that B has created. The open acknowledge packet contains a confirmation of the message number contained in the open. It also contains a list of variable length fields identical in format to those in the open request. Each field's opcode specifies the opcode of the request's field to which it is a response, and the data (if any) contains any data that the response might entail. The acknowledgment is then sent to A. Figure B-1-b contains a description of the open acknowledge packet.

In all subsequent messages during the connection, the AID's that A and B allocated initially are included in the UDP header. For requests (A←B[34]), the source field is A's AID, and the destination field B's AID. For acknowledgments (B←A), the source field is B's AID, and the destination field A's AID. A and B are also required to store the message number, or MNO, that the open request contained. For all subsequent requests, this number is incremented and included as the MNO of the request. At the server, all requests with a MNO less than or equal to the current MNO are discarded. Each request may thus be thought of as canceling the previous one.

[34] Read the arrow as "sends a packet to"

(a) **Request** (A ← B):

UDP header: source = A's AID — destination = 70

```
 1 byte    1 byte       2 bytes      string     1 byte   n bytes     1 byte
 ---------------------------------------------------------------------------
|  02    | file status | current MNO | Filename |   0    |  fields ... |   0  |
 ---------------------------------------------------------------------------
```

Optional Field:

```
  1 byte    1 byte      n bytes
 ------------------------------
| opcode | count (n) |   body   |
 ------------------------------
```

(b) **Acknowledge** (B←A):

UDP header: source = B's AID — destination = A's AID

```
 1 byte     2 bytes      n bytes
 ---------------------------------
|  03    | current MNO | field list |
 ---------------------------------
```

Figure B-1: Summary of OPEN Sequence

B.5.1 Error Recovery on Open: Things That Go Bump in the Net

Retransmission of open requests presents a special problem in SFAP, since if no acknowledge is received from an open request with file status *must not exist*, the user does not know whether the file has been created or not. If the file did not exist, but was created in response to the original request, it is essential that the the retransmitted open not be interpreted, or it would flag an error; it would not be possible to distinguish this from the case where the file really did exist, and either the original open request or the error packet it generated was lost.

Nonetheless, the user retransmits the open request if he does not receive an acknowledge. In accordance with the retransmission protocol, described above, however, the server will notice, if the open request had previously been received, that the MNO of newly received request is the same as that of the last acknowledge that the server has sent. The server will then retransmit the acknowledge from the previous request, *without* re-interpreting the 'required file status' field of the open. Thus, no error will be detected for the retransmitted open request that was not detected for the original. Of course, had the server never received the original open request, it would interpret this one, and the result would be equally satisfactory.

B.5.2 Re-Open Syntax

At any time during the life of a connection, the user may discover that no responses are being received from the server, and may suspect that either the server is down, or the requests or their acknowledges have been lost, or the server is up, but had timed out and unilaterally closed the connection. Whenever the user determines (through an implementation dependent number of unanswered requests) that he is getting no response from the server, he should try to 're-open' the connection. To leave the door open for a simple addressing of user authentication and privilege problems, SFAP does not provide for a true 'resume connection' function. The approach taken is to abort the current connection and request a new one. As a courtesy, the user sends an error packet of invalid opcode (=0) and appropriate error number (=3). The packet must specify that the connection has been closed. Then the user creates a new AID, and opens a new connection. Note that the 'required file status' field of the new open should always be 'must exist' to have the correct effect.

If the server is down, the user will timeout from waiting for the open acknowledge, and can abort the connection. If the server has not closed the previous connection, but the various requests have been lost in transmission, that connection will soon be aborted.[35]

Even once the connection is successfully re-opened, the user can not be sure whether the last request before the re-open was carried out or not. If this request were a read or describe request, there is no problem, since these are passive. But if it were a write request, it would need be retransmitted to ensure the consistency of the file.

[35]Note that the re-open syntax described here is possible only because SFAP does not address the problem of multiple accesses to a file. For example, if it is specified that when a connection is aborted all changes to the file done through it should be undone, a re-open would need have the effect re-establishing the old connection.

