Twenty Years Later: Still Improving the Correctness of an NFS Server

Robert Gardner
Hewlett Packard
rob.gardner@hp.com

Scott D'Angelo
Hewlett Packard
scott.dangelo@hp.com

Matt Sears
Hewlett Packard
matt.sears@hp.com

Abstract

The NFS reply cache, also known as the Duplicate Request Cache, was first described over twenty years ago [Juszczak] as a way to help a server give correct responses to certain types of replayed operations. Some operations, called idempotent, can be safely repeated and will do no harm. Other operations, called non-idempotent, can only succeed once [Callaghan]. For example, a request to read a certain block of a file will produce the same result each time. But an operation such as rename will succeed the first time, but a subsequent retry will result in an error being reported to the client. The reply cache keeps track of responses to recently performed non-idempotent transactions, and in case of a replay, the cached response is sent instead of attempting to perform the operation again. In addition to avoiding these client-visible errors, performance is also improved by avoiding unnecessary work.

The trouble begins when the size of the cache is inadequate to deal with the rate of incoming transactions. Now the mechanism breaks down, and replayed requests may result in duplicate work being done and erroneous results generated. Even modest workloads can result in an enormous rate of non-idempotent requests which would necessitate enlarging the reply cache to unacceptable levels. Heavy workloads can cause network congestion and delays that can foil attempts to cache enough transactions to maintain correctness. Simply increasing the cache size, even by large factors, isn't effective.

We address these problems by making the cache smarter instead of larger. First, we add the concept of protecting a cache entry, which temporarily makes it exempt from the usual replacement process. Next, we add some heuristics that grant or revoke the protection of a cache entry. Finally we eliminate automatic expiration of cache entries. Taken all together, this scheme drastically reduces the number of errors reported by clients on a large network.

1 Traditional Design

Linux drew strongly from its predecessors in its implementation of the NFS reply cache, and despite occasional rumors of being rewritten [Kirch], it has changed very little since its inception. In addition to the actual reply data, each cache entry contains other essential information about one NFS transaction, including the client's IP address, the transaction ID (XID), NFS procedure number, timestamp, etc. As entries are created, they are placed onto hash chains indexed by the XID to enable faster searching. When a new request is received, the cache is searched. If a match is found (a hit) then the cached reply is sent back to the client. Otherwise, a new entry is made, replacing an existing entry via a least-recently-used policy. An entry "expires" after two minutes, which excludes it from searches, even if other cache entries have not replaced it. This is to avoid issues with transaction ID re-use [Werme]. The cache entries are also kept on an additional linked list that is ordered by time of use. When an entry is created or touched, it is moved to the head of this list. This makes the least recently used entry instantly accessible when replacement is invoked.

The size of the NFS reply cache is critically important, since it directly affects residency time of a cache entry. If entries are replaced too quickly, then a replayed transaction will not be found in the cache and a client visible error will result. We call this a *critical reply cache miss*. A worse consequence of a critical reply cache miss is the "lost write due to replayed truncate" problem [Sun] which can cause data loss.

The design and sizing of the reply cache dates back to 1989 [Juszczak] when processing power and network bandwidth were rather limited, which in turn constrained the rate of incoming NFS requests. The power of a modern server makes configurations possible which were probably not envisioned twenty years ago. Nowadays it is not uncommon for a server to handle requests

from hundreds, or even thousands of clients simultaneously, so it is not a huge leap to realize that the reply cache must be sized appropriately for the expected load.

The number of entries in the reply cache has been subject to many changes over the years, with various implementations (i.e., BSD, DEC Unix, HP-UX, Solaris, etc) employing widely differing sizes, and the Linux code settling on 1024 entries. The paucity of data to justify any particular choice suggests that some guesswork was involved in each implementation. A clever scheme to dynamically resize the reply cache based on demand was described in [Banks], and this work will probably be incorporated into the Linux kernel in the very near future. Our experiments with a reply cache 16 times larger than normal (16384) showed cache entries surviving for mere seconds. With typical RPC replay timers being multiples of *minutes* [Eisler], we were convinced to look at possibilities that did not involve enlarging the cache.

