COMMON INTERNET FILE SYSTEM PROXY

CS739 PROJECT REPORT

ANURAG GUPTA, DONGQIAO LI

{anurag , dongqiao}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin, Madison
Madison – 53706, WI

May 15, 1999

1. Introduction

1.1 Server Message Block (SMB)

Server Message Block (SMB) is a protocol for sharing files, printers, serial ports, and communications abstractions such as named pipes and mail slots between computers. It is a client server, request-response protocol, except for the case when the server sends an asynchronous lock break request to the client if the client requested a lock on a resource. Clients can connect to servers using TCP/IP, NetBEUI or IPX/SPX. Once they have established a connection, clients can then send commands (SMBs) to the server that allow them to access shares, open files, read and write files and other file system activities. However, in the case of SMB, these things are done over the network.

1.2 Common Internet File System

The **Common Internet File System** (*CIFS*) **[1]** is a file sharing protocol that is the Microsoft® version of a distributed system. It is intended to provide an open cross-platform mechanism for client systems to request services (*e.g. file and printer*) from server systems over a network (*potentially the Internet*). It is based on the standard Server Message Block (*SMB*) protocol widely in use by personal computers and workstations running a wide variety of operating systems.

The CIFS protocol supports features like file access, file and record locking, safe caching, read-ahead and write-behind, file change notification, protocol version negotiation, extended attributes, distributed replicated virtual volumes, server name resolution independence, batched requests and Unicode file names.

CIFS is transport independent. It assumes a reliable connection oriented message-stream transport and makes no higher level attempts to ensure sequenced delivery of messages between the client and server. It also assumes that there exists some mechanism to detect failures of either the client or server and to deliver such an indication to the client or server so they can clean up their state. For example, CIFS

can be run over NETBIOS over TCP or just over TCP. If a CIFS server receives a transport establishment request from a client that it is already conversing with, it terminates all other transport connections with that client.

The CIFS protocol supports the following SMB protocols:

- PC NETWORK PROGRAM 1.0
- LANMAN 1.0
- LM1.2X002
- NT LM 0.12

1.3 Samba

Samba is a popular implementation of CIFS. The Samba software suite **[2]** is a collection of programs that implement the Server Message Block protocol for UNIX systems. It is an open source software suite that provides *seamless* file and print services to SMB/CIFS clients. Samba is freely available under the GNU General Public License. It is a widely adopted solution to integrate Microsoft/IBM style desktop machines or Microsoft (*etc*) servers with Unix or VMS (*etc*) servers. Samba provides the following features:

- *SMB server*: to provide Windows NT and LAN Manager-style file and print services to SMB clients such as Windows 95, Warp Server, smbfs and others

- *NetBIOS nameserver*: which among other things gives browsing support. Samba can act as the master browser on the LAN

- *ftp-like SMB client*: to access PC resources (disks and printers) from UNIX, Netware and other operating systems

- a tar extension to the client for backing up PC's

- limited command-line tool that supports some of the NT administrative functionality, which can be used on Samba, NT workstation and NT server

3. Project Overview

3.1 Objective

The objective of the project was to design and implement a Common Internet File System Proxy for multiple CIFS clients and a CIFS server with caching and locking support.

3.2 Design Approaches

3.2.1 Event Driven, Multi-threaded Proxy Model

In our proxy implementation, there is one thread (*the dispatcher*) listening for client requests on a port and one thread (*the responder*) listening to responses from the server. Each time a new request from a client comes, the dispatcher creates a new thread. This newly created thread is in charge of servicing the client request. If a request of a file is made which is in the proxy cache, the proxy starts serving the request. If the proxy does not have a cached copy of the file it forwards the request to the server and caches the file as it is being sent by the server. In this way we guarantee that no client is waiting while the proxy is servicing requests from other clients, making the proxy more scalable.

Between the proxy and the CIFS server, there is only one SMB connection. Thus, all requests from different clients need to be multiplexed by the proxy to the same proxy-server connection and all responses from the server intended for different clients must be de-multiplexed at the proxy. We achieved this by using the Multiplex Id (*mid)* field in the common SMB message header. Each time an '*Open*' request from a client comes, the dispatcher maps the *(clientfd, mid)* pair into a new *proxy_mid* which is

unique among all clients connected to the proxy and the proxy saves this mapping in an in-memory mapping table. The dispatcher also sets the *mid* field in the SMB message header and all subsequent SMB messages to this *proxy-mid* before it forwards the message to the server. When a SMB message comes in from the server, the responder looks up the client socket descriptor and client mid in its mapping table using the *proxy_mid* from the message header. De-multiplexing is achieved when the responder resets the *mid* field to *client_mid* and it then forwards the message to the corresponding client. This way the client sees the same *mid* as it had sent.

