Some Consequences of Excess Load on the Echo Replicated File System

Andy Hisgen, Andrew Birrell, Charles Jerian,
Timothy Mann, Garret Swart
DEC Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301, USA
August 1992

Abstract

Understanding the workload is crucial for the success of a replicated file system. The system must continue to provide service in the presence of excess load, otherwise, its availability will be compromised. This paper presents some shortcomings of the Echo replicated file system in handling excess load. We discuss the consequences of excess load on our primary-secondary design, on our communication software, on our election algorithm, on memory usage in our file servers, and on our distributed client caching algorithm, and we speculate on possible solutions.

Introduction

Echo is a replicated distributed file system that has been designed and built at DEC SRC [3, 4, 8]. Exploiting replication to increase availability has been a major goal. The system has been in active use at our laboratory since September 1990, and became the principal file system for about 50 people in January 1991. On the whole, we have been satisfied with Echo. But there are shortcomings in the design and implementation of Echo in its handling of excess load. This paper reports on those problems, and on some possible solutions.

Echo's goal for availability is to tolerate a single failure of a server component and keep providing service. Our failure model is that server CPU's and disks are fail-stop, but clients can be Byzantine [7]. With Echo, Byzantine clients can cause denial of service by generating load, but cannot cause corruption of data or of other clients. We assume that the network can partition and can lose, replay, or delay messages. Server machines are assumed to have timers whose rates differ by a known bound.

The Echo file system semantics are one-copy serializability. Clients are thus not permitted to observe differences between replicas.

Overview of Echo

This section presents a summary of those aspects of Echo that are needed for understanding the rest of the paper.

In Echo, data storage is implemented by server computers and disks. As independent choices, an Echo hardware configuration may have replicated disks and/or multiple server computers. In general, disks are multi-ported and connected to several server computers. Each such disk has a *multi-port arbiter*, which recognizes at most one connected server computer as its *owner* at a time. Ownership is subject to timeout. Echo can also make use of single-ported disks; in this case, software on the single physically-connected server simulates the multi-port arbiter, and other logically-connected server computers access the disk via this server over the network.

The server computers are organized in a primary-secondaries scheme. Briefly, a primary is elected by having each server try to claim ownership of a majority of the disks: if a server succeeds in claiming a majority, it becomes the primary. In configurations with an even number of disks, witnesses are used to break ties [10]. A version-stamp scheme is used to determine which disks have up-to-date data [8].

Because disk ownership is subject to timeout, the current owner must refresh its ownership periodically. After a failure of the primary, a secondary must wait for the disk ownership timeouts to expire before it can become the new primary. The waiting is required in order to guarantee that there is never more than one primary. Thus, the ability to fail over quickly is dependent upon short ownership timeouts with frequent refresh.

In general, upon any failure of a server disk or server CPU, or of the communication medium between server disks and server CPU's, a new election may be held. Relatively good communication is required between the CPU's and disks that make up a single replicated service, with low latency and high

0-8186-3170-4/92 $3.00 © 1992 IEEE

bandwidth.

During service, all client reads and updates are sent to the primary. For client updates, the primary writes to all disks that are up and in communication. For client reads, only one disk needs to be read, since there is at most one primary and all update traffic goes through it.

Echo employs a distributed caching algorithm between clients and servers, in which servers keep track of which clients have cached what files and directories [5, 6, 9]. This caching information is replicated in the main memories of the server computers. In order to cache a file or directory, the client machine must first call the primary server to request the appropriate cache token, read or write. If another client machine(s) holds a conflicting token, the server will call the token back from that client(s). Before returning to the original client machine, the primary will first RPC to the secondary to inform it about granting the token.

The tokens that a client machine holds are associated with its *session*. The client must refresh its session by calling the server periodically. The refresh makes its session valid for a time period that is agreed upon by both the server and the client, the *lease* [2]. If an RPC from the server to the client to take back a token fails (e.g., because of network partition), then the server marks the session as invalid, and after waiting for the lease to expire, revokes the tokens associated with the session. Attempts to refresh an invalid session are rejected. This scheme ensures that, even in the presence of a network partition, when the server revokes a token, the client will have already concluded because of the passage of time that its cache is no longer valid.

