Leaf and Sequin Protocols

Jeffrey Mogul [Stanford University/Computer Science Department]
July 23, 1982

Abstract: **Leaf** is a Pup-based protocol for remote random access to files. Leaf is implemented on top of **Sequin**, a reliable packet protocol. This document describes Sequin, Leaf, their interaction, the intended use of Leaf, and some implementation details.

0. Introduction

Leaf is a protocol which allows *clients* to access files across networks via *servers*. It is distinct from **FTP** in that, whereas FTP operations involve copying entire files from one place to another, Leaf involves read and write access to selected parts of a file. Instead of copying the file, it is (possibly) modified in place. The Leaf protocol is intended to provide some amount of security and reliability, and to provide for some sharing of file access.

The Leaf protocol uses, as a transport mechanism, the Pup-based **Sequin** protocol. Sequin is a *connection-based, full-duplex, sequenced, duplicate-free*, reliable packet protocol. What this means is the two parties (a client and a server) communicating via Sequin maintain some state information not explicitly contained in the packets to provide a system whereby every packet sent by a user of Sequin 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. 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, except that the PupID field of the packet is redefined to contain four bytes of Sequin-specific information [see fig. 1.] The Sequin header bytes are:

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

Receive Sequence - This byte also starts at zero, and is the Send sequence number that you are awaiting. The Send Sequence number of a control packet must match this number in order to be processed.

Allocate - specifies the *total* buffering available at the reciever. 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 functon 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, e.g., 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.]

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.

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

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

11 = **SequinQuit** - Response to SequinDallying. Both parties go 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 stored in bytes, they will wrap around fairly often. 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 A.

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 2 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 timeout 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 the grace period'', unbroken locks are still secure.

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.

2. Leaf

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 43B. Leaf/Sequin packets are always of Pup Type 260B.

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 2. The *Answer* flag is set if the LeafOp is an answer, and clear if it is a request.

File addresses within LeafOps are represented by *LeafAddresses*, whose format is shown in figure 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 which 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 which 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: The IFS implementation of Leaf allows *negative* byte addresses in files, used for accessing the Leader Page'' of a file. **[This is a kludge, and should be replaced by LeafOps that manipulates file properties.]** 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 negative address. Note that write access is only permitted to restricted fields of the leader page (CREATE-DATE and FILE-TYPE).

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 5.)

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

The LeafOps are depicted in figures 6 *n*:

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 filehandle should be 0. The LeafOpenMode (shown in figure 4) 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.

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: [In general, use None or Any.]

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.

Version Number Default - Controls rules for determining the file version nmber 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 nromal writing; i.e., write'' is set and Multiple writers'' is not.

The rest of the LeafOpen LeafOp consists of five IfsStrings, in order, the username and password under which the file is to be accessed, the connect-directory and password, and the filename. The filename 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 should be ignored.

2 = LeafClose
Both LeafClose, and its expected answer, contain the filehandle of the file to be closed.

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. **[Warning: Do not try to delete a file open only for reading.]**

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. However, excessive use of this operation may lead to inefficiency, since it causes cached data to be flushed.

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.

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. 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(8) byte LeafRead, the lengths returned would be 2400, 1400, and 400. The maximum number of data bytes in a LeafRead Answer is 1000(8).)

7 = LeafWrite
This is used to write to a file. The LeafOp contains the filehandle, starting LeafAddress, the length in bytes, and the data. The answer contains the filehandle, starting address, and the length (the length actually written, in the case of DontExtend LeafAddresses). A LeafWrite must be contained within a single Sequin packet.

8 = LeafReset
If a Leaf connection is Broken'' (i.e., a lock has been broken), it must be *Reset* before any further operations can take place (otherwise, these operations will return the BrokenLeaf'' error code.) The LeafReset contains 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.

1 - Reset a broken'' Leaf connection. Once an error occurs on a connection, it cannot be used until it is reset.

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

The rest of the LeafReset LeafOp contains two IfsStrings, the username and the password under which the connection is to be opened. 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.

