December 1986

Report No. STAN-CS-86-1137

The Leaf File Access Protocol

History and Specification

by

Jeffrey Mogul

Department of Computer Science

Stanford University
Stanford, CA 94305

The Leaf
File Access Protocol

History and Specification

Jeffrey Mogul
Department of Computer Science
Stanford University

1 December 1986

The Leaf File Access Protocol

History and Specification

Jeffrey Mogul
Department of Computer Science
Stanford University

Abstract

Personal computers are superior to timesharing systems in many ways, but they are inferior in this respect: they make it harder for users to share files. A local area network provides a substrate upon which file sharing can be built; one must also have a protocol for sharing files. This report describes ***Leaf,*** one of the first protocols to allow remote access to files.

Leaf is a remote file access protocol rather than a file transfer protocol. Unlike a file transfer protocol, which must create a complete copy of a file, a file access protocol provides random access directly to the file itself. This promotes sharing because it allows simultaneous access to a file by several remote users, and because it avoids the creation of new copies and the associated consistency-maintenance problem.

The protocol described in this report is nearly obsolete. It is interesting for historical reasons, primarily because it was perhaps the first non-proprietary remote file access protocol actually implemented, and also because it serves as a case study in practical protocol design.

The specification of Leaf is included as an appendix; it has not been widely available outside of Stanford.

Key words and phrases: Leaf, Sequin, remote file access protocols, distributed systems, network file systems

Table of Contents

List of Figures

List of Tables

1 Introduction

Personal computers are superior to timesharing systems in many ways, but they are inferior in this respect: they make it harder for users to share files. A Local Area Network (LAN) provides a substrate upon which file sharing can be built; one must also have a protocol for sharing files. This report describes ***Leaf,*** one of the first protocols to allow remote access to files.

When computer networking arose, computers were few and far between. The simplest way to share files across high-delay long-haul networks was to move an entire file; a File Transfer Protocol (FTP) is used for this purpose. FTP is quite useful, but true sharing requires more than this.

The trouble with FTP is that it creates a copy of the file at the destination host. This means that:

- Updates to the copy are not automatically reflected in the original. This complicates sharing, because two users may have different views of the file if one of the copies is updated. If both copies are updated, there may be no consistent view of the file; this makes it impossible to use FTP for shared databases.
- Making a complete copy is wasteful if only part of the file is needed. FTP is best suited to applications that read the entire file; some applications need access only to a small part of a file, usually because the file is a random-access database. Even applications that use sequential text files might read only part of a file. Copying a huge file in its entirety when only a few bytes are of interest wastes significant processor time; if the application runs on a workstation, the copy might not fit at all on the local disk.
- Copying an entire file imposes a high initial latency from the time the user asks for the file until the first byte is available; most FTP implementations require that the entire file be copied before the user can reliably access the first bytes. This reduces the opportunity for parallelism between file transfer (which often leaves the workstation processor idle for usable periods) and file processing, such as parsing or display.

In spite of these problems, several successful systems have been constructed using whole-file copy for workstation access to shared files. Examples include the ***VICE*** file system, part of the ***Andrew*** system [18], and the Cedar File System [26]. These two systems succeed not by solving the problems of whole-file copies but by avoiding them; their applications require neither simultaneous sharing nor large files.

Instead of making a copy of a file at the site where the application runs, we can take a different approach: leave the file where it is, and perform file access operations over the network. This mechanism is called Remote File Access (RFA). RFA solves the problems associated with making file copies; moreover, it allows the use of diskless workstations, resulting in cheaper, smaller, quieter, and cooler machines that can live unobtrusively on desks without sacrificing functionality or performance.

RFA can be implemented using a general-purpose remote operation protocol, such as Remote Procedure Call (RPC) or simpler message-based services. It can also be done using a File Access Protocol (FAP), a special-purpose protocol for RFA. While RPC-based systems have become quite popular, FAPs have the advantages that they can be based on simpler mechanisms; they do not depend on the availability of a remote invocation protocol, and they do not suffer from the costs of protocol layering.

Leaf was one of the first FAPs usable in a heterogeneous environment. My intent is not to push Leaf as the best way of doing RFA, for clearly it is not. Rather, I will relate the history of the Leaf protocol, and attempt to extract useful lessons about file access, protocol design, heterogeneous systems, and in general the process and context of computer systems design.

I begin in section 2 by covering the history of Leaf in as much detail as is now available, almost a decade after it began. Section 3 gives a technical overview of Leaf, describing the operations it supports and the protocol architecture. In section 4 I evaluate the protocol according to several criteria. Appendix I contains the specification of the Leaf protocol, for although Leaf is fairly widely used, the specification has never been generally available.

2 History of Leaf

In this section, I examine the historical context of Leaf. It is not always clear where various ideas first arose, or who first thought of them; the citations given are the best I was able to find.

Leaf was an evolutionary step in the design of network file access. Earlier protocols did not allow user programs to perform those operations on remote files that could be performed on local files; the semantics of local and remote files differed. More recent designs not only allow local and remote files to be treated the same, but provide exactly the same "syntax" (programming interface); some follow the transparency paradigm to the point where local and remote file access are indistinguishable. Leaf falls between these two points; while programs must use a different mechanism to access remote files, essentially all the operations that an application can use locally can also be used remotely.

2.1 Antecedent Protocols

Almost from the beginning of computer networks, there have been file transfer protocols (FTPs). The first FTP for the ARPAnet was proposed in 197 1 [2]. File transfer over networks was a significant improvement over previous methods, such as magnetic tape. Given the relatively low bandwidth and high latency of early networks, and the lack of applications that could make direct use of networking, users were satisfied to use a FTP utility program to transfer files.

Relatively soon, however, proposals were made to support direct access by application programs to remote files. The UCSB network file system [33], proposed for the ARPAnet the same month as the first ARPAnet FTP, was a partial step in this direction. It only supported sequential access; write operations could only append to the end of a file, not modify existing contents, but read operations could start at arbitrary offsets, after use of a skip operation to position the read pointer.

A more general File Access Protocol (FAP) was proposed for the ARPAnet in 1973 [7]. The ARPAnet FAP supported random access, including modification of existing data. It was recognized that "FAP (to as large as degree as is practical) should allow remote users to access files in the same way as local users may" [31]. Although it was also recognized, 'to meet this goal, that "FAP should provide means for a remote user to acquire certain status and 'descriptor' information about a given file" [31], it is not clear if the protocol design was ever changed to support this. The ARPAnet FAP was never implemented nor was any other proposed for the ARPAnet or Internet communities [24].

By 1976, Digital Equipment Corp. had done the first implementations of the Data Access Protocol (DAP), part of the Digital Network Architecture (DNA) [6] and now part of DECNET. DAP provided random access to remote files, and was meant for use in heterogeneous networks. Since the original implementations were for the RT-11 and RSX- 11M operating systems, both of which are more suited to technical computing than for use as workstation or timesharing systems, DECNET did not come into wide use for several years. While support for heterogeneity was an explicit goal of DNA, it is proprietary and so has only been implemented for Digital's operating systems, and even there does not provide complete semantic or syntactic transparency. Still, DAP may well have been the first heterogeneous file access protocol implemented.

While file access protocols were struggling to be born in the early 1970s, work began on true distributed systems: systems composed of a number of hosts on a local area network on which processes could run and communicate without much regard to host boundaries. Perhaps the first of these was the Distributed Computing System (DCS) [9], whose file system was distributed among various hosts on a network [8]. The symmetrical organization of a distributed file system contrasts with the client-server asymmetry of remote file access. (Although in both kinds of systems, one host can be both client and server, in a distributed file system the distinction between local and remote file service isn't as important.)

2.2 Origins of Leaf

Our story now moves to Xerox Palo Alto Research Center (PARC). By 1975, PARC had developed both the Alto [30], one of the first personal workstations, and the Ethernet [13], a high-bandwidth, low-latency local area network. The combination of these new technologies made it feasible to experiment with workstations manipulating files stored on server hosts.

The first such experiment at PARC was the Woodstock File System (WFS) [27], designed in 1975. WFS allowed clients to read and write arbitrary pages of files, create and delete files, and lock files against simultaneous writes. WFS also provided access to properties of both files and individual pages; properties included system-maintained information such as timestamps, as well as client-defined properties. Clients invoked WFS through unreliable **Pup** [4] datagrams; client code was expected to retry

an operation until a response packet was received from the server.

WFS was designed to be simple and reliable. Operations were atomic, and WFS retained no state between operations other than what it stored in the file system. The communication protocol was connectionless, each packet sent and acknowledged independently. There was no directory mechanism; files were identified by UIDs rather than by names. Finally, there was no protection mechanism; any client had full access to all files.

In 1977, a new file system was written for the Alto. This file system, called IFS (Interim File System, although it is still in use almost a decade later [5]), is similar to many timesharing system file systems; users are authenticated using passwords, and files are protected against unwanted access. IFS includes a directory system; files are identified by names rather than low-level identifiers.

IFS supports FTP but did not originally support remote random access to files. The WFS protocol was thus reimplemented on the IFS server, as WIFS, providing WFS clients the ability to store their files on the IFS. However, because WFS did not support file names or protection, WIFS files were disjoint from normal IFS files; WIFS and IFS users could not share files.

In addition to its lack of naming and protection, the WFS protocol had efficiency problems. Because each request was independent, the system could not optimize disk transfers of many contiguous pages. Because each network packet was independent, WFS could not avoid the use of one acknowledgement packet for each data packet, and could not transmit multiple packets before stopping to wait for an acknowledgement. WFS used sequence numbers to avoid acting on delayed duplicate packets, but did not use them otherwise.