Our proxy is an Application Level Proxy, which means all the clients are proxy aware and the proxy server knows the address of the CIFS server. The client-proxy-server setup is shown Figure 1.

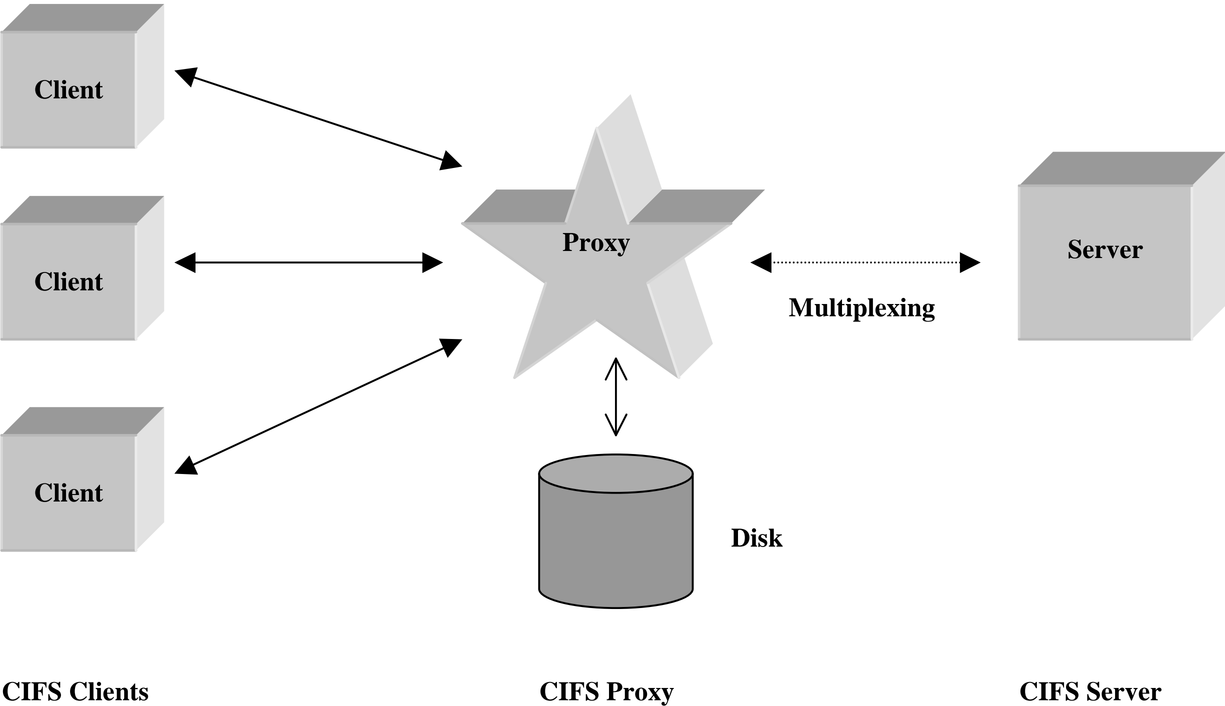

Figure 1. The Big Picture

3.2.2 Caching

Proxy caching has been proven to be an effective method for reducing network traffic and improving the response time for clients for requests to the server **[3]**. By caching common client requests for files on the proxy, a big saving can be achieved in network traffic and the response time to the client improves significantly as the resource is brought closer to the client. In our implementation, we took advantage of a disk cache on the proxy while using the Clock algorithm for cache replacement.

Cache coherence is maintained by using the *last write stamp* sent in the open response from the server on an open request from the client. The proxy caches the last write stamp sent from the server and on a subsequent request for the same file, before it starts servicing the request, it checks the last write stamp sent by the server with the time stamp that it has. If the two time stamps are the same, the proxy serves the request, otherwise it invalidates the cached copy of the file and the file is got from the server as normal.

Our proxy cache is *crash resistant*, which means if the proxy crashes or terminates for any reason, the restarted proxy will still be able to take advantage of all the files in the cache to serve client requests. This feature increases the overall hit ratio and reduces the overall caching overhead, thus improving the client perceived latency. When the proxy starts, it checks its disk cache to see if it has any previous cached files. If it does, it reads those files into its in-memory cache data table. If the file was changed at the server during the time the proxy was down, it will be invalidated by the invalidation method described above.

