PAPER TITLE HERE

YOUR NAME HERE

Abstract

Chain replication storage systems such as CRAQ[2] target at improving read throughputs for read-mostly workloads. However, write requests can only be handled by the head node of the chain, resulting in low write throughputs. Amazon's Dynamo[1] sacrifices strong consistency for high availability, but only eventual consistency is achieved. Neither of these two systems finds the balance point between availability and consistency. From user's point of view, data objects are generally classified as important and unimportant. Important objects prefer strong consistency and low write throughput is acceptable, while eventual consistency is sufficient for unimportant objects, but high availability is preferred. Based on this observation, we propose CRAQamo, a storage system which supports strong consistency and eventual consistency at the same time.

1 Introduction

Many commercially deployed systems, such as Amazon Dynamo[1], sacrifices strong consistency properties for high availability and throughput. This sacrifice is driven by the need of providing client applications with an "always-on" experience. For example, a customer needs to manipulate items in the shopping cart without any latency, but it is usually acceptable to the customer if deleted items appear again due to conflicts or failures unless it is not the final check-out. Another example may be a user trying to change his profile picture. Strong consistency is not required here because the user usually allows some time for the change to take effect. However, when the user place the order, strong consistency is highly preferred.

CRAQ[2] is a chain replication storage system which guarantees strong consistency, but it targets at read-mostly workloads and cannot provide high write throughput/availability because every write request needs to go through the head of the chain. Apparently, given inexpensive hardware, one can never guarantee strong consistency and high write throughput at the same time. Based on this assumption, we ask the question: in terms of the performance, can we flexibly switch between CRAQ and Dynamo? In other word, can we design a storage system which still supports $put()$ and $get()$ operations, but $put()$ accepts one more parameter - $consistency$? $consistency$ is a binary flag indicating whether the object being updated is strongly consistent or eventually consistent.

We call such a system as CRAQamo, implying that it is a hybrid of CRAQ and Dynamo. Specifically, when the client application does not require high availability, for example, the shopping cart, CRAQamo only supports eventual consistency. When the client application requires strong consistency, for example, the customer's list of orders, CRAQamo sacrifices write throughput to guarantee strong consistency. CRAQamo is not a system which simply runs CRAQ and Dynamo in parallel, because each of these two systems have their own arrangements of nodes, failure detection and recovery strategies. CRAQamo merges them into a single system for maximal efficiency.

2 Design

2.1 General framework

In CRAQamo, each object is given a tag of strong consistency or weak consistency. In all reads, we check the object's consistency level. If it is a weak consistency object, then we immediately return the data. If extra time is allowed, we can send a read request to other nodes, and return the newest version of the object, and overwrite the out of date data. If it is a strong consistency object, we perform the standard algorithm of CRAQ. In all writes, we check the object's consistency level. If it is a write to a strongly consistent object, we send a write to the head and apply the normal CRAQ write pipeline. If it is a write to a weakly consistent object, we multicast the write to all servers. If we have extra time, based on a user-defined value, k, we wait for k successful responses to come back, at which point we notify the user. This will contribute by being able to provide a dual storage mechanism, we will be able to meet the needs of the user in terms of efficient consistency when needed, and less efficient consistency for increased throughput when needed.

We are planning on running similar evaluations as CRAQ to confirm that our additions do not slow it down in the strongly consistent case. However, we will be doing additional tests with scalability at a global scale.

We are planning on running on multiple nodes, with enforced high latency between them, simulating data centers spread around the globe. We think that we will be able to greatly outperform CRAQ in this situation, when we are operating with weakly consistent objects.

Our work plan initially begins with setting up CRAQ, as we think that a large portion of the difficulty will lie in the understanding of how to build and deploy it. After getting that done, we are planning on attempting modifications to add data on each object about whether it is strongly or eventually consistent. Next, we will at the breaking points between where we are strongly and eventually consistent, and will code up the eventual consistency. Lastly, we will add the ability to change consistency with a special style of write, and do the extra work in user modifications we want to get to if we have extra time.

We are planning on dividing the labor evenly. Aside from small investigations we are working on setting up and building the system, we are planning on doing the work together. We will work as a group together on the code. We do not want to formalize exactly what individual people will be doing, as we do not yet know the system well enough to divide up the labor completely. Instead we will use tri-programming until we get a good basis for being able to then separate the tasks. This will likely occur after we have figured how to build and deploy the system, and after we have learned how to add strong vs eventual consistency tags. Tentatively, we will split it up by one group member handling reads, one group member handling writes, and one group member handling conflicts and failure resolutions.

2.2 Interfaces

2.3 Strong consistency read and write and failure recovery

Our system is built upon CRAQ, thus it has the full capability of CRAQ. CRAQ naturally supports strongly consistent read and write. We keep the strongly consistent storage interfaces in CRAQ and leave an additional flag for the user to determine whether to apply strongly consistent operations. We also adopt the failure recovery mechanism in CRAQ.

2.4 Broadcasting data

In CRAQ, data is only passed between neighboring nodes, which causes the write operation inefficient because each write operation needs to go through all the node although strong consistency is guaranteed. In CRAQamo, when strong consistency is not a must, we allow direct communication between any two nodes. Specifically, we have an internal method called “eventual consistency propagate" to help pass information globally among all the nods. We show the necessity of having this internal helper method in the following sections.

2.5 Comparing versions

Like CRAQ, each node in the chain also store multiple versions of objects in CRAQamo. CRAQ uses a monotonically-increasing number as the version number, which is local to the node. Synchronization is not a problem to CRAQ since as mentioned previously, data is only passed between neighboring nodes. However, such a simple representation for the version can handle the necessity of global broadcasting well. Instead, CRAQamo uses global timestamps to represent versions. We would like to emphasize that replacing monotonically-increasing version numbers with timestamps do not affect the original functionalities of CRAQ, thus strongly consistent operations are still supported by CRAQ and we do not discuss them in detail here.