As a response to the problems with WIFS and the WFS protocol, Dave Boggs of PARC designed the Page-Level Access Protocol (PLAP), but it was never implemented. In 1978, Steve Butterfield of the Xerox Advanced Systems Division (ASD), driven by ASD's need for a file access protocol and inspired by the PLAP, designed the Leaf and Sequin protocols. Leaf turned out to be a desirable middle ground between the simplicity of the PLAP and other, more complex designs created at PARC during the same period. ASD had practical needs and Leaf was a practical solution.

Curiously, although Leaf was a novel protocol at the time, and was used by a number of Xerox applications, not only was nothing published about it but Xerox did not write down a Leaf specification for internal use. Some of us at Stanford University thought Leaf might be useful; Xerox had donated an IFS and a number of Altos to Stanford, and we had implemented most of the Pup protocols for Unix and TOPS-20 hosts. Xerox offered to describe Leaf to us if we would write the specification. In February of 1981, Brian Reid and I spent a day with Ted Wobber, then with Xerox Systems Development Division (SDD), who provided me then and in subsequent discussions with enough information to write down a specification.

2.3 Implementations and Uses of Leaf

The first implementation of a Leaf server, and the only one done within Xerox, was written for the IFS by Steve Butterfield in 1979. Once the server was running, Ted Wobber took it over and maintained and improved it for a couple of years. Because no written specification existed, the IFS implementation served to define the protocol.

A number of different applications within Xerox included a Leaf client. These client implementations were usually written from scratch (based on folklore and reading the code of other implementations) because of the different programming languages and environments used for the applications. The Grapevine system [3] used Leaf to maintain log files; the Laurel mail-reading program, which ran locally on users' workstations, allowed users to store their old messages remotely via Leaf. Users of InterLisp-D workstations [34] could access remote files using Leaf; some versions of SmallTalk [11] had a similar ability. There a few other minor Leaf applications.

At Stanford, once the specification was available, several different Leaf implementations were started. The most involved work was done for Unix, in 1981, by a group of graduate students: Frank Boyle, Doug Hartman, John R. Craig, and Eric Aubery (many other people at Stanford, Xerox, and USC-ISI improved the code over the following years). The first implementation was a client package that could be called from Unix programs; it presented a bare interface to the Leaf operations. This client package was ported to the MC68000, and in 1982, Mike Nielsen wrote code to emulate the Unix file system calls using Leaf. By using this library, one could port many Unix programs to diskless workstations without major modifications.

Nielsen used Leaf through this mechanism to provide a preliminary file system for a multiprocessor operating system [20]. Remote file access was entirely transparent to the user, except for access to file properties (see section 4.4). Although he ul-

timately implemented a local disk-based file system, he continued to use Leaf for disk backups and for moving files between the local disk and remote file systems.

Also in 1981, the first version of a Leaf server was written for Unix. Because Leaf embodies features specific to the IFS, it is not easy to implement it exactly on top of the Unix file system. The Unix server does not support any file locking; originally, it did not support file version numbers or leader-page access, since Unix has neither versions nor leader pages. However, the most important client for this server turned out to be Xerox InterLisp-D workstations, whose users wanted full IFS compatibility. In 1982, Craig Rogers at USC-ISI wrote code to emulate leader page reads by deducing what fields the client was trying to read and extracting the appropriate data out of the Unix inode structure; leader page writes were ignored. Later that year, Gabriel Robins at IS1 modified the server to encode IFS version numbers in Unix file names. Judging from the occasional complaints about the Unix Leaf server, it is still widely used at this writing, by sites both at Stanford and elsewhere.

By the end of 1981, Eric Schoen of Stanford's SUMEX-AIM group had implemented a Leaf server for TOPS-20. This implementation was sufficiently complete to support InterLisp-D workstations, and was used by several sites outside of Stanford. It too is still in use at Stanford, at least.

2.4 Subsequent Protocols

As one of the first general-purpose file access protocols implemented, Leaf was a novelty. In that respect, it was long overdue by its introduction in 1979. Being overdue, it was almost out-of-date; the state of the art in systems design had passed it by.

One problem with Leaf is that it was based on Pup, a proprietary protocol that has become nearly obsolete even within Xerox. To demonstrate that Leaf was-sufficiently independent of Pup that it could be used with other transport layers, Frank Boyle designed the Reliable Internet Protocol (RIP), to replace the Sequin protocol used by Leaf (see section 3 for more on the relationship between Leaf and Sequin). RIP was based on the Internet Protocol (IP) [23] instead of Pup; the Leaf protocol layer was used unchanged. A client implementation of RIP was done but it was never actually used.

Today we would not design a new protocol for file access, or indeed for any remote facility not embedded in a specific distributed system. Instead, we would define a procedural interface and then use a general-purpose communication mechanism (such as Remote Procedure Call (RPC) [19] or a message-passing IPC) to invoke this interface. Some designs following this model include the CMU CFS [1] and Sesame [32] file system, Sun Microsystems' Network File System [25], and the Xerox Filing Protocol [35].

Although various RPC designs have been proposed for the ARPA Internet community, none have been adopted as a standard. Perhaps for this reason, there have not been any standards proposed for remote file access since the FAP of 1973. The Sun NFS protocol has largely filled the gap; although it is not really a general-purpose protocol, it provides nearly transparent access for Unix systems, and most workstations in the IP community are now Unix-based.

At Xerox, attention was focused less on these syntactic issues and more on semantic issues; that is, what functions a file system should provide. A central theme to these efforts was the use of transactions, which were not supported in Leaf. The Xerox Distributed File System (XDFS) also known as Juniper [14], was built at about the same time as Leaf; it too can be seen as a successor to WFS, but concentrating on WFS's features for supporting databases reliably rather than file access. XDFS, however, did not prove useful for either database or file access, and this led to the design of Alpine [5]. Alpine was meant as a replacement for both XDFS and the IFS (and hence Leaf); it was intended to run fast, support transactions, and used RPC instead of a specially-tailored protocol. Although Alpine was implemented, it has not yet entirely replaced the IFS. This may be because of the successful use of the whole-file-copy paradigm at Xerox, currently embodied in the Cedar File System [26].

The Chaosnet [17] family of protocols was developed in the late 1970s at MIT, in parallel with and almost completely uncontaminated by the work at Xerox. Chaosnet never caught on outside of MIT and Symbolics Inc., which by 198 1 had produced the Chaosnet FILE protocol [28]. This evolved into NFILE [29], which is independent of Chaosnet and can be based on any suitable byte-stream protocol. NFILE provides much the same capabilities as Leaf, but follows a sequential-access model with an operation to set the file pointer, instead of a true random-access model. Although it includes explicit data transfer operations, it appears that in normal operation NFILE is used to set up streams which are then treated by Lisp machine programs the same as streams to local files; remote data transfers are implicit. NFILE, like Leaf, uses an unsynchronized full-duplex communications mechanism, instead of a synchronous mechanism such as RPC. This creates far more complexity in NFILE than it does in Leaf: while Leaf operations carry explicit file offsets and

thus are relatively independent of each other, NFILE uses a separate operation to change the file pointer, thus requiring resynchronization before each random-access data transfer.

3 Overview of the Leaf protocol

In this section I sketch the Leaf protocol, to illustrate both its capabilities and its structure. For a more detailed specification of Leaf, see appendix I.

3.1 Layering

There are three interesting layers in the Leaf architecture. The lowest layer is the ***Pup*** intemet datagram protocol, described by Boggs et al. [4]. Pup datagrams are checksummed to detect data corruption, but are otherwise unreliable and potentially unsequenced.

The second layer is ***the Sequin*** protocol. Sequin, used only with Leaf, is a connection-based protocol that provides reliable, sequenced, duplicate-free, full-duplex packet transmission. Sequin supports multi-packet windows, so that bulk data transfer can be done without individual acknowledgement of every packet. Sequin, in effect, cleans up the Pup datagram abstraction so that a Leaf client and server can ignore the inherently unreliability of the lower layers. This results in two advantages over the unreliable datagram transport used by Leaf's predecessor, WFS:

- Reliability mechanisms are centralized in one piece of code, instead of being replicated in the client code for every operation.
- Sequin's multi-packet windows reduce the need for acknowledgement packets when reading large amounts of data.

The final layer is the Leaf protocol itself. All Leaf operations involve a single request message and one or more reply messages. Each Leaf message is contained in one Sequin datagram. Leaf is entirely client-synchronous; the server may not initiate any operations.

The layering between Sequin and Leaf is relatively clean, with one exception. Leaf locks open files against conflicting access; these locks may be broken if the server believes the client has failed. Failure detection is done using a timeout mechanism, and since Leaf is client-synchronous, an idle connection is maintained in the Sequin layer. The choice of timeout interval may depend on the particular application, so a Leaf operation allows a client to set the server's Sequin timeout interval.

3.2 Sequin Layer

Sequin is symmetrical; the client and server follow the same protocol. However, client implementations are not expected to receive, and need not support, requests for opening connections.

Each Sequin packet is transmitted as exactly one Pup datagram. A Sequin packet contains four header fields; these are overlaid on a field of the Pup header, blurring the boundary between these two layers. The Sequin header contains both received and send sequence numbers; this "piggy-backing" of acknowledgements on data packets makes separate acknowledgement packets unnecessary. Another header field indicates the buffering available at the receiver, thus providing simple flow control.

The final header field specifies the function of a packet within the Sequin protocol. Most packets are data, to be passed up to the Leaf layer. Other packet types control connection creation, maintenance, and destruction; connections are closed using a 3-way handshake to avoid half-closed connections.

3.3 Leaf data abstractions

Leaf views files as arrays of bytes rather than pages. This makes Leaf useful in environments with a variety of page sizes. Since Leaf operations are not constrained to start on a page boundary or transfer complete pages of data, Leaf clients are independent of file server page sizes.

Leaf was designed for use with the Xerox IFS file system. Consequently, Leaf supports an idiosyncrasy of the IFS: read and write operations may be performed at negative byte-offsets. This is used to access the "leader page" of IFS files, which contains information about the file such as its creation date, owner, protection, type, etc. The IFS permits write access to only a few fields of the leader page.