9 = LeafNoOp
This is obsolete.

11 = LeafParams
This is used to set varous 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:

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.

0 = LeafError
This LeafOp indicates that an error has occured. The LeafError LeafOp contains an error sub-code, and echos the LeafOpCode and filehandle from the offending LeafOp. The error subcodes are:

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	Username	failures from login or connect
217	Userpassword	" "
218	FilesOnly	" "
219	ConnectName	" "
220	ConnectPassword	" "
1001	BrokenLeaf	file lock timeout has occured
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	

Leaf Timeouts and Locks

Normally, Leaf allows one writer or (optionally, and) multiple readers of any given file. A file is *locked* against conflicting access when it is opened, and access locks are checked upon subsequent opens. Similarly, 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 wll 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; the locks are maintained on a per-Sequin connection basis. However, it is not unreasonable to maintain just one open Leaf file on each of several Sequin connections; this effectively gives per-file locking.

Acknowledgements

This paper 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, Eric Schoen, and Mark Roberts.

Appendix A: 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.):

```
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);
      -100B..-2: return(duplicate);
      1..100B: return(ahead);
      else: return(outofrange);
   end;
end;

begin
   case SequenceCompare(IncomingSendSeq, ExpectedSendSeq) of
      outofrange: (* Sequin is broken, abort connection *);
      ahead: (* Request retransmission of unacked packets *);
      duplicate: (* Drop packet *);
      previous: (* Change Incoming control to SequinRestart,
                     then fall through *);
      equal: (* fall through *)
   end;
   case SequenceCompare(IncomingRecSeq, AckedSeq) of
      outofrange: (* Sequin is broken, abort connection *);
      previous: (* Drop packet *);