For eventually consistent operations, we need to remotely compare versions of objects between nodes. Therefore it is necessary for each node to fetch the newest timestamps on other nodes and return boolean values indicating whether the current node keeps the newest version of the object of interest. Specifically, a node signals other nodes to return their current timestamps. Since the timestamps are global, comparison and be easily done by comparing the actual values of timestamps (larger value means newer and vice versa). Such version comparing method also severs as an internal helper method we detail the usages in the following sections.

2.6 Eventually consistent read

Eventual consistency allows read operations to return newly written objects before they are fully synchronized among all nodes. This is the key to improve the read throughput comparing with CRAQ because CRAQ is constrained by the strong consistency capability, thus only committed objects can be returned in CRAQ. In the eventual consistency mode of CRAQamo, the newly written object can be directly return to the requester as long as the timestamp of the object is sufficiently new. Note that it is not strictly required that the timestamp of the object on the current node is the newest. We apply a Dynamo style parameter R so that the user can determine the level of eventual consistency.

When an eventually consistent read operation is performed on a node. The node first compares the timestamp of the requested object with the timestamps of the object on all other nodes. To speed up the process, the comparisons are done in parallel. Once there are R other nodes agree that the current node has the newer version of object, the current node directly returns the object to the user.

If any of the R nodes disagrees that the current node has the newer version of object, “eventual consistency propagate" is executed so that the newer object can be passed from the node which is holding the new version of object to the current node. By varying R, the level of eventual consistency can be flexibly changed. For example, lowering R lowers the probability that the slow “eventual consistency propagate" method is called so that the overall read throughput can be increased.

2.7 Eventually consistent write

A critical reason that chain replication storage systems are generally slow in write operations is that the write operation needs to go through all the nodes in the chain. While this guarantees strong consistency, the latency is also large when nodes are placed spatially distant to each other. However, if strong consistency is not a must, one can choose to write new data to any node in the chain instead of writing to the head node. In CRAQamo, we allow the user to write to any node and safely check whether there are other writes performed on other nodes when the write operation is being performed. Note that write operations happening almost concurrently on multiple nodes rarely happens, so the synchronization can take slightly longer time and it will not critically affect the overall write throughput.

When an eventually consistent write operation is performed on a node. The node first takes the input object, writes it locally and attaches a timestamp to the updated object. There is a possibility that when it writes the object, another write operation of the same object is performed on another node, thus the written object is no longer the newest even if the current node finishes writing. To make sure that the newly written data is the newest, a safety check is done when the write finishes. The current node compares the timestamp of the newly written object with the timestamps of the object on all other nodes. Once there are W other nodes agree that the current node has the newer version of object, the current node signals the user that the write is successful and uses “eventual consistency propagate" to broadcast the newly written object to all other nodes. If any of the W nodes disagrees that the current node has the newer version of object, “eventual consistency propagate" is also executed so that the newer object can be synchronized among all nodes. After synchronization, the current node signals the user that the write is successful.

2.8 Implementation

3 Evaluation

3.1 Experiment setup

3.2 Write efficiency

Vary the proportion of weak consistency write

3.3 Read efficiency

Vary the proportion of weak consistency read

4 Related Work

4.1 Chain Replication (CR)

The chain replication storage system is a distributed object storage system which is capable of providing strongly consistent operations. In chain replication, data-centers (or nodes) are linked in a chain in which only neighboring communication is allowed in normal cases. The chain replication storage system supports two operations *read* and *write*. All read operations are handled by the tail and all write operations are handled by the head. During writing, the new data needs to be propagated from the head to the tail until the updated object is marked as committed. When a read request occurs, the tail node simply returns the newest committed object. While the working mechanism of chain replication is simple, it is obvious that the latency of both read and write operations can be very high, especially when the nodes are placed geographically distant to each other. For example, if the tail node is placed in the US, users from other countries still need to request data from the tail and the latency is expected to be high. The situation gets even worse for write operations because the new data needs to be propagated all the way from the head to the tail and it also results in high latency.

4.2 CRAQ

CRAQ is a variant of the chain replication storage system and it targets at improving the throughput of strongly consistent read operations by allowing users to access data from any node in the chain. The idea is to let each node keep several versions of data, and to use a flag indicating whether the version is *dirty* or *clean*. In terms of keeping track of the version, CRAQ simply uses a monotonically-increasing integer as the version number of the object. *Clean* flag means that the version is committed and it is safe to directly return it to the requester. *Dirty* flag means that the newly written data is still being propagated in the chain or the commitment message is still being propagated back from the tail. Then the current node needs to ask the tail for the latest committed version number and

returns the data associated with that version number. This modification greatly improves the read throughput of CR as in most cases, the newest version of object is ready to be returned. CRAQ also naturally supports eventually consistent read operations because each node keeps multiple versions of objects and eventual consistency read can be easily achieved by directly returning the latest version. However, CRAQ only supports strongly consistent write operations, so the write throughput still remains a bottleneck for CRAQ.

4.3 Dynamo

Dynamo is a high available key-value storage system deployed by Amazon. The main observation through Amazon's own services is that sometimes availability is highly preferred so that it is possible to sacrifice strong consistency for high availability. In order to further increase write throughput, conflict resolution is executed during read instead of write so that write operations look instant to the clients, giving an "always-on" experience. Unlike chain replication, Dynamo uses consistent hashing to partition the data, and each node takes care of a number of keys.

4.4 Eiger

5 Conclusions

References

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[2] J. Terrace and M. J. Freedman. Object storage on craq: High-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference*. San Diego, CA, 2009.