Leaf operations have other IFS-derived idiosyncrasies, some of which are hard to emulate in a server written for another file system (such as Unix or TOPS-20). IFS stores multiple versions of files, so Leaf allows clients to specify file version numbers when opening or creating files. Leaf also supports both exclusive-writer and multiple-writers lock policies; clients are responsible for synchronizing writes in the latter case.

Files are identified by sting names, not UIDs. Leaf itself imposes no syntax on file names, although the Unix Leaf server converts between IFS and Unix file name syntaxes if it suspects that the client is using IFS syntax. A name is used only when opening a file; subsequent operations are done on a temporary

"file handle" created when the file is opened. Authentication is done at the same time; the client must supply a valid user name and password. The use of file handles means that authentication and name translation need only be done once.

Because all Leaf file operation messages specify a file handle, it is possible to access several files simultaneously over one Sequin connection; it is even possible for several users of one client host to share a Sequin connection to a server host.

3.4 Leaf operations

All Leaf operations are initiated by the client sending a packet to the server. For each operation, there is a corresponding ***Answer*** packet that is returned by the server if the operation is successful; otherwise, the server returns ***an LeafError*** packet that indicates what went wrong. Most operation packets and corresponding Answer packets carry a file handle. Since there is no other identification information in the Answer packets, the client must rely on the sequencing provided by Sequin to correctly match Answers with requests.

Leaf operations can be divided into three categories: file handle management, data transfer, and connection management. File handle management includes operations to open a file, close a file, and force the file server to flush buffered data to the disk. Files can be created as a consequence of the open operation; there is also an operation to delete a file.

The two data transfer operations read and write file data. Data transfer can start at any offset from the beginning of the file. Write requests must fit in one packet, containing up to 512 data bytes. A read operation may request an arbitrary amount of data; the server returns the data in one or more Answer packets, as necessary.

Connection management includes operations to change timeouts and maximum packet sizes. If a connection times out but neither client nor server fails, the client may reset the connection and then continue to use it.

4 Evaluation of Leaf

Leaf is a mixed success. While it was never well-known outside the Pup community, nor has it served as a model for subsequent protocols, it was and is useful for a remarkably wide range of applications. What lessons can we learn from Leaf? In this section, I look at how Leaf differs from what came before and what came later; what it got right and what it got wrong; and how well it performs.

4.1 What is different about Leaf?

A system is said to be ***network-transparent*** if it does not matter to a client program whether a resource is on the same host or a remote host. ***Transparency*** is the most important conceptual difference between the earliest file transfer protocols and recent distributed or network file systems such as LOCUS [22] and VICE [18]. True transparency exists only when remote and local resources are named in the same way, and when remote performance and reliability are no worse for remote resources than for local resources. Even without performance or name transparency, users may still benefit from ***semantic*** transparency, the ability to perform the same operations on remote objects as they can on local objects, and ***syntactic*** transparency, use of the same operations (programs or procedures) to perform these operations remotely and locally.

FTP is not transparent. At best, it can transparently support certain forms of whole-file copy operations; it cannot by itself support most file operations provided by any modem file system.

Leaf is half-way to transparency. The semantics of Leaf operations are substantially the same as those provided by a local file system, but the syntax is different. Leaf's message-based, request-reply syntax can be hidden under a layer providing a procedural abstraction, but it would be hard to see Leaf itself as syntactically equivalent to a local file system. Name transparency also is only available through the use of another layer, and in a heterogeneous environment it may not be possible to hide such differences as case-significance and version numbering.

Whether or not Leaf provides performance transparency depends heavily on how it is used. In section 4.5, we will see that Leaf performance is about an order of magnitude slower than a local disk in a well-designed file system, so it would appear that Leaf (or any similar mechanism) cannot be performance-transparent.

There are two situations where Leaf can approach performance transparency. The first is where a

powerful personal computer has, for economic reasons, a slow local disk drive or little memory for disk buffers. In this case, the speed of a high-performance disk drive attached to a server host with lots of buffer memory might more than compensate for the cost of network communication. This probably is feasible only over a local area network, because it depends on relatively low-latency communication.

The second such situation is when Leaf is used for random access. Local file system throughput is considerably less for random access than for sequential access, but this makes no difference in cost of network operations. For example, the Fujitsu M2351 "Eagle" disk drive has an average access time of about 25 milliseconds, typical of popular high-capacity drives. The time required to perform a Leaf request/reply operation over an Ethernet is comparable; in this case remote operations take about twice as long as local ones. It is easier to reduce the network communication costs than it is to reduce the disk latency time, so in principle it should be possible to make remote random access nearly indistinguishable from local random access.

Leaf makes little provision for reliability. It has a "close transaction" operation that simply guarantees all file modifications have been forced from buffer memory onto the server's disk, but there is no way to provide failure atomicity within Leaf. (To be fair, neither the IFS file system nor most timesharing systems do any better.) Leaf is also notably less secure than local access, since user passwords and file data are all transmitted in the clear over easily-tapped Ethemets.

There are other ways to do network file access that differ in spirit from Leaf. Leaf, like many of the systems mentioned in section 2.4 (for example Alpine, Sesame, the Xerox Filing Protocol), are constructed according to the client-server model. Distributed systems such as LOCUS follow an "integrated" model, in which it is not possible to identify individual servers in isolation from the rest of the operating system. There are also semi-integrated systems, such as Sun's NFS, in which the server can be separated from the Unix kernel but there is no practical way to remove the client from the kernel.

The level of integration determines how strongly the architecture is layered. Within the client-server model, one finds cleanly-layered abstractions, and usually cleanly-layered implementations. A system following the integrated model might not have a layeredimplementation. The disadvantage of layering is decreased performance; the advantage is that it encourages the construction of heterogeneous systerns, by forcing protocol designers to take advantage of nothing in the environment beyond what is provided by other layers.

4.2 What did Leaf get right?

Leaf was successful primarily because it was a good match to its environment: a loosely-coupled local network of workstations and servers without strong security requirements. More specifically, Leaf is useful because it is not tied to a particular file system, but rather is sufficiently general to be usable with a wide variety of client and server systems.

The features of Leaf that make it so generally useful include:

Random access: Most files are accessed sequentially most of the time, but random access does take place; one study found that 20% of all file opens were for non-sequential access [12], and another study found that more than 30% of data bytes were transferred non-sequentially [21]. This makes it important to maintain the distinction between a sequential access and a sequential-access ***mechanism.*** It is possible to perform efficient sequential access using a random-access mechanism; the converse is not possible. Since both random and sequential access are necessary, Leaf's random-access mechanism is clearly more general than sequential-access mechanisms such as FTP.

Random access is useful in another way: it frees file access from the stream paradigm imposed by many operating systems. It is hard to extend streams across the boundary between two different operating systems without running into inconvenient flow control, synchronization, and connection management issues. Leaf does not entirely avoid this problem, since it uses the Sequin packet-stream protocol instead of a message protocol or RPC, but the programmer's view of the Leaf operations is not forced into the stream paradigm.

Unconstrained file naming: File naming is an area where file systems differ widely. One approach is to separate directory service from file service; a client translates a file name using the directory service, then presents the resultant UID to the file service to identify the file. The advantage of this scheme is that it is possible to construct a uniform naming environment over a variety of systems. Many existing file systems, however, do not provide externally visible UIDs for files, but rather have integrated directory systems, which makes it hard to separate these functions in access protocols.

Leaf takes the simple, if non-transparent approach

of passing uninterpreted file names from the client to the serving file system. Although this means that clients do not see a uniform name syntax across a variety of servers, it also means that Leaf can be used with any file system with explicit file names or identifiers.

Leaf includes support for optional file version numbers, which are treated as part of the file name. Normally they can be omitted in the expectation that the file system defaults to the right version, but there is an ambiguity when a file is opened for writing: should a new version be created, or should the latest version be reopened for modification? Leaf's version number support allows a client to specify how to resolve this ambiguity, if the server supports versions at all.

Minimal shared state: Leaf requires a server to maintain relatively little information about clients between client operations. This simplifies both the protocol and its implementation. Although a server might maintain additional state information to improve performance, it needs-only to maintain a map between file handles and the files they refer to, as well as several bytes of Sequin connection state. In particular, the server does not store a "file pointer" indicating where in a file the next data transfer will start, nor does it record authentication information about clients. Instead, file addresses are passed explicitly with each data transfer request, and authentication information is passed each time a file is opened.

A file handle serves both to identify the file in question and record the fact that the holder of the file handle has been properly authenticated. (Leaf's limited security goals make it unnecessary to guard against stolen file handles.) The lack of additional server state means that if the server crashes and restarts, the client can continue simply by re-opening the file and obtaining a new file handle. It also means that the server need not devote much storage space to each open file; this was important because Leaf originally ran on a 16-bit host.

Simple implementation: The features listed here, including unconstrained file naming and limited server state, help to make Leaf relatively easy to implement on existing systems, both clients and servers. Although there is some complexity associated with Sequin's flow control and sequencing mechanisms, Sequin is about as simple as a reliable protocol can get, and a minimal implementation can be even simpler if it limits its window size to one packet. Above the Sequin implementation, a Leaf client requires no additional asynchrony (all Leaf operations are client-synchronous), and a Leaf server can be single-threaded if performance is not critical. Simplicity of implementation means that Leaf clients and servers can be gotten off the ground quickly; the implementations can then be refined for better performance without disrupting users.

Leaf operations are required to fit into a single packet. This is the right compromise between convenience and implementability: it is not hard to program a loop to transfer large buffers using multiple Leaf operations, but it would have complicated the Leaf protocol to make it transport larger buffers. Also, since a Leaf write operation is contained in one packet, a Leaf server need not buffer writes in memory but may immediately transfer them to disk.