B.6 Read/Write Sequence

Once A has established a connection to B, access to the file may begin. This is accomplished using the **read** and **write** requests. A read request specifies the current MNO, and the number of the block in the file to be read (the first block is block #0). It is acknowledged, assuming all is well, by a packet containing the number of the block that was read, the number of bytes read from the block, and the data in the block. The number of bytes read will always be 512—the SFAP block size—unless the block is the last in the file. In this case, the number of bytes read is the number of bytes that exist in the block. This may be zero, if the file's length is a multiple of 512. A summary of the read request and acknowledge is given in figure

(a) **Request** (A←B):

UDP header: source = A's AID — destination = B's AID

```
 1 byte   2 bytes   2 bytes
 ---------------------------
|  04   |   MNO   | Block # |
 ---------------------------
```

(b) **Acknowledge** (B←A):

UDP header: source = B's AID — destination = A's AID

```
 1 byte   2 bytes   2 bytes       2 bytes           n bytes
 ------------------------------------------------------------
|  05   |   MNO   | Block # | no. bytes read (n) | Data      |
 ------------------------------------------------------------
```

Figure B-2: Summary of READ sequence

The write request contains the current MNO, the number of the block to be written, the number of bytes to write, and the data to write into the block. A write request may be used to replace specified portions of a file, or increase its length. Thus, when the number of bytes to write is 512, the specified block is replaced by the data given; if the block is the last block of the file, the file is extended to contain a new, empty block after the one being written. A write of fewer than 512 bytes must be directed to the last block of the file, and must write at least as many bytes as were previously in that block. Attempting to write a shorter last block of the file or a block that is not the last block with less than 512 bytes, results in an error. A summary of the write request is given in figure B-3

B.6.1 Things that Go Bump in the Net

If an error occurred when B tried to carry out a read or write request, an error packet would be returned. If an acknowledge were lost, A would time out from waiting for it, and could retransmit the request. If this occurred several times, A should attempt to re-open the connection as described in the section on "Re-Open Syntax."

(a) **Request** (A←B):

UDP header: source = A's AID — destination = B's AID

```
 1 byte   2 bytes   2 bytes    2 bytes                 n bytes
 ----------------------------------------------------------------
|  06   |   MNO   | Block # | no. bytes to write (n) | Data      |
 ----------------------------------------------------------------
```

(b) **Acknowledge** (B←A):

UDP header: source = B's AID — destination = A's AID

```
 1 byte   2 bytes   2 bytes
 ----------------------------
|  07   |   MNO   | Block # |
 ----------------------------
```

Figure B-3: Summary of WRITE sequence

B.7 The 'Describe' request

In some applications, it is desirable to have certain information about the file being accessed. Since determining this information may be costly (see the final section for a discussion of this) and the information will probably not usually be needed, it is not desirable to return the information in the open acknowledge. Thus, the **describe** request is included.

The format of the request is summarized in figure B-4-a. As in all messages, the AID's of the host accessing the file and the host on which the file is stored are contained in the UDP header of the packet.

The host on which the file is stored returns an acknowledgment packet containing a number of fields. Each field is preceded by a single byte tag, describing its contents. Next is a byte specifying the number of bytes of data the field contains. Finally, the data is included. The list of fields is terminated by a zero byte.

There are currently two defined fields. One, tagged 'L', contains four bytes of data. The first two bytes form the sixteen bit number of the last block in the file. The other two bytes form the nine bit offset in the file's last block of the last character in the file. The seven most significant bits of this number must always be zero. The other field, tagged 'P', contains a single byte describing the access privilege that has been granted the user. Bit 0 of this byte (LSB) is 1 when write requests will be honored (will not generate an error) during this connection. Bit 1 is 1 when the read requests will be honored. The other bits of this word may be used by cooperating pairs of hosts to provide more information. Other fields may be included by different servers, and should be skipped over by users that do not want them or do not know how to interpret them. The idea is for every server to supply what information is possible for it to supply, without requiring that all users know or care what all the information means.