      duplicate: (* Drop packet *);
      ahead: (* Release packets awaiting retransmission, fall through *);
      equal: (* fall through *);
   end;

   AckedSeq := IncomingRecSeq;

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

Appendix B: Enabling the IFS Leaf Server

There are 2 steps in this process. The first is to invoke IFS with the /L switch. This does *not* enable Leaf, but an IFS started in this manner should display an L'' after its version number. The second step is to chat to the IFS and issue an Enable'' command for Leaf. If Leaf is enabled, the statistics (for) servers'' command will show statistics for it.

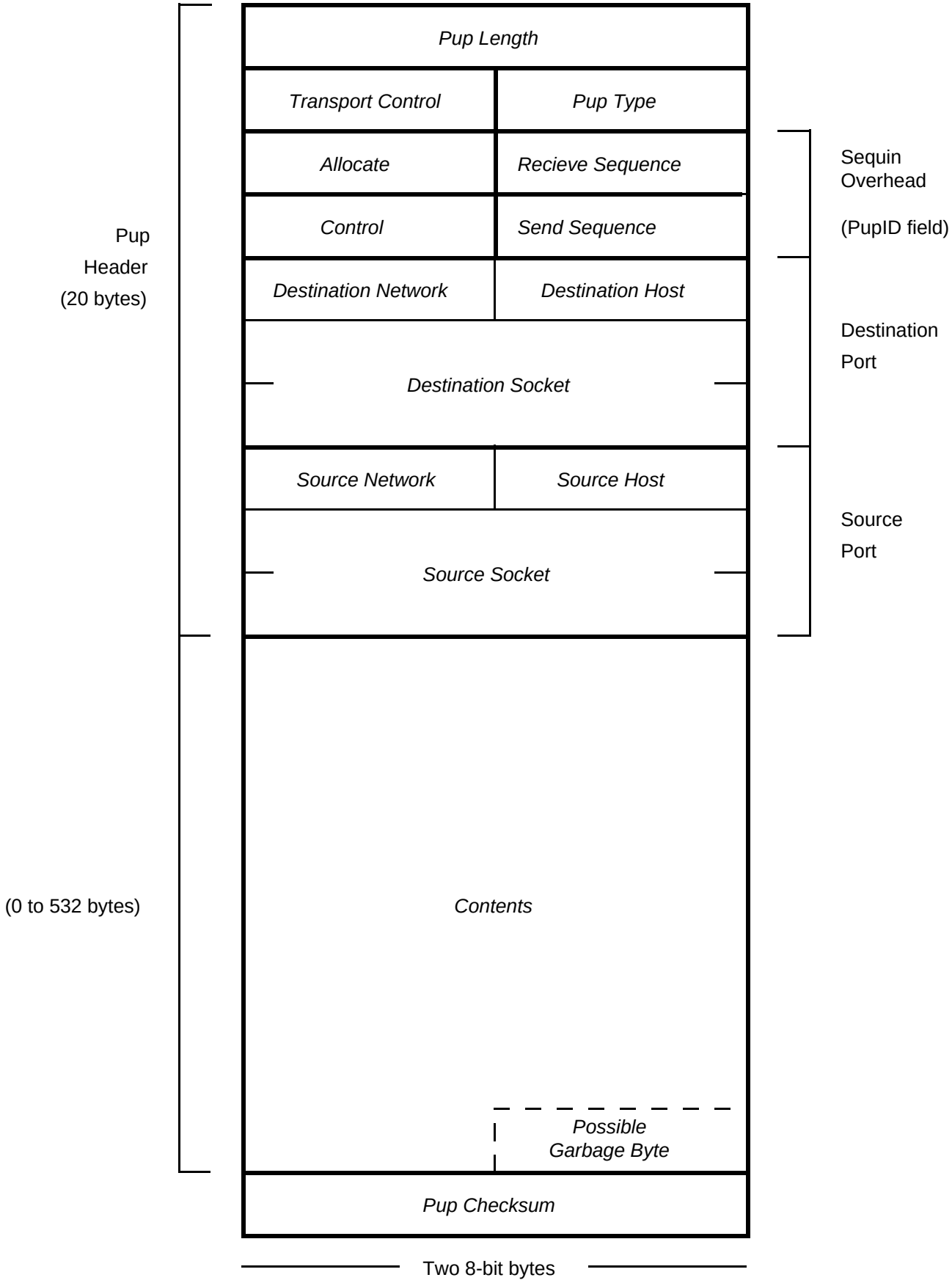

Figure 1: The format of a Sequin Packet