4.3 Where did Leaf go wrong?

With a decade of hindsight, it is easy to identify design mistakes in Leaf. As I wrote in section 2.4, today we would design Leaf as a procedural interface for use via RPC, instead of a set of message formats for use with a special-purpose transport protocol. When Leaf was designed, however, RPC was not really an option; the design is not so much flawed as outdated.

If we accept the basic premises under which Leaf developed (the Pup protocol family, a low-security environment, semantic but not syntactic transparency), the greatest problem with Leaf is that it is a "success disaster"; it succeeded too well, so that users have come to rely on it more than the implementors wanted. Obsolete Pup and Leaf implementations must be maintained as long as there is no widely-available alternative to Leaf.

Although it was not hard to produce simple implementations of Leaf servers for timesharing systems, both the Unix and TOPS-20 servers run into difficulty when many files are open at once. This is not a flaw with the design of Leaf, but rather with the use of timesharing systems designed before networking became important. Unix, for example, assumes that a single process will need only a relatively small number of open files. The Leaf server process on Unix can handle about a dozen open files per connection before it must start temporarily closing file descriptors. Since it records the name of the file associated with a file handle, the server can later reopen the file (unless it has been renamed in the interim) but this descriptor "swapping" results in severe performance reduction.

One flaw in the Sequin protocol is that all packets to the Leaf server on a given host go to the same Pup socket. This makes it harder to demultiplex multiple connections each to its own server process; although it is possible to demultiplex packets based on the

client's address, this complicates the server implementation and might not be possible in some environments.

Another blemish in Sequin is its re-use of a Pup header field for the 4-byte Sequin header. This does not cause any operational problems, but it violates the layering aesthetic. On the other hand, it leaves four more bytes available for carrying data. Since the Pup architecture severely limits packet lengths, a Leaf operation can carry at most about 512 bytes of data.

4.4 What is missing from Leaf?

Although Leaf operations are semantically much closer than those of FTP to the operations of a local file system, several functions are notably lacking. These fall into three categories: database support, file properties, and directory enumeration.

Database support includes facilities such as record locking, update logging, and transaction support. Although Leaf's random-access mechanism is intended for use on simple databases, these advanced facilities would be hard to implement across a wide range of file servers.

A *file property* is some piece of information about a file not stored as part of its contents; other terms are "file attributes" or "file status." Although many useful programs do not need access to file properties, some programs do. For example, the *make* program [10] (used under Unix to control compilation) needs to know when source and object files were last modified. Remote access to a file is not truly transparent unless the file's properties are available as well as its contents.

Essentially all modem file systems record file properties, but each system stores a slightly different set, and there is little uniformity in how these properties are accessed. A transparent file access protocol must deal with both problems; Leaf avoids the issue entirely. In its original environment, this did not cause problems, because Leaf allows clients to read and write the "leader page" of an IFS file, where its properties are stored. This is inadequate as a general solution. For more on file properties, including a discussion of their use in heterogeneous environments, see my thesis [15].

One could argue that a file access protocol should not provide operations on directories. However, since Leaf uses file path names instead of lower-level file identifiers, it already effectively provides ''lookup,'' ''enter,'' and ''remove'' operations on directory entries. What it does not provide is a way to list the entries in a remote directory. Within the Unix model, this is not a necessary operation, since a directory is a file and can be read and listed by a program that knows its structure. In a heterogeneous environment, this confusion between the directory abstraction and its implementation is unacceptable, for it means that a client must be able to decode the directory representation of any arbitrary file system. Since Leaf doesn't tell the client what kind of file system is running on the server (and shouldn't have to), it is almost impossible for clients to decode server directories. The lack of a directory enumeration operation in Leaf seriously reduces its utility.

The TOPS-20 Leaf server implementation extended the Leaf specification with the notion of an indexable file handle, which allows the LeafOpen operation to iterate over a set of files specified by a wild-carded file name. This feature was never used; instead, the Inter-lisp-D workstation (which is the primary remaining client for the Unix and TOPS-20 servers) uses the Pup FTP protocol to perform directory enumeration. This workaround has the advantage that it will work with any server that supports both Leaf and FTP; it is not limited to a particular implementation of the Leaf server.

4.5 Leaf performance

The potential performance of the Leaf protocol, as opposed to the actual performance of a Leaf implementation, should not be significantly lower than other file access or transfer mechanisms. Leaf uses the minimum number of network packets to perform anything except file writes, given that each operation is explicitly acknowledged and that the maximum packet size is constrained by the Pup architecture. Explicit acknowledgements are a practical requirement; for most operations they return necessary information, and for file writes they allow synchronization and some assurance that a write succeeded.

Leaf's potential performance on long file writes would be improved if a write operation could transfer buffers longer than 512 bytes; this limit means that Leaf incurs the overhead of an acknowledgement packet for each buffer, even though the client might be willing to settle for a single aclmowledgement of a much longer transfer. Also, Leaf's performance on all data transfers would be nearly doubled if it could use 1500-byte packets on an Ethernet instead of the 576-byte packets required by Pup.

With these theoretical limits in mind, we can compare the performance of an implementation of Leaf to implementations of other file transfer protocols. The measurements presented here are meant to be indicative rather than exact. They were all made using clients and servers running on Vax computers with

the 4.2BSD Unix operating system, and all were conducted using a 10 Mbit/sec Ethernet.

The protocol implementations for both Leaf and the Pup Byte Stream Protocol (BSP) are entirely in user code; they make use of a special kernel mechanism called the ***packet filter*** [16]. This mechanism makes it easy to implement and modify network software, but reduces performance by up to a factor of 6. Thus, these measurements can only indicate the relative performance of Leaf and the Pup/BSP FTP implementations, not the best obtainable performance with this hardware.

Table 1 compares the performance of Leaf and Pup/BSP FTP between a pair of Vax-11/780s, and where the client is on a Vax-11/750 and the server is on a Vax-11/780. In all cases, the client is reading a file of about 100 Kbytes from the server.

Protocol	Window (packets)	Transfer rate (bytes/sec)	Relative performance
Vax-11/780 to Vax-11/780:			
Leaf	1	11659	0.61
Leaf	4	21789	1.14
Pup/BSP FTP	1	19125	1
Vax-11/780 to Vax-11/750:			
Leaf	1	10362	0.63
Leaf	4	19795	1.21
Pup/BSP FTP	1	16375	1

Table 1: Relative performance of Leaf and Pup/BSP FTP

Leaf performance clearly depends on the window size. A large window size avoids many acknowledgement packets and similarly reduces the number of context switches. On client read operations, an ideal implementation of Leaf should have about the same performance as an ideal implementation of FTP. On client write operations, the required aclmowledgement per packet places Leaf at a theoretical disadvantage to FTP.

The FTP implementation appears to be about 20% slower than the Leaf implementation even when both use a 4-packet window; this may be due to a policy difference on delayed acknowledgements. A Leaf client normally withholds acknowledgment until all the packets for a multi-packet read answer have been received and processed (the acknowledgement is piggybacked on the next operation). In the BSP protocol, acknowledgements are specifically requested by the sender when it thinks it has filled the receiver's advertised window. The receiver, which responds immediately with an acknowledgment, may not yet have been able to pass all the received data to its client and so may have to advertise a "temporarily" smaller window to the sender. Once the receiver gets behind the sender, the window may seldom open up to its full size, and the transfer will be less efficient than it could be.

(The original code in both the Unix Leaf server and client package does not correctly handle multi-packet reads. The utter absence of complaints about this strongly implies that no existing Leaf client uses this feature, and all Leaf users have therefore put up with half the potential performance.)

How is the performance of the Unix Leaf implementation affected by the implementation architecture, especially by implementation outside the Unix kernel? We can get a rough idea of by comparing Pup/BSP FTP to IP/TCP FTP, a similar protocol implemented within the 4.2BSD kernel. Table 2 shows the relative performance of these two protocols; the kernel-resident code is four to five times faster. While it is not entirely safe to extrapolate this ratio to the Leaf protocol, it is reasonable to assume that a kernel-resident Leaf implementation would be several times faster than the current Unix implementation.

Protocol	Transfer rate (bytes/sec)	Relative performance
File transfer between two Vax-11/780s:		
Pup/BSP	19125	0.20
IP/TCP	95306	1
File transfer from Vax-111780 to Vax-11/750:		
Pup/BSP	16375	0.24
IP/TCP	67558	1

Table 2: Relative performance of Pup/BSP FTP and IP/TCP FTP

To put these transfer rates into perspective, they should be compared to the rate of local disk I/O. A Vax-1 1/780 with a fast disk drive, running 4.2BSD, can read from its disk about 275 Kbytes per second; this is more than an order of magnitude faster than Leaf, and about three times as fast as IP/TCP FTP. In practice, the difference is less severe: sequential access to large files is not the dominant mode of timesharing file systems. Most files are short [21] and much of the work done by a file system is overhead, for disk allocation and name translation [12]. The added cost of remote access is much less noticeable for access to short files.

We want low latency for short operations when Leaf is used for random-access or access to short files. Since each Leaf file operation carries a file offset, the network performance of the Leaf protocol is nearly independent of whether it is used for sequential transfer or random access; we can therefore

estimate the round-trip latency for Leaf operations from bulk transfer rates.

Table 3 shows the Leaf operation rate as a function of packet size, for read operations between two Vax-11/750s. The operation rate decreases more steeply than it should with increasing packet size, perhaps because the Unix Leaf implementation requires copying the data several times.