3.2.3 Locking

Locking is an essential part of CIFS. CIFS supports opportunistic locks, or *oplocks* for short, which allow clients to dynamically alter their buffering strategy in a consistent manner. Network performance can be greatly increased since a client does not need to inform the server immediately about

every change it makes to a file, or have to worry that other clients can make its information about the file out of date. The current version of samba client doesn't support locking. We modified the client and added in locking support (*for exclusive locks only*). The server sends an asynchronous oplock break message to the client when it receives another lock request for the same file. On receiving such a message from the server, the proxy should invalidate the cached copy of the file, if it has one, and should then send the message to the client.

3.3 Implementation

3.3.1 Read and Write caching

Caching support is achieved by maintaining a disk cache at the proxy. Both the reads and writes are cached. Thus, each time new data is brought in from the server as a response to some client request, or new data is sent to the proxy from the client for a write, it is cached at the proxy. When a data request comes from a client, it is checked against the current cache contents. If there is a hit, this request is served by the proxy, which then constructs a reply message with the requested data and sends it to the client. Otherwise the request is simply forwarded to the server. All clients at all time share al the files in the cache. We cache the writes also as subsequent read requests for the same file by the same or any other client can be serviced by the proxy (*after making sure that the cache is coherent*).

3.3.2 Invalidation of Cache Contents

Stored on disk, along with the actual data, is a time-stamp called *last-write-time*, which is the last modified time of the file from the server, when the proxy caches the data. All the *open* requests are forwarded to the server even if the proxy has the file in its cache. When the server response comes back, the *last-write-time* in the response is compared with the cached time-stamp value. If the two times are the same, then we have a cache hit, and the *in_cache* flag for that file in the in memory cache table is set.

Otherwise, the cached copy is considered stale on the proxy and the corresponding cache content is invalidated. Storing the time-stamp on disk is required by the demand of crash-resistant cache.

3.3.3 Replacement of Cache Contents

Adding new content to the proxy cache whenever there is new data will eventually exhaust the storage capacity at the proxy. To make the proxy more practical, we added the Clock replacement algorithm to replace some cache content when there is a possibility of cache overflow. A *clock arm* is maintained at the proxy among a circular linked list of all the files in cache. Each node in the linked list has a reference bit. If the bit is set, it indicates that the file was last accessed. Initially, as long as there is no need for replacement, all the files added in the cache have the reference bit set. Also, the reference bit for a cached file it set whenever the proxy services a cache hit. When a new file needs to be cached and there is not enough space in the cache, a search will be initiated starting at the location of the clock arm. If the reference bit at the current location is set, it is cleared and the clock arm is advanced. Otherwise, the node with the reference bit cleared is chosen for replacement. This ensures that that file is replaced from the cache that had not been accessed previously (*if it had been accessed the last time, the reference bit would have been set and it will not be chosen for replacement*). Thus, the clock algorithm ensures that that file which had been brought in most recently is not the one that is replaced whenever the contents of the cache need replacement.

3.3.4 Crash-Resistancy

To provide crash resistancy, all the time-stamps are stored on disk along with the actual cached files. When the proxy starts up, it scans the disk cache to read in the names of all the files currently in the cache. These file names are then used to initialize the circular linked list for the Clock replacement algorithm. In this way if the proxy crashes or dies due to any reason, all the files in the cache can still be used to serve later requests.

3.3.5 Locking

Locking introduces an extra level of complication into the proxy design since locks can be broken and the corresponding cached files have to be invalidated accordingly. The current version of *smbclient* does not support locking. To implement and test locking in our proxy, we changed the client to be able to send out exclusive lock requests with the open request. When another client also sends out an exclusive lock request, the server asks the first client to release its lock by sending an oplock break message. When the proxy sees this message, it invalidates its cached copy of the file before forwarding the message to the client.

4. Correctness and Testing

4.1 Basic Proxy Functions

The basic function of a proxy is to forward the messages between the CIFS server and clients. The client should be able to perform all of its operations just like there is no intermediate proxy. We tried all the current functions provided by our modified *smbclient*, and the results were identical as if the client was communicating directly with the server.

4.2 Caching

4.2.1 Basic Caching

The *smbclient* provides two functions for copying files, the ftp-like *get* and *put*. For testing the basic caching, we do a *get* on a file from the server. The proxy forwards the request and caches the data as it is passed through to the client. After that, we do a *get* again on the same file from the same client and from different clients and see that the request is serviced by the cached copy of the file by the proxy. To test the write caching, we do a *put* of a file, which transfers the file to the server. The proxy caches the file

as it is sent to the server from the client. We then do a *get* on the same file and see that the proxy services the request from its cached copy.