In addition to the sizing dilemma, the cache replacement algorithm is rather unintelligent. The simple least-recently-used policy doesn't take into account any statistical data available. For instance, busy clients may consume many cache entries, but this should not have adverse effects on less busy clients. Network congestion can cause replies to be lost and wreak havoc on the simplistic cache replacement algorithm. Since there is a practical limit to how much memory should be devoted to the cache, a solution more appropriate for today's enterprise environments is needed.

2 Is this a problem worth solving?

A human operator may not notice, may not care, or very likely will not know how to interpret the symptom of a critical reply cache miss. All that will be seen by a user is a puzzling failure of a `mkdir` command, for instance, when in fact the directory was created successfully. A user faced with this scenario is likely to pretend the whole thing was a dream, and move on. Consequently, problem reports are rare, and unlikely to identify the true source of the problem. This line of reasoning might explain why we haven't seen many efforts to correct the problem.

Now envision a different type of scenario. There's a file server with hundreds or even thousands of active clients. The clients are running applications that are updating databases, or something equally critical. Perhaps client applications exit prematurely when they see unexpected failures of simple file operations. In these scenarios, critical reply cache misses are likely to be noticed, since applications are much less forgiving than human operators when faced with unexpected results. At best, this will result in customer complaints, and at worst, there may be data corruption.

New file serving protocols, such as NFSv4.1 [Noveck] [Shepler], do not suffer from this problem. But history has shown that emerging technologies do not instantly displace old ones, and NFSv3 will likely be around for many more years.

3 New Paradigms, New Problems

A High Availability NFS server coupled with a clustered filesystem [CITI] [Bhide] creates new problems for the NFS reply cache. Although high availability and cluster issues are not the focus of this paper, the myriad problems that they expose provided much of the impetus to design new methods. These new ideas are completely applicable to a standalone server.

Transient service disruptions, such as failover events, often exacerbate networking back-off mechanisms and can cause extended replay delays. In practice, a failover may require several seconds or more to complete, but may incite transaction replay delays of a minute or more. Failover events are hot spots of trouble for the reply cache because of the greatly increased probability of a lost reply and a subsequent replayed transaction.

For example, when a failover event is initiated for administrative reasons, i.e. maintenance, load balancing, etc., the contents of the reply cache from the failing server must be transported to the takeover server. Various schemes for doing this have been suggested [Bhide] and the straightforward method we use is to simply have the failing server write out the contents of the reply cache to some network accessible location, and then have the takeover server read it back. This approach immediately caused a new problem, as the flood of new cache entries from the failing server displaced more recent cache entries that were already resident on the takeover server. New logic had to be created to deal with these competing sets of cache entries.

4 Evolution and Implementation of a New Approach

The cache entry competition caused increased client-visible errors right after a failover event. As a first step, *acquired* cache entries should not displace existing entries on the takeover server, but rather should be merged with them as space allows. Since clients of the failing server are victims of a transient service disruption, they are more likely to retry requests, and thus their reply cache entries are more important than existing ones. This led to the first major change to the reply cache logic. A `protected_until` field was added to the cache entry structure, signifying that this entry is exempt from replacement and reuse until a certain time in the future. This gave the acquired entries a survival advantage over normal entries, and greatly eased the problem of failover related client request errors.

Once this protected status was available for cache entries, other opportunities arose to implement policies that could be applied to grant or revoke this special status, and thus help in other problematic scenarios. For instance, when a cache hit is found, the lookup code checks to see if it's for a transaction that is still being processed by the filesystem. If so, we have no answer to give the client, so `RC_DROPIT` is returned, which has the effect of sending no reply whatsoever. It's not clear why the client sent a duplicate request. Perhaps the original request was delayed due to a media problem, disruption in connection to a SAN or RAID, filesystem repair, etc. In any case, no response has yet been sent for the original request, and now the code is dropping the reply to the duplicate request. It seems like a sure bet that the client is going to try the request again! As such, the cache entry is a perfect candidate for "protected" status.