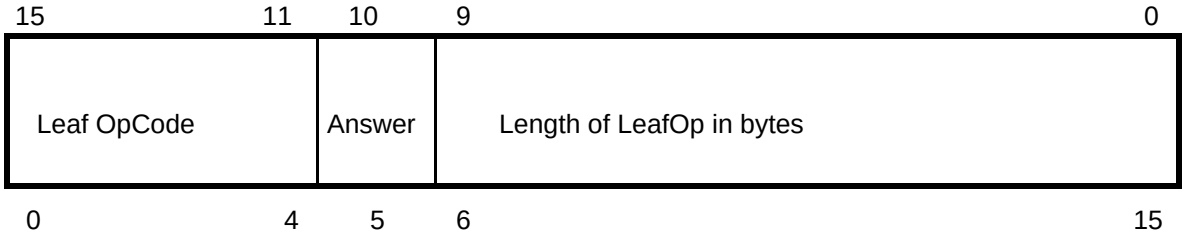

Figure 2: Format of first word of a LeafOp

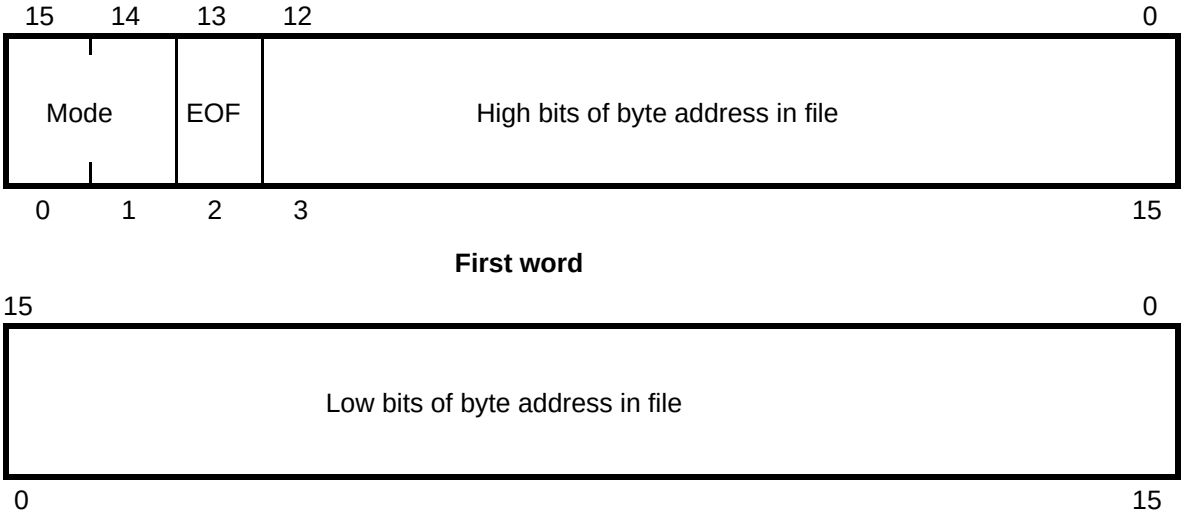

Figure 3: Format of LeafAddress

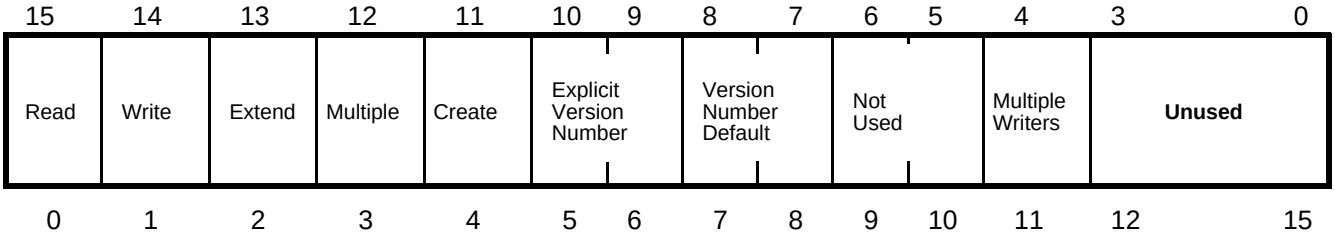

Figure 4: Format of LeafOpenMode

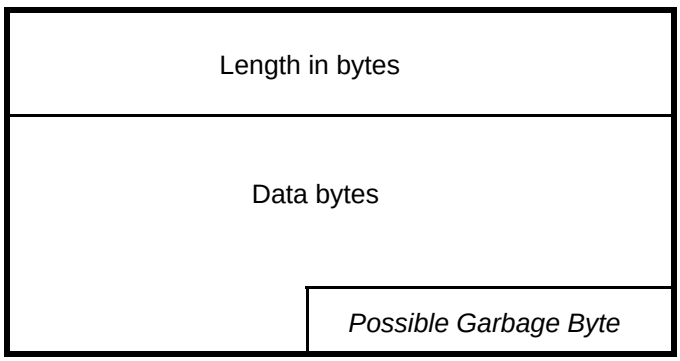

Figure 5: Format of an IFSstring

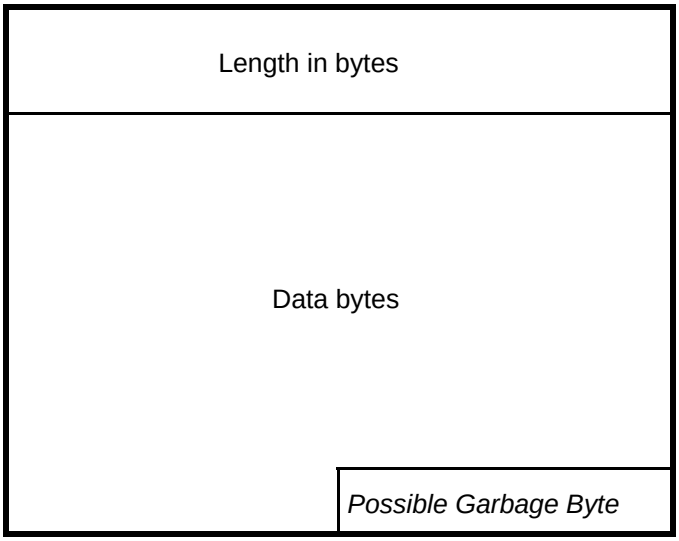

Figure 5: format of an IfsString

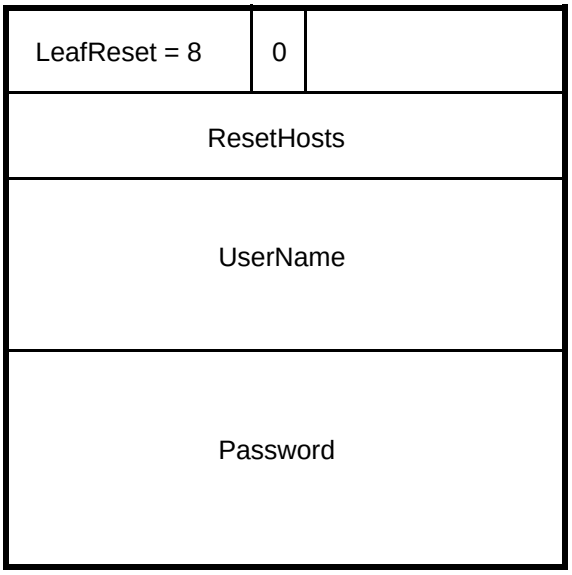

Figure 6: Format of a LeafReset

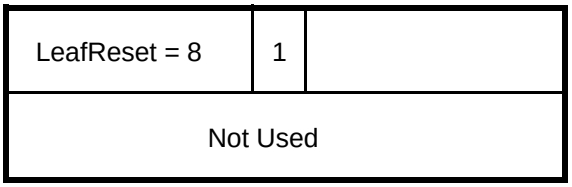

Figure 6a: Format of a LeafReset Answer

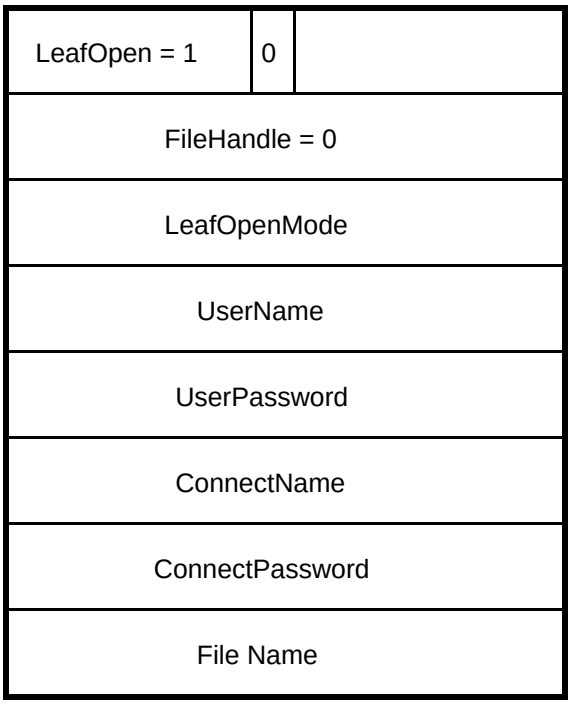

Figure 7: Format of a LeafOpen

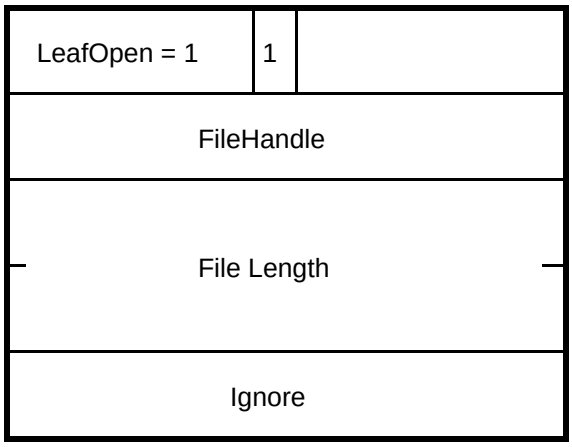

Figure 8: Format of a LeafOpen Answer