4.2.2 Invalidation

To test the correctness of the cache coherence implementation, we do a *get* on a file from the server. We then do a *get* again on the same file and see that it is serviced from the proxy cache. We then do a *touch* on the file on the server. Then, we initiate another *get* on the same file from the client. We see that the proxy does not service the *get* request from its cache now (*it has invalidated its cached file from the last write stamp sent from the server*). It forwards the request to the server and again caches the new copy of the file. In this way we provide cache coherence.

4.2.3 Replacement

To test the clock replacement algorithm implemented in the proxy, we set the cache size to a small number (to accommodate two files of approx. 8K each). We then do a *get* on the two files, which are cached by the proxy. We then do a *get* on the third file (the size of this file is such that the cache will overflow) and see that one of the two previous cached files is replaced. We then do a *get* on the file which was not replaced earlier (to make it the most current one accessed) and then we *get* another file and see that the file we got before is not replaced since it was most recent one accessed, validating the clock replacement implementation.

4.2.4 Crash-Resistancy

After killing the proxy and restarting it, we send a request for a file that is currently in the proxy cache but has not been accessed by the restarted proxy. The result is that the proxy services the request from its own cache, even for the very first request after the proxy restarts, confirming the crash resistancy feature of our proxy.

4.2.5 Locking

To test whether our implementation of the proxy does the proper invalidation of its cache contents on receiving an oplock break message from the server, we modified the client program so that it requests an exclusive lock for all file transfers (*get* and *put*). And before the client closed the file it was put to sleep for some time (so that on the server side, it would still be holding the lock). During this sleeping time, another connection was made to the server for the same file. The server sent an oplock break message that was sent to the client after the proxy invalidated its cache. Thus, the proxy did not service the request for the same file by the second client and the file was transferred from the server to the client.

5. Future Direction

We have tried to make our proxy implementation as scalable and as general as possible and have tired to incorporate as many features as possible (*caching both reads and writes, maintaining cache coherence, crash resistancy, locking, etc*). Some of the areas in which our proxy implementation can be further improved are:

- The ability of the *smbclient* to send out different kinds of oplock requests is essential for the testing of the proxy's ability to handle those locks. In other words, because the current version of *smbclient* has no oplocks at all (our modified version can handle exclusive locks), there was really no way to test if our implementation of the locking protocol could handle all kinds of locks without having to make substantial changes to the smbclient program. Changing the client to send different kinds of locks and testing if our proxy can handle them would be an interesting and essential future work.

- For the cache replacement, currently we are using the Clock algorithm. It might be more adequate to use other cache replacement algorithms for our proxy. For example, the *Greedy Dual Size* has been

proven to be an effective cache replacement algorithm for web objects and might be worth implementing.

- Our proxy currently works for SMB clients and servers running on UNIX (or its clones). The commands and messages between an NT client and server are different than that between then UNIX clients and servers. To make our proxy more general (a *black box* which can be put between *any* CIFS client and server), the added functionality of handling NT client or server requests should be incorporated.

6. Conclusion

We implemented a multi-threaded proxy with caching support between a CIFS server and client. The proxy cached both the reads and writes. The proxy maintained cache coherence and it used the clock algorithm for cache replacement. We also modified the *smbclient* to add locking capability (for exclusive locks) and tested our proxy's ability to handle locking. Our test results showed that our proxy was able to cache file requests and service cache hits successfully. Combined with its features such as its capability to manage cache with an efficient cache replacement algorithm, its crash resistancy and its ability to handle locking, our proxy can be used effectively between a CIFS server and client.

7. Acknowledgements

We would like to thank **Prof. Pei Cao** for her guidance and useful suggestions throughout the life span of the project. We also would like to thank **Colby O'Donnel** for setting up the Samba server on a public machine in the department for testing purposes and spending time setting up accounts on those machines for us.

8. References

[1] http://www.microsoft.com/workshop/networking/cifs/default.asp

[2] http://www.samba.org

[3] http://www.thursby.com/cifs/file

[4] Jussara Almeida and Pei Cao, *"Measuring Proxy Performance with the Wisconsin Proxy Benchmark"*, Technical Report 1373, Computer Sciences Dept., University of Wisconsin-Madison, April 1998

[5] Pei Cao and Sandy Irani, *"Cost-Aware WWW Proxy Caching Algorithms*", Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems, Dec 1997, pp. 193-206