A similar piece of logic in the lookup code discards the reply for a request received too soon after the previous identical request. Again, it's not clear what could cause this to happen, but if it does, we know that the client may not have received a response to its previous request, and it won't receive a response to the current one either. Likewise, this cache entry ought to be protected since it is very likely that the client will try again soon. These hints were very successful in reducing critical reply cache misses.

The two minute cache entry expiration came under scrutiny next. If extreme conditions could cause replays to be delayed for long periods, then the two minute expiration time would foil any attempt to do anything more sophisticated. The best explanation available for the existence of the expiration period is that it prevents false positive cache hits when a client reuses XIDs without cycling through the entire 32-bit space available [Werme]. This may happen, for instance, when a client reboots or when multiple NFS clients originate from the same IP address [Oracle]. Though XID reuse could well be considered a client bug, with the fix belonging in the client code, a responsible server should attempt to handle this better. Our reply cache code stores a simple checksum of some of the data payload along with the traditional cache keys and requires a match of the checksum before declaring a cache hit. This substantially reduces the probability of a false positive cache hit. Given this new logic, the two minute expiration logic was eliminated entirely, thus paving the way for cache entries to have extended lives in the system.

Following the successes of the previous hints, an exploration was made of other available information that could be used to predict which cache entries are likely to be needed in the future. One predictor seemed to be very powerful: a client running a single threaded NFS application generally does not issue a new request until it receives a reply to the last one. This means that a *replayed transaction is very likely to be the last one received from that client*. This suggests that the cache hit rate can be greatly increased by simply remembering each client's most recent transaction. This rule for single threaded clients only seems to fail when the client application writes lots of data, since *write* operations may be reordered, delayed, or grouped by the client.

Storage is allocated to keep essential information about the most recently used transaction for a fixed number of clients. This new structure is called the Most Recently Used (MRU) list. It is indexed using a standard hashing function of the client's IP address and keeps track of the XID of the most recent transaction, the time that the transaction occurred, and a pointer to the actual reply cache entry for the transaction.

When a cache entry is being made, the MRU list is updated with the information describing the new transaction. The reply cache entry itself is marked with a protect time in the future using the previously described `protected_until` cache entry field. If there is an existing *most recently used* cache entry associated with the client, its protection time is cancelled, thus making it eligible for replacement.

5 Test Environment and Results

During the course of rigorous product testing, our servers are routinely subjected to extremely high loads and unusual configurations. It was in this harsh environment that many subtle problems came to light. Most of the problems can be boiled down to critical reply cache misses, and so for this paper, test data was generated using a relatively small network and artificially injecting transaction reply losses into the traffic. These losses caused the clients to retry after 60 seconds. Ten clients were doing nothing but generating a non-idempotent workload on the server, while two other clients ran the well known test suite, Connectathon [CTHON]. Connectathon encompasses a large variety of tests, each designed to exercise some particular aspect of functionality over NFS, but only the "special" tests were run since those emphasize operations that are sensitive to reply cache behavior. This setup generated roughly 450 non-idempotent transactions *per second*. Linux kernel version 2.6.29 was used, changing only the reply cache size from its original value of 1024.

Critical reply cache misses are reduced to *zero* for single threaded NFS clients, which also brings the number of client visible errors on unlink, rename, etc. to zero. The graph (Figure 1) shows the number of critical reply cache misses for several easily observed non-idempotent operations using a variety of reply cache sizes (1024, 4096, and 16384) on the stock Linux kernel. Increasing the cache size to 16384 makes essentially no difference in the number of client visible errors. The cache size must be increased to over 27,000 entries before a significant improvement in behavior is seen. Also shown is the result for a reply cache modified with our MRU logic. Note that this modified reply cache only had 128 entries.

Do be aware that although the Connectathon test suite is not sensitive to the subtle corrupting effects of replayed *write* operations, the MRU list has the same power to deal with these as it does with other replayed non-idempotent operations.