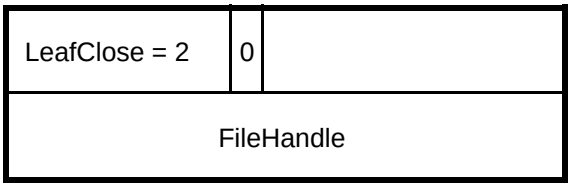

Figure 9: Format of a LeafClose

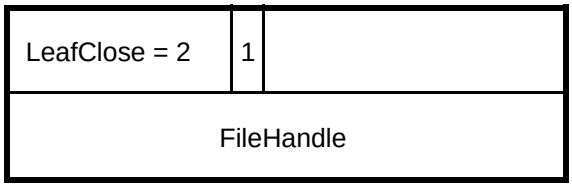

Figure 10: Format of a LeafClose Answer

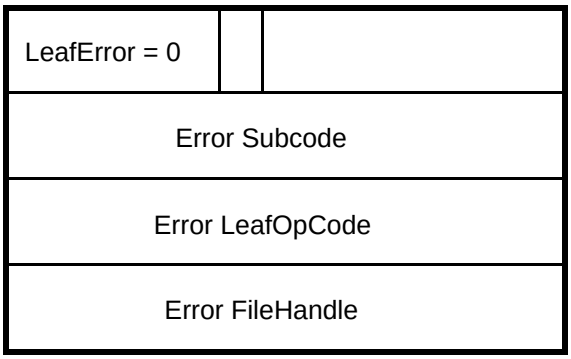

Figure 11: Format of a LeafError

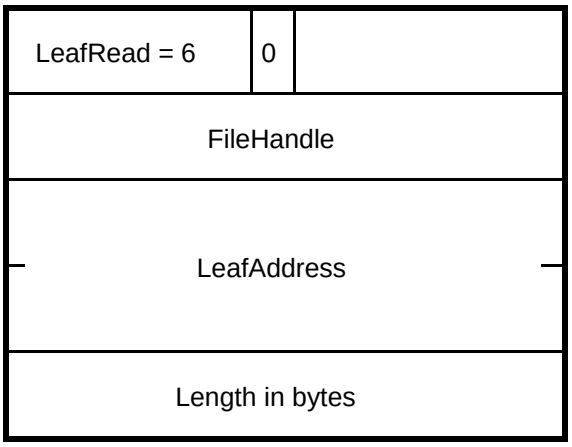

Figure 12: Format of a LeafRead

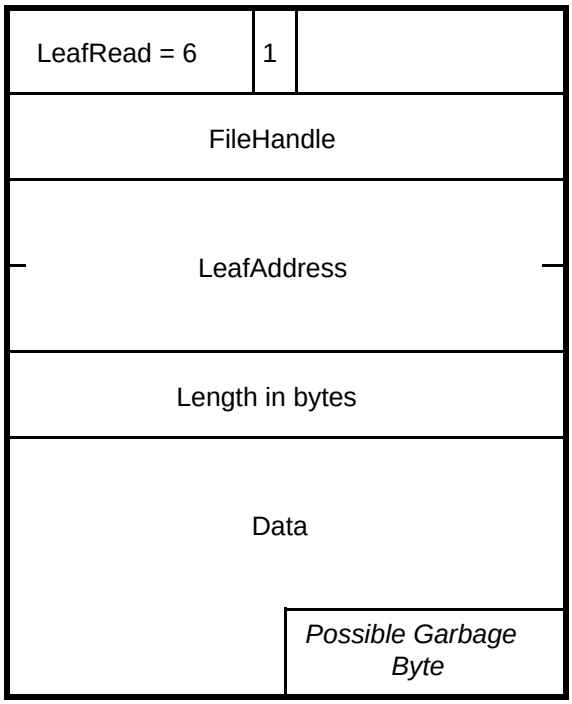

Figure 13: Format of a LeafRead Answer

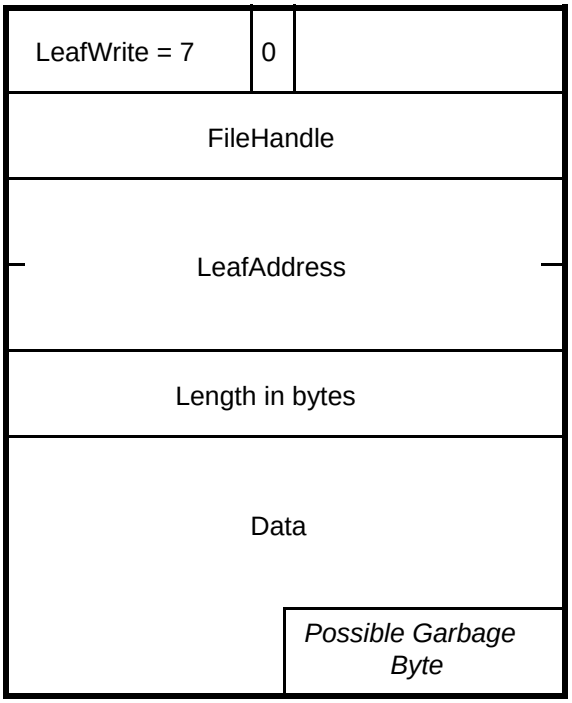

Figure 14: Format of a LeafWrite

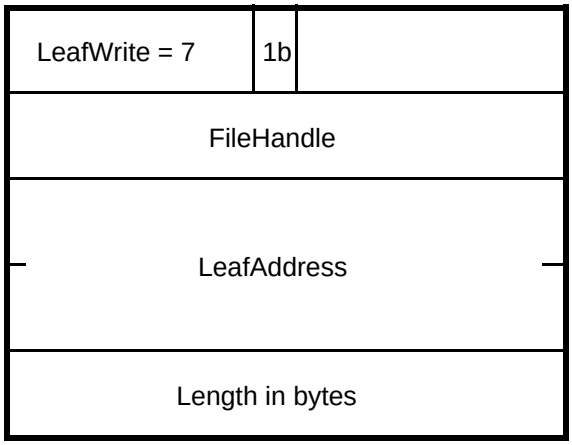

Figure 15: Format of a LeafWrite Answer

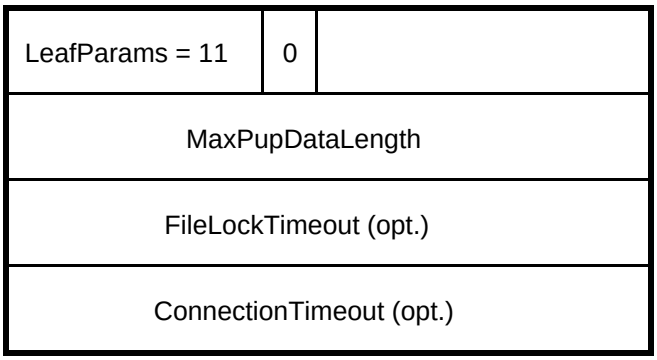

Figure 16: Format of a LeafParams

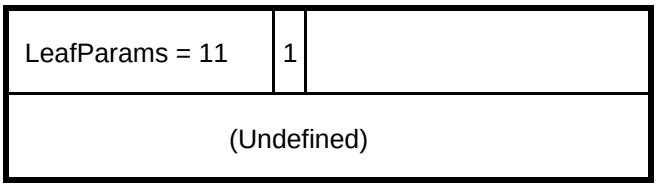

Figure 17: Format of a LeafParams Answer