Sun Network File Protocol Design Considerations

Bill Joy

Company Confidential

ABSTRACT

The Sun Network File Service uses a connectionless datagram-based remote procedure call protocol to provide shared file access. This document explains the design principles in the protocol, and the facilities provided by the File Service which is to be implemented using the protocol.

The protocol is specifically designed for use by clients who use local data and naming caches to improve performance. The protocol is optimized for this caching case, and is very efficient when used in this manner. Cache flushing and consistency checking occurs at specified points; in normal UNIX operation this is at file ***open***, ***close*** and ***fsync*** calls. This increases system performance greatly with minimal impact on applications.

Guaranteeing full distributed consistency with the protocol is also possible but quite a bit more expensive.

Introduction

This document describes the protocol used by the Sun Network File Service. A general description of the Network File System Provided by the server is to be found in the document "Design of the Sun Network File System." A description of the modifications to UNIX to support this and other file protocols can be found in "Sun UNIX Modifications to use the Sun Network File Service."

Principles

The guiding principle of the File Service is that it is stateless. Clients make requests of the service by datagrams and receive responses by datagrams. The file service caches authentication information supplied by clients, recent requests which have been completed, and the data supplied in response to recent requests. This cache often allows the File Service to avoid reexecuting requests which are resubmitted because a confirmation of the execution of the request was lost or did not arrive at the client quickly enough. The service, however, makes no guarantees to keep old responses, and reserves the right to reexecute duplicate requests.

The File Service does not guarantee that operations which are submitted to it are performed unless acknowledgements are received. Operations may fail to be performed because they are lost enroute to the server, or because the server crashes after receive them. Operations may be performed without confirmation to the client because the acknowledgement is lost or because the server (or client) crashes.

When the host supporting the File Service crashes it loses **all** the state of client requests and authentication information which was provided to it by these clients. This differs from the cache flushing which can occur during normal operation only in the amount of information which is lost. Client nodes recover just as they do when normal cache flushing occurs: by resubmitting authentication information, and by recovering from requests for which responses weren't received by resubmitting requests and processing the responses as they do whenever retransmissions are required.

Discussion

The File Service is designed to be used with caches on each client node, where clients desire to trade absolute consistency checking for much greater performance. Thus is allows read-ahead and write behind and caching of directory entrys in pathnames, and should greatly improve the performance of the system.

The File Service does not provide a transaction facility, as discussed in the Network File System design document. Given that it does not, it helps its clients maintain consistent file system state by providing the clients with a set of atomic operations. Under these conditions, it would be possible to attempt to maintain end-to-end state with each client about all the files the client had open and the exact state of each operation which the client has in progress.

The difficulty of this approach is that there are a potentially enormous number of open files in the network and a very large number of clients active working with each server. Requiring a server machine to maintain hundreds or even thousands of open files in tables could easily cause the server to waste precious space on rarely used information. After a client crash, the server would have to poll to notice that a machine had crashed to be able to reclaim the space.

The benefit for all this is very little if the server can operate without *guaranteeing* to hold all this information. Rather than holding information about all the files in use by all the clients, the server can be designed to cache recently used information and not make guarantees to keep any of it permanently. Thus the response datagram from the server to a client, which has about a .02% chance of being lost, is not acknowledged by the client, minimizing message traffic. Even between a rarely communicating client and server, a single *read* operation can involve only one message exchange, 50% less messages than any scheme where the server guarantees to retransmit any response until it is acknowledged.

We believe this approach has the potential for very high performance and can serve large number of clients with huge numbers of open files quite successfully.

Facilities provided by the File Service

The File Service maintains a hierarchical file system containing a number of files, each of which has a set of file attributes, and contains an uninterpreted array of bytes. These files are referenced by clients using low-level names supplied by the File Service to the client in response to requests.

When a client authenticates itself to a server it receives in response a *fhandle* (a low level name) for the root directory of the File Service's file system. A set of requests exist which take a *fhandle* representing a directory file in the file system and a name, and perform an operation affecting the directory: entering a link, removing a file, renaming a file, creating a file, or, most importantly, looking up the fhandle for a specified name within the directory.

The following operations on fhandle's do not involve directories:

write[fhandle, offset, length, data] => [errcode, length]
read[fhandle, offset, length] => [errcode, length, data]
writeattr[fhandle, attributes] => [errcode, attributes]
readattr[fhandle, attributes] => [errcode, attributes]

The *fhandle* structure consists of a integer *inode* number within the File Service file system, and a unique-id further identifying the *inode*, since UNIX occasionally reuses the same *inode* number for different files.

The *errcode*'s are from a small set defined by the protocol, and include

FERR_BADFHANDLE	fhandle is invalid
FERR_QUOTAXCEED	quota exceeded on write
FERR_NOPERM	insufficient permission to perform operation
FERR_IOERR	physical device error executing operation

Each of the operations also passes a protection context; in UNIX terms this is a *uid* and a set of *gid*'s. An authentication message to the server executes an authentication protocol to establish protection contexts. The authentication protocol also allows the establishment of DES keys for further datagrams so that communications can be secured. These mechanisms are part of Sun's Network Architecture and are beyond the scope of this document.

The *attributes* are an externalized form of the *vattr* structure defined for *vnode*s in the file vnode.h. See the appendix to the "UNIX Modifications" document.

The operations defined above correspond to those operations on files defined for *vnodes*, e.g. VOP_RW, VOP_READATTR, VOP_WRITEATTR. In addition the file server has operations which apply to files which are of type directory, call them *dhandle*'s:

lookup[dhandle, name, attributes] => [errcode, fhandle, attributes]
create[dhandle, name, attributes] => [errcode, fhandle]
remove[dhandle, name] => [errcode]
rename[dhandlenew, namenew, dhandleold, nameold] => [errcode]
...