A summary of the describe request and acknowledge is given in figure B-4.

B.8 Closing the Connection

To terminate a connection, A—the host that initiated the connection—should send a **close** request to B. This is a courtesy, since if the close request is lost, B will timeout and unilaterally close the connection in any event. There is no close acknowledge, so the user may terminate after sending the close request. A summary of the close request is given in figure B-5.

If the server should timeout, and decide to unilaterally close the connection, an appropriate error packet should be sent as a courtesy.

B.9 The Error Packet

If at any time a request can not be honored by the host on which the file is stored, an error packet is returned. Most errors do not terminate a connection (though, of course, any error on an open request would prevent the establishment of one). An error packet may also be sent by the user during an attempt to re-open a connection.

An error packet consists of an opcode, an error code, and a string containing a short description of the error. The error code consists of two bytes. The first of these is the opcode of the request that prompted the error packet. The 7 least significant bits of the second byte contain a number describing the type of error that occurred. If the most significant bit of this byte is set (= 1), the error has caused the connection to be closed. The protocol does not

(a) **Request** (A←B):

UDP header: source = A's AID — destination = B's AID

```
 1 byte   2 bytes
 ----------------
| 08    |  MNO    |
 ----------------
```

(b) **Acknowledge** (B←A):

UDP header: source = B's AID — destination = A's AID

```
  1 byte   2 bytes    n bytes              1 byte
 ------------------------------------------------
|   09    | MNO    | describe fields ...  |  0   |
 ------------------------------------------------
```

Describe field (general form):

```
         1 byte    1 byte      n bytes
        ------------------------------
       | opcode | count (n) |   body   |
        ------------------------------
```

Privilege Describe Field:

```
 1 byte  1 byte    6 bits      1 bit      1 bit
 -------------------------------------------------
|  'P'  |   1   |  unused   | Can Read | Can Write |
 -------------------------------------------------
```

Figure B-4: Summary of DESCRIBE sequence

(a) **Request** (A←B):

UDP header: source = A's AID — destination = B's AID

```
 1 byte  2 bytes
 ---------------
| 10    | MNO  |
 ---------------
```

Figure B-5: Summary of CLOSE sequence

specify which errors will cause this to happen, to allow for improved service when communicating with hosts that have good error recovery facilities. Figure B-6 summarizes the format of an error packet. Figure B-7 contains a list of error numbers.

UDP header: source = B's AID — destination = A's AID

```
 1 byte 2 bytes    2 bytes         string        1 byte
 -------------------------------------------------------
|  256 |  MNO  |  Error Code  |  Error Message  |  0   |
 -------------------------------------------------------
```

Error Code:

```
          1 byte                  1 bit            1 byte
    ----------------------------------------------------------
    | opcode of request | closed connection | error number |
    ----------------------------------------------------------
```

Figure B-6: Summary of ERROR Packet Format:

Error numbers are listed by the opcodes that generate them. Those listed as '*' can be generated by any opcode.

op, er #	**description**
0, 3	Aborting this connection due to timeout (May be sent by user or server). This implies that the 'abort' is set (i.e., the 2nd byte is $128 + 3 = 131_{10}$)
*, 0	Undefined; see error message, if any.
*, 1	Access privilege denied
*, 2	Illegal SFAP operation.
open, 3	Access denied due to lack of privilege.
open, 4	Illegal mode
open, 5	Specified file may not be accessed in specified access mode.
read, 3	Read access denied
read, 4	Read of non-existent block requested
write, 3	Write access denied
write, 4	Write of non-existent block requested. The file may only be extended at the last block
write,5	Disk full or allocation exceeded

Figure B-7: Error Numbers

B.10 Future Extensions

For communications between machines with radically different file systems, it would be desirable to support some sort of "netascii" transfer mode. The problem is one of efficiency; the only way to properly find the n^{th} block of a file in this mode would be to convert all of the file up until that block. However, assuming that most files will be read in order (or in order from a point on), some simple but effective optimizations might be carried out by the server supporting the mode. In fact, the possibility of this being useful is the justification for the 'describe' request; pre-processing would be needed to determine a file's length when it is being accessed in "netascii" mode. Forcing the entire file to be pre-processed when it is opened would make it impossible to implement a "convert as needed scheme."