Consequences of Load

Primary-Secondary

As explained above, Echo uses a primary-secondary replication scheme. In practice, we have used dual-ported disks, and paired servers, with each disk connected to both servers. We configure the system so that one server of the pair is the primary for one set of disks, S1, and the secondary for another set of disks, S2. The other server is then the primary for S2 and the secondary for S1. When both servers are up, we achieve some load balancing between the pair, at the granularity of an entire set of disks [3].

However, when one server is down, the remaining server in the pair must handle the entire load. Sluggishness caused by excess load may be unacceptable to applications, and extreme sluggishness is the practical equivalent of the system being unavailable. We can define availability as the proportion of time during which the system responds to requests within an agreed upon threshold. Thus, availability is a real-time problem. The current Echo configuration in our lab has eight servers, arranged in four pairs, and one pair of servers has more disks than the others. With this server pair, the system is noticeably sluggish when one server is down.[1]

During the Echo project, we neglected to build any tools for estimating, measuring, or characterizing the workload. This was a strategic mistake. For example, we should have built a tool to measure the load on a file server pair with both servers up, to be able to predict whether the system would become overloaded with one server down. Ideally, when presented with a workload by a customer and a performance target, we should be able to say whether that workload and target can be handled, how much hardware is needed, and how to configure the system.

Communication Software

Echo is built using SRC's RPC system, which offers excellent performance under moderate load [13]. The RPC system has a fixed number of kernel buffers which are mapped into the address space of every process, including the file server process. While an RPC is in progress, two buffers are in use, one each on the caller and the callee machines. Having an excessive number of in-progress RPC's will exhaust all the buffers, causing other RPC's and pings of the in-progress RPC's to fail.

The RPC interface between the Echo client and server has procedures that can be long-running, in two major categories. First, we pipeline updates from the client to the server. The pipeline is built on top of RPC—each update operation is an individual RPC, and the return of an RPC to the client implies that the update is stable on disk. Each client machine can have multiple update RPC's in progress, and there are many clients. The second source of long-running RPC's is the distributed caching algorithm. As explained above, to acquire a cache token, a client RPCs to the primary server. If another client holds a conflicting token, the server makes a nested RPC on that client. If that client has a write token and dirty data in its cache, it will have to RPC the updates back to the server before returning from the nested RPC.

[1] By using a different connection topology for servers and disks, other than strict pairing, we could have arranged that after a crash of one server, its load is shifted to more than one of the remaining servers: for the disks connected to a particular server, the other ports on the disks should be spread across multiple other servers. Excess server capacity is still required, but not a factor of two.

Furthermore, the long-running RPC's interact badly with another aspect of Echo: if a disk replica or witness is sluggish, but not so badly as to cause a new election, RPC's to update or read can block in the server. But this ties up RPC buffers and consumes CPU resources pinging RPC's, tending to make the system even more overloaded, and so on. Extreme load can cause the primary to not refresh its ownership of the disks within the ownership timeout, triggering a new election.

We could have redesigned the RPC interfaces to eliminate the long-running calls. The pipelined update RPC's could return early, with a (piggy-backed) acknowledgement later that the update is stable on disk. Alternatively, the pipeline could be layered on a stream communication facility like TCP. The token calls for which another client holds a conflicting token could post the request and return early, with a separate call-back later to say that the request has been granted: this scheme substitutes the scarce kernel RPC buffer with ordinary memory in the file server to remember the posted request.

Instead of doing this redesign, and coordinating the changes to clients and servers with our user community, we instead persuaded SRC's RPC implementors to increase the number of kernel buffers. However this expedient hack would collapse again if we doubled or tripled the number of client machines.

An alternative approach to solving the buffer exhaustion problem would be to make RPC resilient to temporary buffer exhaustion. For example, a fixed amount of kernel resources could be dedicated to providing an up-down service – if the up-down service says that the callee is up, then RPC's keep being retried, even if buffers are exhausted. A subtlety with this approach is deciding what it means for the callee to be up—it should mean that the callee process is still alive, and not just the callee kernel. And if the callee process is really making no progress, it would be better to declare it to be down, in order to insulate other servers from it. To determine callee process progress, a software implementation of a dead-man's handle in the callee would be prudent [11].

Election

We have already mentioned the possibility that excess load can trigger an election. Excess load can also cause an election to take a long time to converge.