	Buffer size (in bytes)			
	512	256	64	16
Time (seconds)	14.39	21.88	64.57	238.77
Rate (bytes/sec)	**7964**	5238	1775	480
Operations/sec	15.6	20.5	27.7	30.0
mSec/operation	64.3	48.9	36.1	33.3

Table 3: Leaf operation rate as a function of packet size

The time it takes to perform a remote operation can be divided up into disk time, the time required to perform disk I/O for the operation, and network time, the time required to transfer data across the network. Since the Leaf server averages 5 to 10 milliseconds of disk I/O per Leaf operation, the implication of table 3 is that it takes from 30 to 50 milliseconds of network time to perform a Leaf operation. As mentioned in section 4.1, typical access times for large disk drives are around 25 milliseconds. Inexpensive hard disk drives such as might be attached directly to a personal computer have access times of 50 to 100 milliseconds; floppy disk drives are even slower. Given these figures, remote random access to a high-performance disk via Leaf should be faster than local access to an inexpensive disk.

It should be noted that the measurements in this section were made under conditions of light load on both client and server processors. Performance degrades roughly in proportion to processor load, and disk I/O load on the server also has an effect.

Acknowledgements

Many people contributed to the work described in this report, and many others provided assistance in tracing the history of Leaf and related protocols. I would especially like to thank (in alphabetical order) Dave Boggs, Jeremy Dion, Keith Lantz, Jim Mitchell, Brian Reid, Mark Roberts, Eric Schoen, Dan Swinehart, David Plummer, Jon Postel, Ed Taft, and Ted Wobber. Steve Butterfield deserves special mention, not only for his help in the preparation of this report, but because without him Leaf would not exist.

References

[1] M. Accetta, G. Robertson, M. Satyanarayanan, and M. Thompson. ***The design of a network-based central file system.*** Technical Report CMU-CS-80- 134, Department of Computer Science, Carnegie-Mellon University, August, 1980.

[2] Abhay Bhushan. ***A File Transfer Protocol.*** RFC 114, Network Information Center, SRI International, April, 197 1.

[3] A. Birrell, R. Levin, R. Needham, and M. Schroeder. Grapevine: An exercise in distributed computing. ***Communications of the ACM*** 25(4):260-274, April, 1982. Presented at the 8th Symposium on Operating Systems Principles, ACM, December 1981.

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

[5] Mark R. Brown, Karen Kolling, and Edward A. Taft. The Alpine File System. *TOCS* 3(4):261-293, November, 1985.

[6] G. E. Conant and S. Wecker. DNA: An architecture for heterogeneous computer networks. In ***Proc. 3rd International Conference on Computer Communications,*** pages 618-625. International Council on Computer Communications, August, 1976.

[7] John Day. ***A Proposed File Access Protocol Specification.*** RFC 520, Network Information Center, SRI International, June, 1973.

[8] D. J. Farber and F. R. Heinrich. The structure of a distributed computing system - The distributed file system. In ***Proc. 1 st International Conference on Computer Communications,*** pages 364-370. ACM/IEEE, October, 1972.

[9] D. J. Farber, J. Feldman, F. R. Heinrich, M. D. Hopwood, K. C. Larson, D. C. Loomis, and L. A. Rowe.
The Distributed Computing System.
In *Proc. Fall COMPCON,* pages 31-34. IEEE, March, 1973.

[10] Stuart I. Feldman.
Make — A Program for Maintaining Computer Programs.
Software-Practice and Experience **9(4):255-265,** April, 1979.

[11] A. Goldberg and D. Robson.
Smalltalk-80: The Language and its Implementation.
Addison-Wesley, 1983.

[12] Christopher A. Kent.
Cache Coherence in Distributed Systems.
PhD thesis, Purdue University, August, 1986.

[13] R. M. Metcalfe and D. R. Boggs.
Ethernet: Distributed packet switching for local computer networks.
Communications of the ACM **19(7):395-404,** July, 1976.
Also CSL-75-7, Xerox Palo Alto Research Center, reprinted in **CSL-80-2.**

[14] J. G. Mitchell and J. Dion.
A comparison of two network-based file servers.
Communications of the ACM **25(4):233-245,** April, 1982.
Presented at the 8th Symposium on Operating Systems Principles, ACM, December 1981.

[15] Jeffrey C. Mogul.
Representing Information About Files.
PhD thesis, Stanford University, March, 1986.

[16] Jeffrey Mogul, Richard Rashid, and Michael Accetta.
A Facility for Rapid Prototyping of Networking Software.
1987.
In preparation.

[17] David A. Moon.
Chaosnet.
Memo 628, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1981.

[18] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith.
Andrew: A Distributed Personal Computing Environment.
Communications of the ACM **29(3): *184-20 1,*** March, 1986.

[19] Bruce J. Nelson.
Remote Procedure Call.
PhD thesis, Carnegie-Mellon University, 1981.
Also Technical Report CSL-8 l-9, Xerox Palo Alto Research Center.

[20] Michael J. K. Nielsen.
Design and performance evaluation of a multi-computer system for timesharing environments.
PhD thesis, Stanford University, 1984.

[21] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson.
A Trace-Driven Analysis of the UNIX 4.2 BSD File System.
In ***Proc. 10th Symposium on Operating Systems Principles,*** pages 15-24. ACM, December, 1985.
Published ***as Operating Systems Review*** **19(5).**

[22] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel.
LOCUS: A network transparent, high reliability distributed system.
In ***Proc. 8th Symposium on Operating Systems Principles,*** pages **169-** 177. ACM, December, 198 1.
Proceedings published ***as Operating Systems Review*** **15(5).**

[23] Jon Postel.
Internet Protocol.
RFC 791, Network Information Center, SRI International, September, 198 1.

[24] Jon Postel.
Private Communication.
1986

[25] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon.
Design and Implementation of the Sun Network Filesystem.
In ***Proc. Summer USENIX Conference,*** pages **119-130.** 1985.

[26] Michael D. Schroeder, David K. Gifford, and Roger M. Needham.
A Caching File System for A Programmer's Workstation.
In *Proc. 10th Symposium on Operating Systems Principles,* pages 25-34. ACM, December, 1985.
Published *as Operating Systems Review* 19(5).

[27] D. Swinehart, G. McDaniel, and D. Boggs.
WFS: A simple shared file system for a distributed environment.
In *Proc. 7th Symposium on Operating Systems Principles,* pages 9-17. ACM, December, 1979.

[28] *Chaosnet FILE Protocol.*
Symbolics Inc., Cambridge, Ma., 1981.

[29] *Release 6.1 Patch Notes.*
99803 1 edition, Symbolics Inc., Cambridge, Ma., 1985.

[30] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs.
Alto: A personal computer.
In D. P. Siewiorek, C. G. Bell, and A. Newell (editor), *Computer Structures: Principles and Examples,* pages 549-572. McGraw-Hill, 1982.
Also CSL-79-11, Xerox Palo Alto Research Center.

[31] Bob Thomas.
Comments on File Access Protocol.
RFC 535, Network Information Center, SRI International, July, 1973.

[32] Mary R. Thompson, Robert D. Sansom, Michael B. Jones, and Richard F. Rashid.
Sesame: The Spice file system.
Technical Report CMU-CS-85-172, Department of Computer Science, Carnegie-Mellon University, December, 1985.

[33] James E. White.
Network Specifications for USCB's Simple-Minded File System.
RFC 122, Network Information Center, SRI International, April, 197 1.

[34] *INTERLISP reference manual.*
Xerox Special Information Systems, Pasadena, California, 1983.

[35] *Xerox File Protocol.*
XNSS 108507 edition, Xerox Network Systems Institute, Palo Alto, California, 1985.
(Draft edition).

Appendix I. Specification of Leaf and Sequin Protocols

This specification was set down starting in early 1981; the last substantive changes were made on August 17, 1982. Aside from changes in formatting and minor editing, that version is here reproduced verbatim.

1.1 Introduction

Leaf is a protocol that allows ***clients*** to access remote files via ***servers.*** It is distinct from **FTP** in that, whereas FTP supports copying entire files from one host to another, Leaf supports read and write random access to files on a server. Leaf does not create copies of a file; operations are performed on the original file. The Leaf protocol is intended to provide some amount of security and reliability, and to provide for some simultaneous access to shared files.

The Leaf protocol uses the Pup-based **Sequin** protocol as a transport mechanism. Sequin is a ***connection-based, full-duplex, sequenced, duplicate-free,*** reliable packet protocol. What this means is that the two communicating parties (a client and a server) maintain some state information not explicitly contained in the packets such that every packet is guaranteed to arrive exactly once at its destination. Further, packets arrive in the order they are sent. [Note: Leaf requires a bidirectional packet stream, but not a full-duplex one.] Sequin includes a timeout mechanism that handles the problem of crashed hosts.

1.2 Sequin

Sequin is' a packet-based protocol; that is, users of Sequin send and receive packets. A Sequin packet is formatted as a Pup packet (see the Pup specification [4]), except that the PupID field of the packet is redefined to contain four bytes of Sequin-specific information (see figure I-l). The Sequin header bytes are:

Send Sequence This byte starts at 0, and is incremented by one after every packet sent that contains data. Control packets do not increment the sequence number.

Receive Sequence This byte also starts at 0, and is the Send sequence number that you are awaiting; it implicitly acknowledges the receipt of all lower sequence numbers. The Send Sequence number of a control packet must match your receive sequence number in order to be processed; that is, a control packet cannot be processed until all preceding data packets have been received.

Allocate Specifies the ***total*** buffering available at the receiver. This specifies how many unacknowledged sequence numbers can be sent. To decrease the probability of sequence errors going undetected because of wrap-around, Allocate should not be greater than about 30. This may vary over the lifetime of a connection, so each end should pay attention to its partner's Allocate byte.

Control This byte can take on a number of values, indicating the packet's function within the Sequin protocol:

- **0 = SequinData** Indicates that this packet contains data for the next higher level protocol.
- **1 = SequinAck** Acknowledges a data packet without returning more data, thus allowing the sender to release buffer space. Also acknowledges a SequinNop.
- **2 = SequinNop** Used to maintain activity on a connection, to prevent a timeout.
- **3 = SequinRestart** Means "retransmit everything for which you have not received an acknowledgement." Used, for example, if a packet is received out of order.
- **4 = SequinCheck** A check is a request for acknowledgement, so that you can see what your partner's sequence numbers are. This is useful if you are expecting a reply and either your request or your reply was dropped. If the request was dropped, resend it. If the reply was dropped, you must do a restart. [This is obsolete, and SequinNop should be used instead* .]