Other extensions would be towards support of file security. Identification and authentication of the host requesting access would be useful. More likely, these issues would be addressed by a separate protocol, built on this one.

B.11 Packet Formats (without header)

Open:

```
 1 byte     1 byte         2 bytes      string      1 byte   n bytes      1 byte
 ------------------------------------------------------------------------------
|  02    | file status | current MNO | Filename |   0    |   fields ... |   0   |
 ------------------------------------------------------------------------------
```

Open Field (general form:)

```
  1 byte     1 byte      n bytes
 ------------------------------
| opcode | count (n) |   body    |
 ------------------------------
```

Open Acknowledge:

```
 1 byte      2 bytes       n bytes
 ---------------------------------
|  03    | current MNO | field list |
 ---------------------------------
```

Read:

```
 1 byte   2 bytes   2 bytes
 ---------------------------
|  04   |   MNO    | Block # |
 ---------------------------
```

Read Acknowledge:

```
 1 byte   2 bytes   2 bytes         2 bytes            n bytes
 ------------------------------------------------------------
|  05   |   MNO    | Block # | no. bytes read (n) | Data      |
 ------------------------------------------------------------
```

Write:

```
 1 byte   2 bytes   2 bytes    2 bytes                  n bytes
 ---------------------------------------------------------------
|  06   |   MNO    | Block # | no. bytes to write (n) | Data      |
 ---------------------------------------------------------------
```

Write Acknowledge:

```
 1 byte   2 bytes   2 bytes
 ---------------------------
|  07   |   MNO    | Block # |
 ---------------------------
```

Describe:

```
 1 byte   2 bytes
 -----------------
|  08   |   MNO   |
 -----------------
```

Describe Acknowledge:

```
  1 byte   2 bytes    n bytes              1 byte
 ---------------------------------------------------
|   09   |  MNO   |  describe fields ...  |  0   |
 ---------------------------------------------------
```

Describe field (general form):

```
  1 byte    1 byte     n bytes
 --------------------------------
| opcode | count (n) |   body   |
 --------------------------------
```

Privilege Describe Field:

```
 1 byte  1 byte    6 bits      1 bit      1 bit
 ---------------------------------------------------
|  'P'  |   1   |  unused  | Can Read | Can Write |
 ---------------------------------------------------
```

Close:

```
 1 byte  2 bytes
 ---------------
|  10   |  MNO  |
 ---------------
```

Error:

```
 1 byte 2 bytes    2 bytes         string        1 byte
 --------------------------------------------------------
|  255 |  MNO  |  Error Code  |  Error Message  |  0   |
 --------------------------------------------------------
```

Error Code:

```
   1 byte                  1 bit              1 byte
 ---------------------------------------------------------
| opcode of request | closed connection | error number |
 ---------------------------------------------------------
```

Appendix C

SFAP II Protocol Specification

C.1 Introduction

SFAP, the Simple File Access Protocol, is a protocol which allows random access to files stored on computer systems connected to packet-switched networks. The protocol is built on top of the Angel protocol (see appendix A).

It is the goal of SFAP to be simple. The protocol does not deal with the problems of user authentication, file system or network protection, multiple access to remote files, or the conversion of the data storage conventions of one file system into those of another (for example, the end of line character). All of these problems must either be built into higher level software (where possible) or must be left unsolved.

Access to a remote file is provided by establishing an Angel protocol connection between an SFAP client and an SFAP server. Once the connection is established, the client sends the server an ordered sequence of requests, which are serviced and acknowledged in order. The connection is terminated when either the client or server encounters an unrecoverable error, or when the client has finished with the connection. SFAP presumes that it is possible to gain the effect of re-establishing a lost connection by opening a new one, even though there are file systems on which it is not natural (or perhaps possible) to implement this behavior.

