Tagging responses to clients for disaster recovery

Avinash Kulkarni
Cornell University

Abstract

This paper presents the results of an initial literature survey into methods of implementing tagged responses to clients that primary servers could use to recover their state. The paper details the practical problems associated with the use of asynchronous data replication methods for recovery. We describe existing academic results that are relevant to this topic, and how they will be adapted to implement our solution to the problem of recovery. Finally, we present an initial design, along with an implementation and evaluation plan for the proposed solution.

1 Introduction

The demands of enterprise storage place particular emphasis on the need to ensure fault tolerance. Computing environments in businesses are not immune from disaster situations, and hence invest in maintaining a replicated backup store for their primary storage servers. Such backup stores are usually kept in separate physical environments so a disaster event does not disable both of the data stores simultaneously.

This arrangement introduces the possibility of the primary and backup stores differing in their consistency because of delays in replication. One of the popular methods of ensuring consistency between primary and backup is synchronous replication between the two servers. But the use of synchronous replication reduces the performance of the system. Hence, many businesses opt for asynchronous replication because the performance penalty of synchronous replication is deemed too high.

A way out of the dilemma posed by the two approaches to replication is to choose asynchronous replication method but bolster its reliability by including a mechanism to attain consistency between primary and backup in the event of a disaster. The difference in data between primary and backup could be retained in the system for use during recovery after disasters. The location of storage space for the additional data has to be different from both primary and backup servers. Client caches' present us with a possible location for storing this information, and with the proper amount of metadata, they could be used to recover the state of the system.

This document proceeds as follows. Section 2 will describe related work, and literature survey. Section 3 will describe details about design and implementation of the proposed solution, and Section 4 will present evaluation results of the solution. Section 5 will present the conclusions.

2 Related Work

Message-logging protocols represent a significant body of work in the area of building systems that tolerate crash failures. These protocols require that each process in the system periodically record their state to some form of stable storage that could be used to recover from crashes. These protocols are classified under three categories – optimistic, pessimistic and causal – based on their approach to the question of orphan processes.

A process is termed as orphan if its state is inconsistent with the recovered state of a crashed process. Optimistic protocols rely on the assumption that failures are infrequent, and allow the creation of orphan processes to gain performance during failure free runs. Optimistic protocols include recovery mechanisms that eliminate orphaned processes from the system. Examples of optimistic protocols are in [1], [2].

Pessimistic [3] protocols work by placing restrictions on the steps of the protocol, so orphaned processes are never created. No process in a system running a pessimistic protocol ever sends out a message 'm' before all the messages that preced 'm' are logged. Reconstructing state in such a system is straightforward with logs. A disadvantage of pessimistic protocols is their poor performance because of the potential to block a process on each message it receives.

Causal [4] protocols are an attractive option – they allow the system to tolerate a certain number of failures for the cost of larger size messages containing piggybacked information. The number of failures that can be tolerated can be configured as a parameter in the system. Examples of this protocol are in [5], [6]. A description of optimizations in causal protocol to reduce the number of messages transmitted is presented in [4]. A good survey of message logging protocols is presented in [7].

The Smoke and Mirrors Filesystem [8] represents an example of an optimistic approach to the problem of incorporating asynchronous operation in a replication environment. In contrast to logging, SMFS treats the network as a storage device that contains information required to achieve consistency between primary and backup. As in other optimistic protocols, SMFS does not eliminate the possibility of the creation of orphaned processes.

3 System Model

The proposed solution relies on causal message logging protocol to add piggyback information responses sent to clients. This information can be scraped during recovery to restore consistency on the primary server.

Storing redundancy in client caches introduces a few problems. Firstly, the clients would obviously need modification to support such caching. Secondly, responses from the server to client requests need to include the additional information necessary for reconstruction state of the system. Third, clients need to be instructed by the server to purge their cache and reclaim space. Finally, the system needs a way of scraping caches of clients during recovery to

identify and use the exact segments of data needed for reconstruction.

The first step in bootstrapping the system requires the clients to register with the primary. At this step, the client sends a request to the client with a registration request α that details information on the ports that the client will use for communication. After receiving the request, the primary will synchronously save the communication information of this client at the backup, and send a registration confirmation message to the client.

Once registered, whenever a client sends a write request to the primary, the data is written out to stable storage on the primary, and scheduled to be replicated to the backup. At this moment, the primary and backup are not completely synchronized, and the loss of primary server will cause data loss. Our solution resolves this condition by tagging client response to include instructions to the client that direct it to retain write data in its cache. The tag response will also include a unique sequence number that identifies the order of the write with respect to writes since last update was sent to backup server. This enables the primary to replay write data from client caches in the order in which they were received. The design also provides for the primary to include write data from other clients in the response sent to one client. This would help provide an additional level of redundancy to battle client failures.

Formally, the primary's response to a write from a client will be a determinant:

$$< (tag_1, Data_1), (tag_2, Data_2)\ldots)>$$

Here, 'tag' is a header that constitues the metadata describing the Data that follows. Each 'tag' contains the unique sequence number assigned to 'Data'. The 'tag' also contains a boolean flag that is set by the primary to instruct clients to purge their cache.

Each client will store in its cache, the last write it sent to the server, along with any data that it received in response from the primary server. During normal operation, the client will continue to cache responses until instructed by primary to purge its cache. Primary will issue such a purge request when it completes copying data to the backup, and records a checkpoint.

When the primary fails and restarts, it will read the list of clients from the metadata that was stored on the backup. The information read contains communication parameters of clients, and provides primary the details necessary to connect to clients. After connecting to all the clients that are still active, the primary will request for and receive the cached responses from each of the clients. The primary will then replay the writes in the order of USNs and restore consistency in the system.