'Obsolete operations are described in this specification so that the actions of older implementations may be understood.

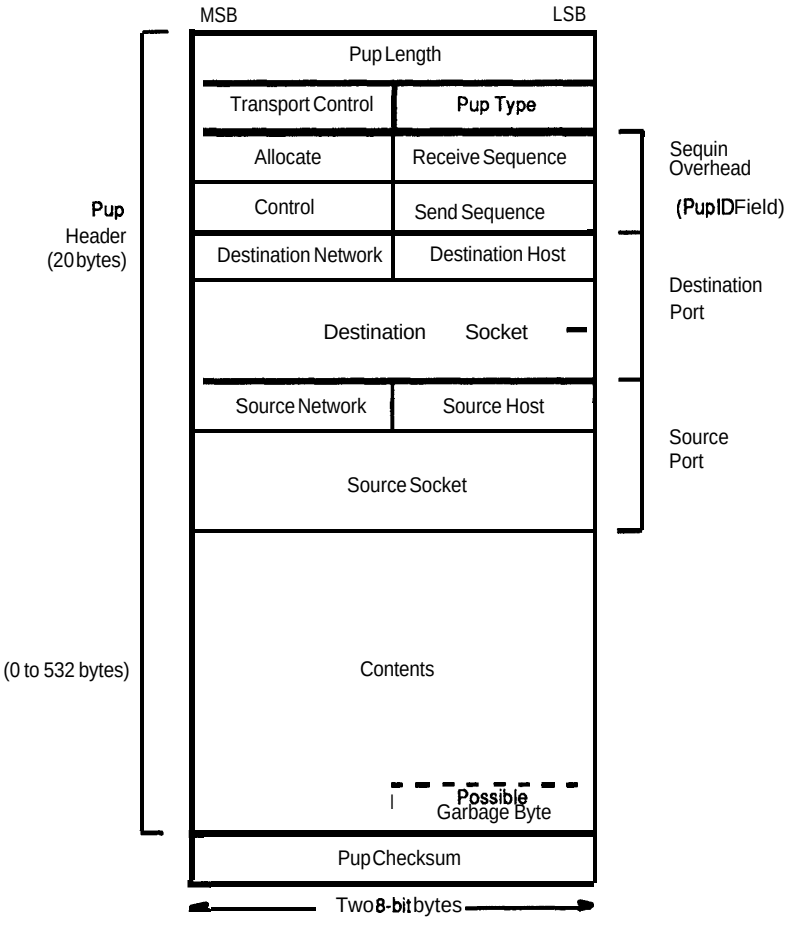

Figure I-1: Format of a Sequin packet

5 = SequinOpen Opens a connection. Both sequence numbers should be reset. This may carry data, thus it must advance the send sequence number. This is always sent from a client to a server, and the Pup Source Socket in the packet is the one on which the client will listen for all further packets (i.e., the client's Port identifies the connection.) Unlike the RTP protocol (which is ***not*** used with Sequin), the server does not return a unique socket to the client. All packets from the client to the server are directed to the server's "Well-known Socket."

6 = SequinBreak Shuts down a connection immediately; intended to indicate that a client is exiting *now*. This elicits a **SequinBroken** in response.

7 = SequinClose Used to put a full-duplex connection into a "closed" state, in preparation for a **SequinDestroy.** [Note: obsolete.]

8 = SequinClosed Acknowledges a **SequinClose.** mote: obsolete.]

9 = SequinDestroy Sent to close a connection. The receiver of this should then send a **SequinDallying,** and wait for a **SequinQuit** from the sender, timing out after a while.

10 = **SequinDallying** Response to **SequinDestroy.**

11 = **SequinQuit** Response to **SequinDallying.** Either party goes away after receiving or sending one of these.

12 = **SequinBroken** Sent to indicate that "all is **lost."** This is a courtesy, sent before the world comes to an end.

Since Sequin packets contain the Allocate and Receive Sequence numbers "piggybacked" on the outgoing data, users need not send special acknowledgement packets if there is data going in the other direction.

Since the Send and Receive Sequence numbers are represented as 8 bits, they will wrap around. For this reason, a heuristic of some sort is needed to determine the meaning of the difference between sequence numbers. For an example, see appendix II.

For simple implementations of clients, the following rules help:

1. If the Allocate byte is zero, it should be interpreted as if it were one (i.e., only one unacknowledged packet at a time.)
2. The receipt of "the latest duplicate" is taken to be an implicit restart, thus allowing a simple implementation to merely resend after a timeout.

To avoid dire consequences related to crashed hosts, Sequin includes the notion of a ***timeout.*** Unfortunately, this interacts somewhat with the Leaf protocol, because the Leaf protocol uses Sequin timeouts to implement file locks. (See section I.3 for more details.) If there is no activity on a connection for a period (10 minutes for the IFS implementation), it enters the "timed out" state. In "timed out" state, file locks can be broken, and, if the system is halting, connections can be closed. (Note that Sequin locks are broken only on demand, i.e. if another user wants to use a file while the connection is timed out.) After a further period (12 hours for the IFS), the connection is broken and the locks are released. If the client resumes activity during this "grace period," unbroken file locks are still secure.

1.3 Leaf Data Types

Leaf is based on a packet format ***known*** as a *LeafOp.* There may be one or many LeafOps sent in one Sequin packet, but a LeafOp must be totally contained within one Sequin packet, and is therefore limited by the conventional bounds on the size of the data portion of a Pup (532 bytes).

The Well-Known Pup socket for Leaf servers is 43s. Leaf/Sequin packets are always of Pup Type 260_8.

Each LeafOp begins with one word that contains a *LeafOpCode,* a flag indicating whether it is a request or an answer, and the length (in bytes) of the LeafOp, inclusive of the header word. The format is shown in figure I-2. The *Answer* flag is set if the LeafOp is an answer, and clear if it is a request.

Figure I-2: Format of first word of a LeafOp

File addresses within LeafOps are represented by *LeafAddresses*, whose format is shown in figure I-3. The fields are:

Mode This field places some restrictions on the allowable addresses for a write operation. It can take the values:

- **0 = Anywhere** No restrictions; any Leaf address is legal.
- 1 = **NoHoles** Forbids any address such that after the completion of the write, there will be a "hole" in the resulting file, i.e., a section that has never been written.
- **2 = DontExtend** On a write operation which would necessitate the extension of the file, write only as much as will fit into the unextended file. An error indication will not be returned; however, the byte count returned in response to the write will reflect the actual number of bytes written.

  When this mode is used for a read operation that attempts to read past the end of a file, an error is ***not*** returned. For all other modes, an error is returned when reading past the end of a file.
- **3 = CheckExtend** If a write operation would necessitate the extension of the file, do not start it, and report an error.

EOF If present in a write operation, indicates that this write establishes a new end-of-file position. In other words, the last byte of this write is the last byte of the file. This is useful for truncation of files.

Address The byte offset from the front of the file; the low 16 bits are in the second word, and the high 13 bits are right-justified in the first word. The interpretation of this 29-bit field is somewhat host-specific, for historical reasons:

- ***Normal Host:*** treat the 29-bit field as an unsigned integer. This supports files up to 512 Mbytes long.
- ***IFS or IFS emulation:*** The IFS implementation of Leaf allows ***negative*** byte addresses in files, used for accessing the "Leader Page" of a file. The IFS ignores the two high order address bits (bits <3:4> of the first word). If the address is greater than 2**26, then it is treated as a U-bit negative address. Note that write access is only permitted to restricted fields of the leader page (CREATE-DATE and FILE-TYPE).

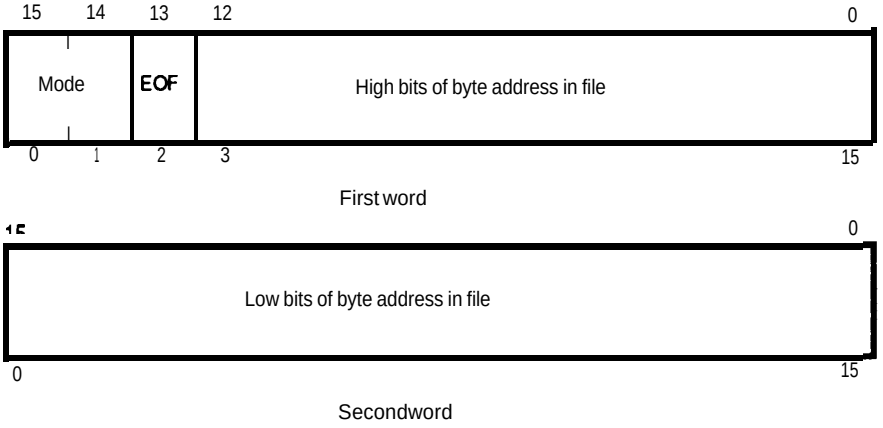

Figure I-3: Format of a LeafAddress

Strings within LeafOps are in *IfsString* format; an IfsString starts with one word containing the string length in bytes, followed by characters representing the string. Odd length strings are followed by a garbage byte to fill out the final word. (See figure I-4.)

Open files are referred to by *Filehandles*, which are 16-bit objects.