C.2 Network File Representation

Although SFAP does not interpret the contents of a file to which it gives access, it must be able to translate the operations of random file access across system boundaries. To do this, SFAP imposes a standardized format for files.

A file is an array of 8-bit bytes (sometimes called *octets*). The bytes in a file are arranged into smaller quantities called *blocks*. A block consists of a block number, and a number of bytes. Every block but the last in the file must contain exactly 512 bytes, and the last block must contain less than 512 bytes (if the file length is a multiple of 512 bytes, then the last block will contain zero bytes).

The block number determines the position in the file of the bytes in a block. First block in the file is numbered block one (1). If a file is long enough to contain more than 65535 (2^{16}-1) blocks, only the first 65535 blocks may be accessed using SFAP. There is no block numbered zero.

```
         +------------------------------------------------------------+
27+5 bit +                 0             | Exact | CT | WT | RT | Ln +
         +------------------------------------------------------------+
  32 bit +                    Creation Time                            +
         +------------------------------------------------------------+
  32 bit +                     Write Time                              +
         +------------------------------------------------------------+
  32 bit +                      Read Time                              +
         +------------------------------------------------------------+
  32 bit +                      Length                                 +
         +------------------------------------------------------------+
 6+2 bit                 +    0    |RA|WA|
                         ----------------
   8 bit                 +   Open Mode  +
         +------------------------------------------------------------+
 N*8 bit +                    File Name                                +
         +         (ascii bytes, until end of letter)                  +
         +------------------------------------------------------------+
```

Figure C-1: Format of the File Descriptor Block

Field Name	*Description and Use*
Exact	This bit is used on Open and Describe requests to indicate whether an error should be returned if any of the fields requested can not be filled in by the server.
Field Tags	These bits, labeled CT, RT, WT, and Ln, are used on the Describe request to indicate a special need for the Creation Time, Read Time, Write Time and Length fields of the descriptor block, respectively. On Describe and Open responses, they are used to indicate which of the above fields were actually filled in by the server.
Creation Time	The time the file was created. All times are 32-bit unsigned numbers, indicating the number of seconds since midnight GMT on January 1, 1900.
Write Time	The time the file was last written.
Read Time	The time the file was last read.
Length	The number of bytes in the file.
Read Access	A bit indicating whether blocks of the file may be read.
Write Access	A bit indicating whether blocks of the file may be written.
Open Mode	The mode in which the file was opened. Described in section C.4.
Name	The full name of the file (in some systems this may be different from the name used to open the file). The file name is an ASCII string, whose length may be inferred from the Angel letter length.

Figure C-2: Description of the Fields in the Descriptor Block

A block of data known as the *descriptor block* is associated with the file being accessed. The descriptor block of a file is not part of the file but contains information about it. The descriptor block is divided into a number of fields. Figure C-1 outlines the format of the descriptor block, and figure C-2 contains a brief description of each of its fields.

On many file systems, it is either impossible or somewhat inefficient to fill in all the fields of the descriptor block. For example, not all file systems distinguish between the "creation" and "last written" dates of a file, and, in some file systems, all the records of a file must be read in order to determine its length in bytes. It is up to the implementor of each SFAP server to determine the mapping between the information available from the local file system and that put in the descriptor block. When there is no way of filling in a field in the descriptor block, the field may be left "blank" by the server. The *field tags* in the descriptor block are used to distinguish which fields contain valid data.

C.3 Network Data Representation

SFAP uses the Angel protocol to transfer data between the client and server. Angel provides its users with a prototype data representation based on datagrams known as *letters*. This section describes the way in which the data transmitted over the network by SFAP corresponds to Angel's "letter-level." We presume here that the reader is familiar with the terminology of the Angel protocol.

An SFAP server and user communicate by sending each other *messages*. Each message fits into a letter. The function of a message is determined by its *type*, which corresponds directly to the Angel letter type.