6 Potential problems and opportunities for further development

The MRU list only keeps track of one transaction per client. If there is more than one thread on a single client that is making requests, the scheme breaks down. If one

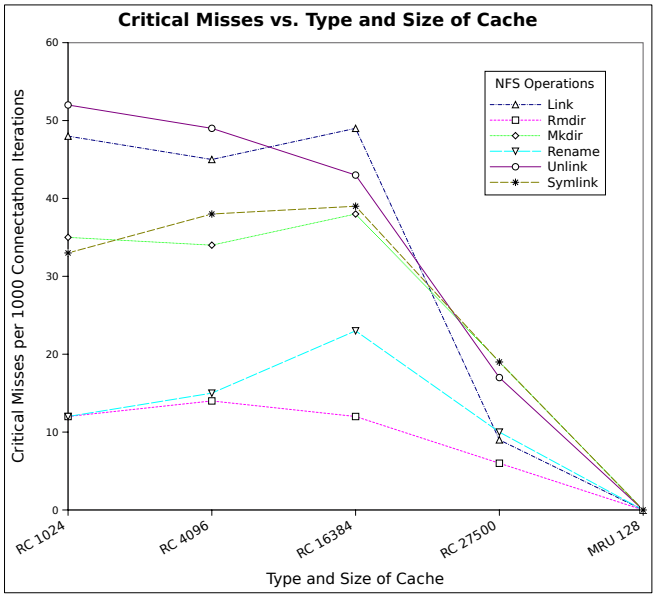

Figure 1: Results

thread gets stuck waiting for a lost reply, the other is not impeded and keeps sending requests. This could result in multiple outstanding RPC transactions, of which only one will be recorded by the MRU list. Ignoring multiple applications running on a client, this primarily affects user space NFS client implementations, such as Oracle's Direct NFS [Oracle], and clients that connect from a private network using network address translation [NAT]. The latter includes some configurations of virtual machines [Xen] [VMware] [VirtualBox].

Is it possible somehow to distinguish between two parallel streams of requests from the same client? Parallel userland NFS clients (such as Oracle DNFS) must each maintain their own TCP connections, each with a unique source port. Keeping an MRU entry for each unique (IP address, source port) pair might solve the problem for this particular case. Parallel userland NFS clients also probably generate different continuous streams of XIDs, so it may be workable to detect this and remember the last XID for each stream. It also may be possible to distinguish different client threads through their credentials (i.e., different UIDs).

When the MRU fails to provide a matching cache entry, there are some hints we can use to attempt detection. XIDs from each client are *often* monotonically increasing with respect to the client's host byte order. If the XID of a new request is numerically less than that of the most recently used one, that indicates that it might be a replay. Though information beyond the single most

recently used transaction is not saved, this situation can be flagged and a log may be kept of how often it occurs. Similarly, if a transaction arrives with an XID that matches the XID in the MRU list, but misses in the cache, then it must be because the protection time given to the cache entry wasn't long enough, and it got replaced by a newer entry. Once again, this situation can also be recorded for later analysis. Although the RPC specification [RFC1831] explicitly prohibits treating the XID as a sequence number as we do here, our use of it in this way is only a heuristic for failure detection and has no effect on the semantics of the NFS server.

Another problem is that the MRU list must keep track of at least one datum for each client. If the number of clients exceeds the size of the MRU list, we're back at square one. So another opportunity for improvement is a dynamically expanding MRU list, or at least a dynamically sizable list that an administrator could configure.

Finally, along with the addition of new data structures and code comes complexity, and subsequently the potential for bugs and performance loss. This is especially important since the reply cache can already be somewhat of a bottleneck, and at least one good effort has been made to remedy this [Banks]. The actual performance implications of the MRU list have not been thoroughly analyzed and there is the potential that improving correctness in this fashion hurts performance in an area where it cannot be afforded. For instance, the MRU list search algorithm is simplistic. A hash of the client's IP address is used as an index into the list, but if there is a collision, then a linear search algorithm takes over. Clearly this could be made better with a more sophisticated hashing scheme, but this tradeoff was made after brief analysis showed that the linear search has to traverse more than a few entries only when the MRU list is nearly full.