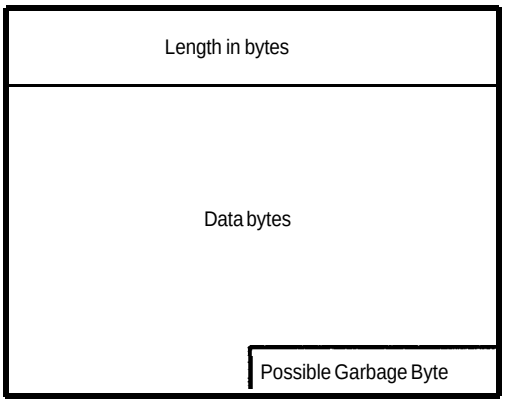

Figure I-4: Format of a IfsString

1.4 Leaf Operations

Leaf OpCode 1 = LeafOpen

This is used to open a file for further access. Note that one may have any number of open files under one Leaf connection. The format is shown in figure I-6. The filehandle field should be 0. The LeafOpenMode (shown in figure I-S) has a lot of subfields:

Read Indicates that the file will be read from.

Write Indicates that the file will be written to. It is legal for both **Read** and **Write** to be set.

Extend Indicates that the file will be extended. In general, **Extend** should be equivalent to **Write.**

Multiple If set, allows "wild card" filename specifications. This should be zero, since it is currently unimplemented by IFS/Leaf.

Create Indicates that the file should be created. Operations on non-existent files will give appropriate errors, and it is not reasonable to try to Create and Read a file without Writing it.

Explicit Version Number

This is used with the filename part of the LeafOpen. It can take on the values:

0 = None Consider it an error if there is a version number in the filename.

1 = **Old** The version number must refer to an existing version; the usual case for a read. [Not implemented in IFS/Leaf.]

2 = NextOrOld
The version number must be that of an existing file, or else the next available version number; the usual case for a write. [Not implemented in IFS/Leaf.]

3 = Any Any legal filename, with or without a version number, is allowed.

In general, use **None** or **Any.**

Version Number Default

Controls rules for determining the file version number in the absence of an explicit version number:

0 = DontDefault
Presumably, this means that there should have been an explicit version number.

1 = **Lowest** Use the lowest extant version.

2 = Highest Use the highest extant version (the current file).

3 = Next Use the next version number.

Multiple Writers When set, opens the file in "write-share" mode, allowing multiple (simultaneous) writers to a file. The only correct setting of the mode bits for opening a file in this mode is: **Extend** and **Create** clear; **Read, Write,** and **Multiple Writers** set. Multiple write-share opens are allowed, but they may not be mixed with non-write-share opens. The size of a file may not be changed while it is open in write-share mode. User processes are responsible for insuring correctness of sequences of writes.

Note: If **Create** is set or the version number default is Next (thus requesting creation of a new version), an error will result unless the LeafOpenMode specifies normal writing; i.e., **Write** is set and **Multiple Writers** is not.

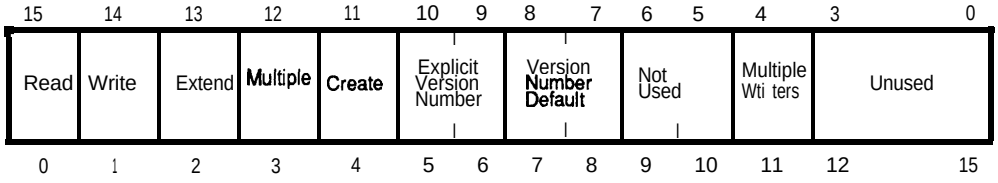

Figure I-5: Format of LeafOpenMode field

The rest of the LeafOpen LeafOp consists of five IfsStrings, in order, the usemame and password under which the file is to be accessed, the connect-directory and password, and the filename. The filename, if relative, is with respect to the connect-directory, in the usual manner.

The answer to a LeafOpen contains the filehandle to be used for further access to the open file, and a LeafAddress that represents the length of the file. There is one further word, which is unused and should be ignored.

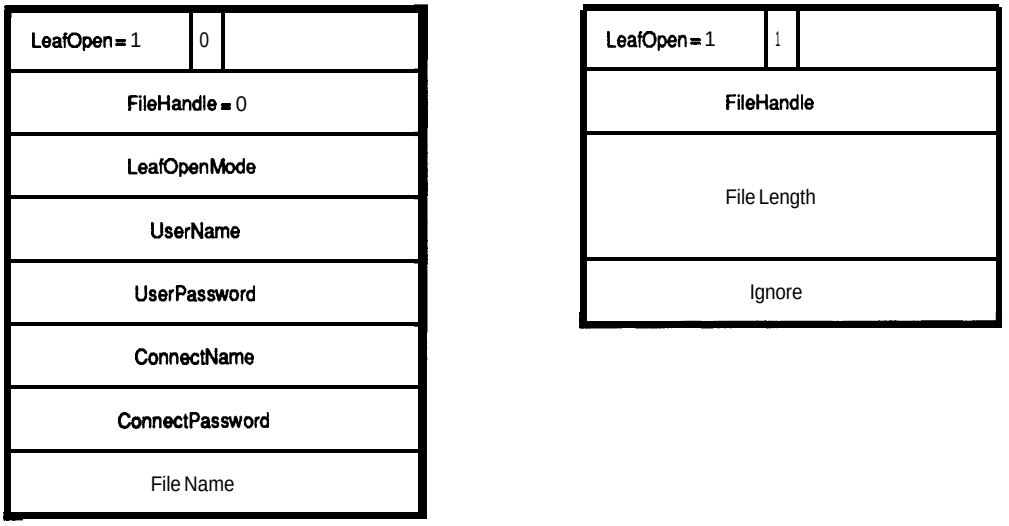

Figure I-6: Format of LeafOpen and LeafOpen Answer

Leaf OpCode 2 = LeafClose

Both LeafClose, and its expected answer, contain the filehandle of the file to be closed; see figure I-7.

Figure I-7: Format of LeafClose and LeafClose Answer

Leaf OpCode 3 = LeafDelete
LeafDelete allows a file open in "Write" mode ***(not*** "Write-share" mode) to be deleted. The LeafDelete LeafOp contains the filehandle of the file to be deleted; the Answer is identical, but has the Answer bit set. (The format of LeafDelete and its answer is the same as for LeafClose, except for the OpCode field.) **warning: Do not try to delete a file open only for reading.]**

Leaf OpCode 4 = LeafCloseTransaction
This is similar in all respects to LeafClose except that the specified file handle is not destroyed (and may thus be used in further operations). In particular, the request/answer format is identical to that for LeafClose, except for the LeafOpCode. The effect of LeafCloseTransaction is to cause the server to write to disk all buffered information relevant to the specified file handle, including all outstanding writes and changes in file length. Thus, it is useful when a client wishes to insure that changes to a file have been committed to stable storage; this is not, however, a true ' 'atomic transaction.' '

LeafCloseTransaction has no effect on files opened for reading only. Excessive use of this operation may lead to inefficiency, since it causes cached data to be flushed.

Leaf OpCode 5 = LeafTruncate
This is an obsolete operation. A file can be truncated by doing a zero-length write with the EOF bit set in the LeafAddress.

Leaf OpCode 6 = LeafRead
This is used to read from a file. The LeafOp contains the relevant filehandle, the LeafAddress of the first byte to read, and the length (in bytes) of the data to be read; see figure I-8. If the length of a read cannot be contained in one LeafOp, several LeafRead answers may be returned. Each answer to a LeafRead contains the filehandle, the LeafAddress of the first byte returned in this packet, the number of bytes remaining to be read inclusive of those in this packet, and the data itself (filled, if necessary, with a garbage byte.) (E.g., for a 2400, byte LeafRead, the lengths returned would be 2400_8, 1400_8, and 400_8. The maximum number of data bytes in a LeafRead Answer is 1000_8 [512_{10}].)

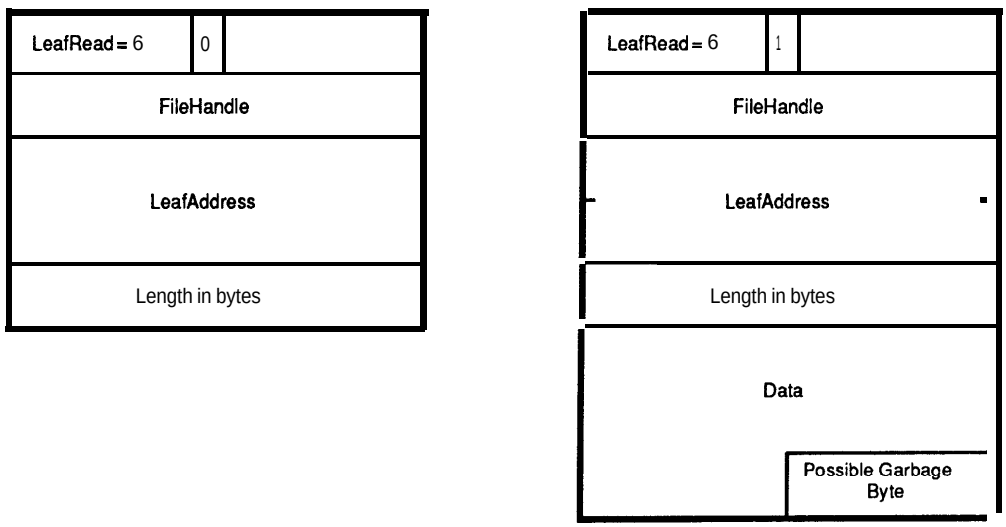

Figure I-8: Format of LeafRead and LeafRead Answer

Leaf OpCode 7 = LeafWrite
This is used to write to a file. The LeafOp contains the filehandle, starting LeafAddress, the length in bytes, and the data; see figure I-9. The answer contains the filehandle, starting address, and the length (the length actually written, in the case of ''DontExtend'' L&Addresses.) A LeafWrite must be contained within a single Sequin packet.