Although there are many different message types defined in SFAP, there are only two formats for messages. One message format is used for messages of type Error; all other message types are in the other *standard* format. Messages of type Error (or error messages) are used to indicate exceptional conditions which may or may not result in aborting the connection. The format and usage of Error messages is described in section C.5.

Some messages will require a response, while others will not. The need for a response to a given message is transmitted using the angel letter-response facility. The form of a response is always the same as that of the request, except that additional data might be included, as in a request to read a block of the file. The circumstances under which a response is and is not required for a given message is describe in the next section.

The standard message format is described in figure C-3. The message type is placed in the Angel *letter type* field, sixteen bits contain a block number (see below), and sixteen bits are reserved for future extensions to the protocol. The rest of the message consists, when necessary, of a block of data which is either a block of the file or the descriptor block. When the descriptor block is called for, the block number field is zero.

```
+------------------------------+
+ letter type = message type   +
+------------------------------------------------------------+
+      block number  (N)       +           0                 +
+------------------------------------------------------------+
+                                                            +
+     Block N itself, when needed (see text)                 +
+                                                            +
+------------------------------------------------------------+
```

Figure C-3: Format of a Standard Message

C.4 Connection Protocol

An SFAP client initiates an SFAP connection by sending a standard message of type ***Open***[36], containing the file descriptor block. The *field tag* bits of the descriptor block must all be zero, i.e., only the name of the file, the Read/Write Access bits, and the Open mode byte are filled in. The Read/Write Access bits indicate whether read and/or write access to the indicated file is requested. A response to this message is required. The Open mode is one of:

1	File must exist.
2	File must not exist; create it.
3	File may or may not exist. If it does not exist, create it.
4	If file exists, truncate it to zero length; otherwise create it.

When SFAP server receives the request, it may decide to refuse it, by sending an error message, to accept it, or to accept it with different access conditions than were requested. For example, a server might always grant both read and write access, or might never grant write access. The *Exact* bit in the descriptor block may be used to cause an error unless the server is willing to grant exactly the access conditions that were requested. The Open mode must always be interpreted this way.

To accept the connection, the SFAP server responds to the Open message with an Open message of its own, filled in with the access conditions actually granted, with the Open mode as requested and with the name of the file being accessed (this may be different from the name used to open the file, if the server used system-dependent defaulting conventions). Other fields may or may not be filled in by the server; the *field tag* bits are used to indicate

[36]Numeric values for the message types referred to herein are given in section C.6

which fields have been filled in, as described in figure C-2. Normally, a server will fill in descriptor block fields that are readily accessible to it, to increase the efficiency of those clients which need only this information.

Once the connection is open, the client may send additional requests to the server to read or write blocks of the file, or the read the file's descriptor block. Requests must always be processed in the order that they were sent.

To read block *N* from the file, a message of type ***Read*** is sent, containing the number of the block to be read. The response is the same message, with the block of data included.

To write block *N* of the file, a message of type ***Write*** is sent, containing the number of the block to be written, and the data to write into it. The response, if required, is the same message, without the block of data. A write to the last block of the file may extend but must not truncate the file. An increase in the number of blocks in a file is accomplished by writing 512 bytes to the last block, which causes a new "last block" with zero bytes in it to come into existence.

There is no need for the server to send a response to a write request, unless the Angel *ResponseRequired* field is set for the request. Sometimes, a number of read and write requests may become "batched" at the client, and the client may wish to send them all to the server at once. In this case, it is only necessary to set the ResponseRequired field for the last message.

To read the descriptor block, a message of type ***Describe*** is sent, containing a block number of zero, and a "prototype" descriptor block for the file. The field tags in the prototype descriptor block indicate fields which should be filled in if possible, even if it is computationally expensive to do so. If the *Exact* bit is set, then an error message is generated if any of the requested fields is not available from the server's file system.

To close the connection, the client closes the underlying angel connection. The server may not close the connection except in error.