Memory Usage

The Echo file server process is implemented in Modula-2+, SRC's dialect of Modula-2 which includes threads and garbage collection [12]. Garbage collection is convenient for implementing a long-running server, because it eliminates two classes of errors: storage leaks and dangling references. SRC's Modula-2+ garbage collector is incremental and runs in parallel with the mutator, exploiting the Firefly multiprocessor hardware [1, 14].

The potential concerns in using garbage collection in a system like Echo are: (i) bounding any pauses caused by collection, so as not disrupt RPC or the election algorithm, and (ii) ensuring that collections occur frequently enough that storage is not exhausted. Although the SRC collector is engineered to avoid these problems in the normal case, it does not make guarantees. For example, if the collector cannot keep up with the mutator over a sufficiently long period of time, it will throttle back certain mutator operations, possibly leading to excessive pauses. We believe that in practice the Echo servers have not suffered from such problems, but excess load could trigger them.

Client Machine Caching Algorithm

Earlier, we summarized the distributed algorithm for caching data on clients, and how it uses leases and refresh. Here, we make two observations. First, when the server calls back on the client to take away a token, the RPC failure time-out, that is, the time for which retransmissions will be attempted when the callee is not acknowledging, should be at least as long as the lease. Observe that in the case where the RPC fails, we are going to have to wait for the lease to expire anyway, and so making the RPC failure time-out at least this large gives the client the maximum opportunity to respond.

Second, an extremely overloaded client machine can cause the RPC from the server to the client to time-out and fail, because the client is too overloaded to acknowledge the RPC. We have seen this effect in our lab, and believe that it occurs when the client machine is paging heavily. The consequences are more serious than we would like: a client with dirty data in its cache must discard the data, meaning that it won't become permanent, rather, it is lost forever. Because Echo permits write-behind, the application that created this dirty data may have already exited, and is no longer around to receive an error notification.[2] We could solve the problem of the RPC from the server to the client timing out and failing by dedicating a portion of the client machine to handling RPC pings and acknowledgements, by pinning the relevant code and data and running threads at high priority. (Of

[2] An application may force dirty data to the server using the *sync* or *fsync* system calls.

course, processing the pings and acknowledgements does not guarantee that the body of the call makes progress, and the cache token will not be relinquished until it does. Therefore, if the overloaded client machine is sharing a file with another client, the other client may block for a long time waiting for the overloaded client.)

Conclusion

Load issues are particularly important for a replicated system, because conditions of excess load cannot be assumed to be independent between different components. Depending on the details of the design, conditions of excess load can even be mutually reinforcing.

References

[1] John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, Palo Alto, California, November 1990.

[2] Cary Gray and David Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. 12th Symp. on Operating Systems Principles*, pages 202–210. ACM SIGOPS, December 1989.

[3] Andy Hisgen, Andrew Birrell, Chuck Jerian, Timothy Mann, Michael Schroeder, and Garret Swart. Granularity and semantic level of replication in the Echo distributed file system. In *Proc. of the Workshop on the Management of Replicated Data*, pages 2–4. IEEE Computer Society, November 1990.

[4] Andy Hisgen, Andrew Birrell, Timothy Mann, Michael Schroeder, and Garret Swart. Availability and consistency tradeoffs in the Echo distributed file system. In *Proc. Second Workshop on Workstation Operating Systems*, pages 49–54. IEEE Computer Society, September 1989.

[5] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[6] Michael L. Kazar. Synchronization and caching issues in the Andrew file system. In *Winter Conference Proceedings*, pages 27–36. USENIX Association, February 1988.

[7] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Computer Systems*, 4(3):382–401, July 1982.

[8] Timothy Mann, Andy Hisgen, and Garret Swart. An algorithm for data replication. Technical Report 46, DEC Systems Research Center, Palo Alto, California, June 1989.

[9] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[10] Jehan-Francois Pâris. Voting with witnesses: A consistency scheme for replicated files. In *Proc. 6th International Conference on Distributed Computer Systems*, pages 606–612. IEEE Computer Society, 1986.

[11] John Robinson and Eric Roberts. Software fault-tolerance in the Pluribus. In *Proc. of the 1978 National Computer Conference*. AFIPS, June 1978.

[12] Paul Rovner. Extending Modula-2 to build large, integrated systems. *IEEE Software*, 3(6):46–57, November 1986.

[13] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[14] Charles P. Thacker and Lawrence C. Stewart. Firefly: A multiprocessor workstation. In *Proc. of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM and IEEE Computer Society, October 1987.