Leaf OpCode 8 = LeafReset
If a Leaf connection is "Broken" (i.e., a lock has been broken), it must be ***Reset*** before any further operations can

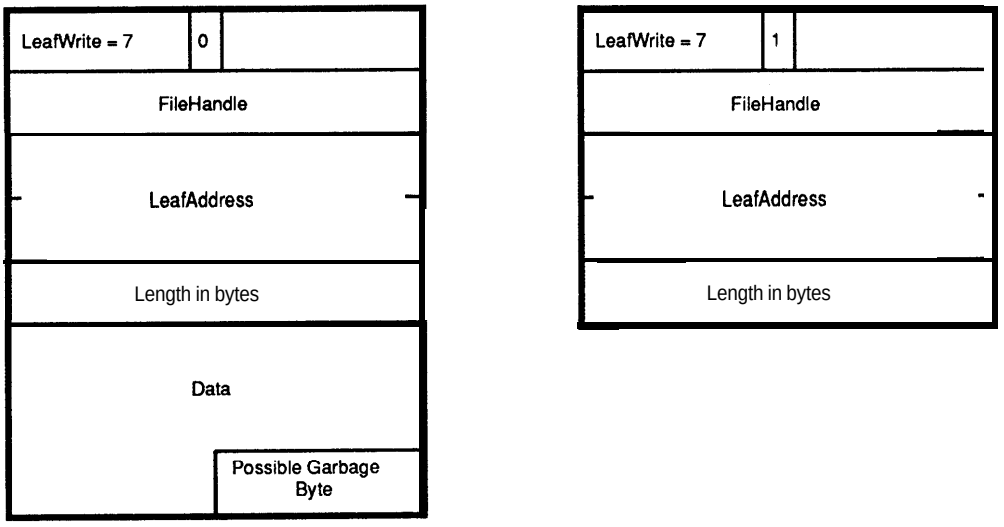

Figure I-9: Format of **LeafWrite** and **LeafWrite** Answer

take place (otherwise, these operations will return the "**BrokenLeaf**" error code). The **LeafReset** contains (see figure I-10) a **ResetHosts** word which should be one of three values:

0 Reset the connections from this host; this should not be used lightly from a multi-user system. [**Note** that Leaf was designed for use on personal computers.]

1 Reset this connection.

-1 Reset connections from all hosts under this usemame; this should not be used lightly!

The rest of the **LeafReset LeafOp** contains two **IfsStrings,** the usemame and the password used when **ResetHosts** = -1. The expected response is a **LeafReset** Answer, whose second word is meaningless. A **LeafReset** with **ResetHosts** = 1 on an unbroken connection is a No-op.

Figure I-10: Format of **LeafReset** and **LeafReset** Answer

Leaf OpCode 9 = LeafNoOp

This is obsolete.

Leaf OpCode 11 = LeafParams

This is used to set various parameters for the Sequin connection; as such, it violates the ideal of a clean separation of layers. It is a variable-length request, containing one to three data words (see figure I-l 1):

MaxPupData Sets the maximum Pup data size (in bytes) for all future Pups on this connection; the default (and maximum) is 532; the minimum is 10.

FileTimeout Sets the "file lock" timeout in units of 5 seconds; the default is 10 minutes. The server need not allow the lock timeout to be raised above the default value; if this is attempted, the timeout will be set to the default, and no error will result.

ConnectionTimeout
Sets the Sequin connection timeout in units of 5 seconds; the default is 12 hours.
A zero value for any of the parameters implies that the system default should be used.

The response to a LeafParams includes one word of data, whose purpose is currently undefined.

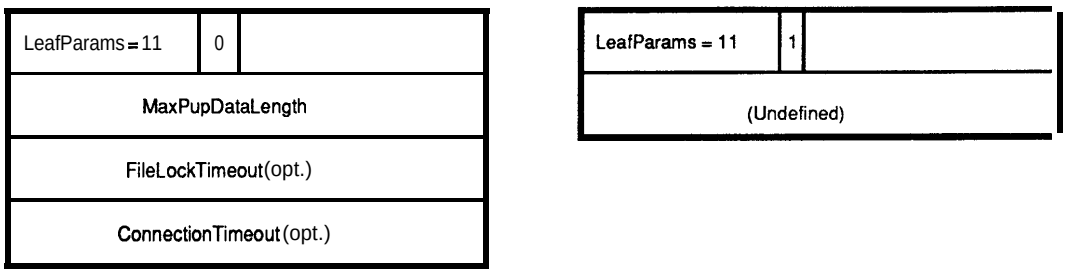

Figure I-11: Format of LeafParams

Leaf OpCode 0 = LeafError
This LeafOp indicates that an error has occurred. The LeafError LeafOp contains an error sub-code, and echos the LeafOpCode and filehandle from the offending LeafOp; see figure I-12. The error subcodes are:

SubCode	SubCode Name	Reason
116	IllegalLookupControl	"Multiple" bit set in LeafOpen mode
201	NameMalformed	illegal filename
202	IllegalChar	illegal character in filename
203	IllegalStar	illegal use of "*"
204	IllegalVersion	illegal version number
205	NameTooLong	
206	IllegalDIFAccess	not allowed to access Directory Information File
207	FileNotFound	
208	AccessDenied	file protection violation
209	FileBusy	file already open in a conflicting way
210	DirNotFound	no such directory
211	AllocExceeded	disk page allocation exceeded
212	FileSystemFull	
213	CreateStreamFailed	probably disk error in file
214	FileAlreadyExists	rename "to" file already exists
215	FileUndeletable	
216	Usemame	failures from login or connect
217	Userpassword	failures from login or connect
218	FilesOnly	failures from login or connect
219	ConnectName	failures from login or connect
220	ConnectPassword	failures from login or connect
1001	BrokenLeaf	file lock timeout has occurred
1010	BuddingLeaf	unimplemented Leaf Op
1011	BadHandle	bad file handle presented
1012	LeafFileTooLong	
1013	IllegalLeafTruncate	
1014	AllocLeafVMem	semi-fatal IFS filesystem error
1015	IllegalLeafRead	
1016	IllegalLeafWrite	

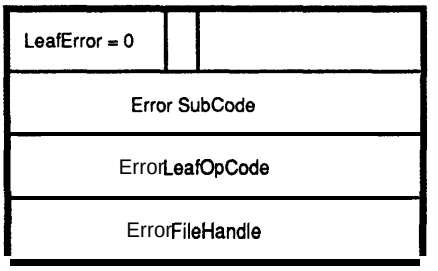

Figure I-12: Format of LeafError

1.5 Leaf Timeouts and Locks

Normally, Leaf allows one writer or multiple readers of any given file. (If the file is opened in write-share mode, multiple write-share opens are allowed.) A file is implicitly ***locked*** against conflicting access when it is opened, and access locks are checked upon subsequent opens. Locks are released when files are closed, or Leaf connections are reset.

Since a locked file cannot be accessed over other connections, if a host crashes, it might be difficult to recover. For this reason, all locks are subject ***to timeouts.*** Basically, a timeout occurs if there is no Sequin activity over a connection during a specified period. This means, among other things, that if more than one file is open under one connection, and if even one file is being accessed often enough to prevent a Sequin timeout, then the locks on all of the open files will remain secure, even if no activity occurs on the other open files. On the other hand, if a connection enters the timed-out state, and subsequently another connection opens a file open under this connection, the locks on all files will be broken. The timeouts are meant to protect against crashed connections, but not necessarily against crashed user processes communicating via these connections.

Since a Leaf server never sends unprompted packets to a client, the client must periodically send something to the server to indicate that it is still alive (a SequinNop will suffice). For the same reason, if a client accidentally times out a connection, but comes back to life after a lock has been broken, it will receive a LeafError (code = BrokenLeaf) when it tries to do any operation; it must do a LeafReset to clear this condition.

There is no notion in Leaf of a per-file lock being broken; the locks are maintained on a per-Sequin connection basis. It is not unreasonable to maintain just one open Leaf file on each of several Sequin connections; this effectively gives per-file locking.

1.6 Acknowledgements

This specification is largely based on information provided by Ted Wobber, of Xerox SDD. He has patiently answered questions and made comments on drafts of this paper, but the responsibility for errors is entirely mine. Other people who have contributed their efforts to this paper, and especially to its accuracy, are Brian Reid, Keith Lantz, Eric Schoen, and Mark Roberts.

Appendix II. Filters on Sequin Sequence Numbers

When a Sequin packet is received, the receiver must compare the Send Sequence number in the packet to its own Receive Sequence number, to determine if the packet is in the proper order. If the numbers are equal, this test has an obvious form. Unfortunately, since the sequence numbers are stored in a byte, they wrap around, and if the two numbers differ by much, it is hard to tell what this signifies.

In the IFS implementation of Sequin, the method used is as shown below (the code is a pseudo-Pascal that allows for sub-ranges in case statements; most actions are shown as comments):

```
const   SequenceMax = 377B;

type    SequenceStatus = (equal, previous, ahead, duplicate, outofrange);
        SequenceNumber = 0..SequenceMax;

function SequenceCompare(x,y:SequenceNumber):SequenceStatus;
begin
        case (x-y) of
                0: return(equal);
                -1: return(previous);
                -l00B.. -2: return(duplicate);
                1..100B: return(ahead);
                else: return(outofrange);
        end;
end;

begin
        case SequenceCompare(IncomingSendSeq, OurReceiveSeq) of
                outofrange: (* Sequin is broken, abort connection *);
                ahead: (* Request retransmission of unacked packets *);
                duplicate: (* Drop packet *);
                previous: (* Treat as if incoming control
                                were SequinRestart, then fall through *);
                equal: (* fall through *)
        end;
        case SequenceCompare(IncomingRecSeq, PreviousIncomingRecSeq) of
                outofrange: (* Sequin is broken, abort connection *);
                previous: (* Drop packet *);
                duplicate: (* Drop packet *);
                ahead: (* Release packets awaiting retransmission,
                                fall through *);
                equal: (* fall through *);
        end;

        PreviousIncomingRecSeq := IncomingRecSeq;

        (* process packet received *);
end.
```