One of the possibilities for failure in such a scheme of recovery is the lack of response from a client. In order to counter this, the primary response contains a fixed number of random (tag, Data) tuple in them. This helps distribute state within the system. The failure of a single client will not cause irrecoverable data loss because it is possible that a particular

tagged write is present in some other client's cache. If some of the clients fail to respond, the primary can use responses from other clients to regenerate the missing data. If such regeneration fails, data loss is declared.

IMPLEMENTATION

We have used libasync to implement communication between primary and client processes. When starting up, both client and primary processes create sockets, bind them to their local addresses and ports, and then enable them for asynchronous operations. The backup storage is implemented as a filesystem that uses Amazon's S3 service for storage.

During normal operation, on receiving a write from a client, the primary stores the data to local storage, and logs it after assigning the write a new unique sequence number. The primary then generates a new response to the client by concatenating a certain number (1-3) of randomly selected log entries, and sends it back for the client to cache. Clients receive these responses and store them in their local caches. At periodic intervals of time, a flush timer is triggered using the delay callback implemenation in libasync. The flush callback routine iterates through the list of log entries on the primary, and writes them out to the backup.

If the primary fails, it can be restarted in recovery mode. In this mode of operation, the primary first reads in the list of clients stored on the backup. The list of clients will contain the IP address of the client as well as the port that the primary can use to connect to the client. The primary will connect to each client on the list, and if a connection is established, issue a query message for the cache contents on the client. The client will respond back with its cache contents. After collecting all the log entries from clients, the primary will then perform a replay of the log entries in the order of their unique sequence numbers.

4 EVALUATION

We evaluated this solution in terms of amount of data that could be recovered for the amount of tag overhead introduced.

If each client were to cache its own writes, then the tags could just acknowledgements of write completion from the primary. No data needs to flow back from the primary to the clients. However, this solution has the obvious drawback of losing data even if a single client fails. Hence, this solution was rejected.

So, our solution requires responses to contain a certain number of randomly selected tags and data in them. We evaluated the effectiveness of the solution based on the number of tags in a response. For a constant number of clients, we varied the number of tags sent back with each response. Then we increased the number of clients, and performed the same experiment.

For the test, we first configured the primary with the number of tags to be sent in a message. Then we created a certain number of clients and let the system run for sometime. After waiting for a fixed interval of time, we killed the primary and

examined the state of the client log and the backup. We then calculated the ratio of the number of the packets that were in primary but not in the backup, and the number of unique packets that were contained in client caches. A packet was lost if it had not been backed up, and did not exist in any of the client caches. This implied that the process of random selection never caused a lost packet to be the part of any tag sent to any client.

Fig. 1 shows the results of this experiment. As we can see, for a fixed number of clients, as the number of tags in a response increase, we can recover a higher number of writes from client caches. A higher number of tags with each response will ensure that some client always contains a copy of a write, and it can be recovered.

For the same experiment, if we increase the number of clients in the system, the reliability of data recovery is higher. This is because the larger amount of cache is available in the system and the amount of redundant tags that can be stored is higher.

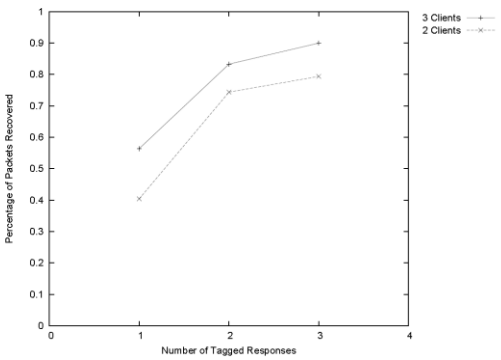

Fig 1.

On the other hand, a higher number of tags with each response introduce overhead in data transmission. Also, the effectiveness of cache is somewhat reduced because some of the cache across the system stores redundant information.

5 Conclusions

Our solution presents an alternative method of exploiting the better performance of asynchronous replication while retaining the ability to reconstruct data on disastrous failures. We have shown that as the number of clients increase, the reliability of the system improves. For fewer clients, we could achieve higher reliability by increasing the number of tags sent in each response.

6 References

[1] D.B. Johnson and W. Zwaenepoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing," *J.Algorithms*, vol. 11, pp. 462-491, 1990.

[2] S. Venkatesan and T.Y. Juang, "Efficient Algorithms for Optimistic Crash Recovery," *Distributed Computing*," vol. 8, no. 2, pp. 105-114, June 1994.

[3] D.B. Johnson and W. Zwaenepoel, "Sender-Based Message Logging," *Digest of Papers: 17th Ann. Int'l Symp. Fault-Tolerant Computing*, pp. 14-19, IEEE Computer Society, June 1987.

[4] Alvisil, L. and Marzullo, K. 1998. "Message logging:pessimistic, optimistic, causal and optimal". *IEEE Transactions on Software Engineering 24*, 2, 149-159.

[5] E.N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback and Fast Output Commit," *IEEE Trans. Computers*, vol. 41, no. 5, pp. 526-531, May 1992.

[6] L. Alvisi, B. Hoppe, and K. Marzullo, "Nonblocking and Orphan-Free Message Logging Protocol," *Proc. 23rd Fault-Tolerant Computing Symp.,* pp. 145-154, June 1993.

[7] E.N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David. B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems"

[8] Hakim Weatherspoon, Lakshmi Ganesh, Tudor Marian, Mahesh Balakrishnan, Ken Birman. "Smoke and Mirrors: Reflecting Files at a Geographically Remote Location Without Loss of Performance". Appears in *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, February 2009