C.5 Errors

At any time during the course of a connection, an error may occur. An error need not terminate the connection, but any error may do so. Errors include events external to SFAP, such as system shutdown, file system corruption, etc.. The only event that may *not* provoke an error message at either the SFAP client or server is the receipt of an error message.

The format of an error message is given in figure C-4. The type of the message is ***Error***. The message contains the letter ID of the offending message, its type, and block number. A number identifying the error as one of a class of SFAP errors, and an ascii string which may give more information completes the message. A list of SFAP error numbers is given in section C.7. The TC bit, when set, indicates that the connection must be terminated as a result of the error.

An error message must always require a response. If TC is set, then the connection may be terminated as soon as the response is received. The response to an error message is a message of type Noop (the convention of responding to a message by a message of the same type has obvious complications here).

Since SFAP allows the client to send more than a single request to the server before receiving any response, error handling becomes more complex. Say the SFAP server encounters an error handling a client's request message with letter ID *N*. The server sends the client an Error message, with response required, with *N* in the "Error on LetterID" field. Then the server ignores all further messages from the client until it receives the desired response. The client, upon receiving the error message, must assume that all requests sent since the specified letter ID. It must also respond immediately with a letter of type Noop and length zero.

When the server receives the Noop message, it may abort the connection (if it needs to do so). Otherwise the connection continues normally.

C.6 Message Type Numbers

The following table gives numeric values for the SFAP message types:

Message Type	Numeric Value
Open	1
Describe	2
Read	3
Write	4
Noop	5
Error	6

```
0                                16                              64
+--------------------------------+
+   letter type = Error          +
+---------------------------------------------------------------+
+   Error on LetterID            +    Error on Type             +
+---------------------------------------------------------------+
+   Error on Block No.           +TC|     Error Number        |TC+
+---------------------------------------------------------------+
+                                                               +
+   Error message (ascii string, to end of message)            +
+                                                               +
+---------------------------------------------------------------+

 (TC = Terminate connection)
```

Figure C-4: The Format of an Error Message

C.7 Error Numbers

The following table lists error numbers which are defined for all SFAP connections.

Number	*Meaning*
0	Random error. The error message contains more information.
1	Access Denied. Attempt to read on a write-only connection or write on a read-only connection.
2	Bounds Error. The block that was referenced does not exist.
3	Disk Full. A write caused the disk allocation limit to be exceeded.
4	Truncate Error. A write of a block containing less than 512 bytes was requested to other than the last block of the file, or an attempt was made to overwrite the last block of the file with less data than it contained.

References

1. D. Boggs, J. Shoch, E. Taft, R. Metcalfe. Pup: An Internetwork Architecture. *IEEE Transactions on Communications* (April 1980).

2. A. Chapin. Connectionless Data Transmission. *Computer Communication Review 12*, 2 (April 1982).

3. D. Clark, K. Pogran, D. Reed. An Introduction to Local Area Networks. *Proceedings of the IEEE 66*, 11 (November 1978).

4. D. Cohen & J. Postel. On Protocol Multiplexing. *Proceedings of the Sixth Data Communications Symposium* (November 1979).

5. D. Cohen. Flow Control for Real-Time Communication. *Computer Communications Review 10*, 1&2 (January/April 1980).

6. S. Crocker, J. Heafner, R. Metcalfe, J. Postel. Function-oriented Protocols for the ARPA Computer Network. *Proceedings of the AFIPS Sprint Joint Computer Conference 40* (1972).

7. R. Cypser. *Communications Architecture for Distributed Systems.* Addison-Wesley, Reading, Massachussets, 1978.

8. Xerox/Intel/DEC. The Ethernet/A Local Area Network/Data Link Layer and Physical Layer Specifications. Xerox Corporation, Intel Corporation, Digital Equipment Corporation, September, 1980.

9. F. Heart, R. Kahn, S. Ornstein, W. Crowther, D. Walden. The Interface Message Processor for the ARPA Network. *Proceedings of the AFIPS Sprint Joint Computer Conference 36* (1970).