The *name* argument to these calls is a counted byte string. It is used to index a single level of the directory hierarchy of the File Service.

The *lookup* operation returns a *fhandle* for a particular name, and also returns the attributes of the resulting *fhandle*.

The *create* operation makes a new entry in the hierarchical file system with the specified attributes (e.g. type, file mode, etc), returning a handle for the new file.

The *remove* operation deletes an entry from the directory structure. This causes the underlying *fhandle* to become invalid, unless there is another name in the directory hierarchy for this *fhandle*.

Issues

1. Cached translations?

We expect that client machines will use their local memory to cache data blocks and also translation triples:

[dhandle, name] => [fhandle]

Such translations are relatively safe when used in the middle part of a long path name translation provided the last component is checked with the server. This means, however, that translations where the target yields a non-directory are less safe.

We can address stale translations by doing a lot more work with the server to insure that stale translations are not used, or we can not worry about it and simply force translations to time out periodically. I propose we cache long-term on the client only directory name translations and to revalidate translations for files each time a UNIX-level operation takes place. This validation would take the form of a ***lookup*** and we will request an attribute which returns a generation-id for the file indicating the last time it was modified. This insures that at least at the point a file is reopened all stale cache buffers are flushed.

In any case it must be possible on a per-filesystem, per-file, per-directory or per-process basis (not all of these, just some probably) to request no caching.

2. Detecting server reboots to force cache flushing

The clients should also detect when a file server has rebooted since this can permit a data block in a file to change without the client being able to tell by looking at the generation number of the file in its attributes at the next ***open***. If this weren't the case then we would have to do two file i/o operations for each write to the file: one to write the new data after one to update the generation count.

We thus flush the cache completely when we detect that the server has been rebooted. Since the server forces us through an authentication message each time it reboots, we can insure this by returning the boot time of the server as a result of each authentication sequence and, if this does not correspond to our understanding of when the server was booted, we flush all our caches about the server.

3. How do we recover from lost messages?

Consider a request to make a link from ***a*** to ***b***. To be able to recover from a lost response to the ***link*** the client must conceptually perform a series of steps, e.g.:

1. Check that ***b*** does not exist, and repeat until you get a response. If ***b*** exists then exit with an error.
2. Check that ***a*** exists, getting a fhandle for it; repeat until a response is received. If there is no ***a***, then exit with an error.
3. Request that the link be made. If there is a response, then that is the answer.
4. We didn't get a response. Check to see if ***b*** is the same as ***a***; if it is, then we completed the link successfully and the response was just lost. If it exists, but is not the same, then there is an error. If it doesn't exist, then we try to create it again, starting at 3.

In the case we have local caching, then we are likely to know whether we believe that ***b*** exists, and to know what fhandle corresponds to ***a*** (from recent activity). This allows us to skip steps 1 and 2. As we normally receive a response to step 3, the entire operation can be achieved in one message exchange. If we wish to run without caching then the full set of steps must take place, with a message exchange per step. In this case, more messages are involved then would be required in a state-based protocol (6 versus 3).

4. Referencing files after they are removed?

Some UNIX programs may depend on the quirk that files can be removed from directories and still accessed even though they have no names. Such files can take up a lot of disk space and can't be reclaimed till the reference goes away. The File Service has no notion that there is a reference to such a file, and removes it when it is removed! If the program really wants its open files to never go away it should insist that they continue to have names, so that they remain referenced.

It has always been possible to reduce the length of a file that was in use to 0. This as good as makes it go away, and is what most people would expect a remove to do. The current strage behaviour is rarely supported on other operating systems, and thus it can be considered antisocial behaviour for a program to have in a heterogeneous network environment.

Such programs will need to be cleaned up.

5. Devices?

Remote devices have state. They are thus not real suitable for access using this protocol, and are the subject of another protocol.

6. Long requests?

Since we don't handle devices, the protocol can limit single read and/or write requests to some moderate length. Long write requests from clients to the server are nice, because they minimize disk transactions but they also tie up buffer space for a longer minimum period if they are to succeed.

7. Performance?

This design makes no requirements on the server other than to do write operations and other requests which modify the file system and to provide read data when requested. The client nodes are allowed to perform arbitrary amounts of caching, and the consistency of such data in the presence of parallel operations on different machines is outside the scope of the file server. If there are no such conflicts, or the clients choose not to concern themselves with the consequences of such conflicts, then all is well.

8. "World" access?

UNIX lacks the "world" access protection class that VMS has, which would be very useful to distinguish files which are to be accessible to boot programs and others which are not authenticated. I propose that the default user for such operations be such that he is not an other, but a nothing in group WORLD, so that only files which are in the WORLD group can be accessed, and the root directory and perhaps /stand and files therein would be owned by WORLD.

Implementation

The local UNIX file system has been modified so that it can be called directly by the File Service server process. This means that the File Service implementation consists of remote procedure call handling code, a lookup in the systems ***vnode*** cache which supports caching of ***fhandle***'s directly, and packaging of the results of the operation to send them back to the client node.

Remote procedure call support and support for transmission of data types in the network has been constructed by Bob Lyon as part of the support for the Sun Network System Protocols and this work should be directly usable here.

In the first implementation, servers can accept all requests from clients, ignoring authentication, and pass back the fhandle's in the clear. The authentication work that Bob Lyon and David Goldberg are doing for protocols in general will transfer directly into the File Server protocol, so need not be considered further here.