7 Conclusions

The current NFS reply cache implementation is not sufficient for today's enterprise environments. The fixed size cache is not large enough and there may not be a practical way to make it large enough to deal with heavy workloads without client visible errors. The logic of the existing solution only takes into account the age of a cache entry when deciding on an entry to replace. There is abundant information available that could help to make more intelligent replacement decisions, but none of it is utilized. Our contribution is to make use of some of this data, and make better decisions about which cache entries to keep and which to throw away. By making the reply cache algorithm *smarter* instead of simply larger, we have minimized the likelihood of errors in both the clustered/HA environments as well as the single server node environments.

8 Acknowledgements

Thanks to all the diligent and critical reviewers, especially Chet Juszczak, J. Bruce Fields, Greg Banks, and Stuart Friedberg. We would also like to thank Hewlett-Packard and the Storage Works Division for allowing us the opportunity to explore, develop and improve on the existing code base. These improvements are not a theoretical novelty done for research purposes, but rather are a genuine attempt to solve real world problems, and code implementing these ideas are being shipped as part of the HP StorageWorks Scalable NAS File Serving Software product.

References

[Juszczak] Juszczak, C., *Improving the Performance and Correctness of an NFS Server*, USENIX Conference Proceedings, Winter, 1989

[Callaghan] Callaghan, B., NFS Illustrated, ISBN 0-201-32570-5

[Kirch] Kirch, O., *Why NFS Sucks*, Linux Symposium, 2006, Ottawa

[Werme] Werme, R., *RPC XID Issues*, Connectathon 1996 Talks, `http://www.connectathon.org/talks96/werme1.pdf`

[Sun] Sun Microsystems, *NFS: Network File System Version 3 Protocol Specification*, 1994

[Banks] Banks, G., *Making the Linux NFS Server Suck Faster*, Presented at linux.conf.au, 2007

[Eisler] Eisler, M., *NFS over TCP, Again*, March 1, 2006, Connectathon Talks, `http://www.connectathon.org/talks06/eisler.pdf`

[Noveck] Noveck, D., *NFSv4.1 Current Status*, Feb. 5, 2007, Connectathon Talks, `http://www.connectathon.org/talks07/NFSv41update.pdf`

[Shepler] Shepler, S., Eisler, M., Noveck, D., *NFS Version 4 Minor Version 1*, IETF Draft, Dec. 15, 2008, `http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-minorversion1-29.txt`

[CITI] Center for Information Technology Integration, *Linux NFS Requirements for Cluster File System and Multi-protocol Servers*, Feb. 2008, `http://www.citi.umich.edu/projects/cluster_nfsv4/google+citi-SoW-redact.pdf`

[Bhide] Bhide, A., Elnozahy, E., Morgan, S., *A Highly Available NFS Server*, Proceedings of the Winter 1991 USENIX Conference

[CTHON] *Connectathon Test Suite*, `http://www.connectathon.org/nfstests.html`

[Oracle] *Oracle Database 11g Direct NFS Client*, Oracle White Paper, July 2007

[NAT] *The IP Network Address Translator (NAT)*, Network Working Group, May, 1994, `http://www.ietf.org/rfc/rfc1631.txt`

[RFC1831] *Remote Procedure Call Protocol Specification Version 2*, Internet Engineering Task Force, `http://www.ietf.org/rfc/rfc1831.txt`

[Xen] Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield (2003). *Xen and the art of virtualization*, In SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, New York, NY, USA, pp. 164-177. ACM Press.

[VMware] `http://vmware.com/`

[VirtualBox] Sun Microsystems, *Optimizing the Desktop using Sun VirtualBox*, `http://www.virtualbox.org/`

[Sandberg] Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, *Design and Implementation of the Sun Network Filesystem*, USENIX Conference Proceedings, USENIX Association, Berkeley, CA, Summer 1985.

[Pawlowski] Pawlowski, B., C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz (1994). *NFS Version 3 Design and Implementation*, In Proceedings of the Summer 1994 USENIX Technical Conference, pp. 137-152.

[RFC1094] Network Filesystem Specification, Version 2, RFC 1094, Sun Microsystems, Inc., March 1989, `http://www.ietf.org/rfc/rfc1094.txt`

[RFC1813] Network Filesystem Specification, Version 3, RFC 1813, IETF, June 1995, `http://www.ietf.org/rfc/rfc1813.txt`

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*