10. Arpanet Network Information Center. Internet Protocol Implementation Guide. SRI International, August, 1982.

11. ISO-SC 16. Reference Model of Open Systems Architecture. Tech. Rep. ISO/TC97/SC16, International Standards Organization, November, 1978.

12. I. Jacobs, R. Binder, E. Hoversten. General Purpose Packet Satellite Networks. *Proceedings of the IEEE 66*, 11 (November 1978).

13. P. Janson. Using Type Extension to Organize Virtual Memory Mechanisms. Ph.D. Th., MIT, September, 1976. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-167

14. L. Kleinrock, H. Opderbeck. Throughput in the Arpanet - Protocols and Measurement. *Proceedings of the Fourth Data Communications Symposium* (October 1975).

15. B. Lampson, R. Sproull. An Open Operating System for a Single-User Machine. *Proceedings of the Seventh Symposium on Operating Systems Principles* , ACM Order # 534790 (December 1979).

16. B. Liskov, et al. *CLU Reference Manual.* Springer-Verlag, New York, New York, 1981.

17. D. Moon. Chaosnet. Tech. Rep. AIM 628, MIT Artificial Intelligence Laboratory, June, 1981.

18. B. Nelson. Remote Procedure Call. Ph.D. Th., Carnegie-Mellon University, 1981. CMU report number CMU-CS-81-119; also available as Xerox PARC report number CSL-81-9

19. Xerox. Internet Transport Protocols. Tech. Rep. XSIS 028112, Xerox Corporation, December, 1981. This document contains protocol specifications for a variety of protocols, including the Sequenced Packet Protocol, which is mentioned in the text.

20. J. Postel, E. Feinler. Arpanet Protocol Handbook. Tech. Rep. NIC 7104, USC-ISI, January, 1978.

21. J. Postel. User-Datagram Protocol. ARPANET RFC # IEN-88, USC-ISI, May, 1979. This RFC is reprinted in [23]

22. J. Postel. Telnet Protocol Specification. ARPANET RFC # RFC-764/IEN-148, USC-ISI, June, 1980. This RFC is reprinted in [23]

23. J. Postel. Internet Protocol Handbook. Tech. Rep. RFC 774, USC-ISI, October, 1980.

24. J. Postel. DoD Standard Internet Protocol. ARPANET RFC # RFC-791, USC-ISI, September, 1981. This RFC is reprinted in [23]

25. L. Pouzin, H. Zimmermann. A Tutorial on Protocols. *Proceedings of the IEEE 66*, 11 (November 1978).

26. D. Ritchie, K. Thompson. The UNIX Time-Sharing System. *Communications of the ACM 17*, 7 (July 1974).

27. J. Saltzer. Protection and the Control of Information Sharing in Multics. *Communications of the ACM* (July 1974).

28. J. Saltzer, D. Reed, D. Clark. End-to-End Arguments in System Design. IEEE, April, 1981.

29. K. Sollins. The TFTP Protocol. ARPANET RFC # RFC-783, MIT-LCS, June, 1981. This RFC is reprinted in [23]

30. V. Strazisar. How to Build a Gateway. ARPANET RFC # IEN-109, BBN, August, 1979.

31. D. Swinehart, G. McDaniel, D. Boggs. WFS: A Simple Shared File System for a Distributed Environment. *Proceedings of the Seventh Symposium on Operating Systems Principles* , ACM Order # 534790 (December 1979).

32. A. Tannenbaum. *Computer Networks.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

33. USC-ISI. DoD Standard Transmission Control Protocol. ARPANET RFC # RFC-793, DARPA, January, 1980.

34. L. Tymes. Routing and Flow Control in TYMNET. *IEEE Transactions on Communications COM-29*, 4 (April 1981).

35. CCITT. Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment for Terminals Operating in the Packet Mode on Public Data Networks. Tech. Rep. Recommendation Number X.25, CCITT, November, 1978.