# PAPER TITLE HERE

YOUR NAME HERE


## Abstract

Chain replication storage systems such as CRAQ[4] 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[4] 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 read or write request is given a tag of consistency requirement: strong or eventual. Strongly consistent reads and writes are performed as standard CRAQ operations. Enventually consistent read requests can go to any node on the chain like CRAQ, but the most recent data is returned immediately before confirming that every server has updated to that value. Enventually consistent write requests can be served by any node as well, but the written value is send to other servers using ‘multicast‘ instead of propagating from the head node to the tail node. In practice we define a value, $k$ such that before a server respond to any request, it must synchronize other servers and wait until there is at least $k$ successful responses, then reply back to the user. Effectively we implemented a system where a certain number of participants must agree on a value before it can be committed. With both strongly and eventually consistent operations supported in CRAQamo we can meet the the users' consistency and performance requirements in different application scenarios using a single infrastructure.

Intuitively, lowering the consistency constraint to eventual consistency can help us achieve much better write latency and throughput, however, the read performance may degrade slightly due do pair-wise communiacations between servers to compare data versions. The improvements and overheads will be presented quantitatively in our evaluations. The performance advantage of eventual consistenty becomes more obvious as we increase the communication latency between servers to simulate data centers that spread around the globe. We modified the CRAQ source code and implemented data propagation and synchronization procedures using the same RPC interface as used originally. The network latency is controlled by port forwarding using “ncat" program which is able to redirect data from one network port to another while adding a controlled amount of latency.

## 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

Our prototype implementation of CRAQamo was implemented in approximately 3700 lines of C++, built upon the a prototype implementation of CRAQ implemented in approximately 3000 lines of C++. Like CRAQ, CRAQamo uses the Tame [2] extension to the SFS asynchronous I/O and RPC libraries [3] and all network functionality between CRAQamo RPC nodes is exposed via Sun RPC interfaces. Unlike CRAQ, CRAQamo also uses the built in C++ multithreading library. To implement the broadcast reads and writes for consistency checks, CRAQamo needs to spin up multiple threads, and wait until $w$ of them complete their calls before returning. Even after returning, however, CRAQamo needs to continue the consistency checks for the nodes that haven’t yet completed. This proved too difficult of a pattern to express in Tame, so we instead built our own manner of multithreading for these checks. However, for all other asynchronous calls, we used Tame to decrease code complexity.

Our integration with ZooKeeper was provided with by the expansion of CRAQ, and was integrated in the same way, with CRAQamo nodes able to query the same ZooKeeper files as CRAQ nodes. Furthermore, expansion of CRAQ gave us free chain node functionality and the ability to handle membership changes dynamically. Because data is stored in the same manner as CRAQ, and because no node has special knowledge in CRAQamo, CRAQ’s algorithm for membership changes proves sufficient for CRAQamo.

## 3 Evaluation

### 3.1 Experiment setup

We used Princeton University’s *tux.cs.princeton.edu* virtual machines. These machines have 4 Intel(R) Xeon(R) CPUs which each run at 3.07GHz. We used only local connections for our testing, to make the testing both easier, and more consistent. To simulate the high latency environment that would be run in practice, we used ncat to forward ports while adding the needed delay. The amount of delay in $x$ miliseconds will translate to $2x$ milisecond of round trip time, or RTT in real long range network communications. We used a chain size of 3 nodes for the tests.

### 3.2 Performances under different latencies

Vary the latency

### 3.3 Mixing different consistency models

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 as well as a number of the nodes before it depending on the number of replicas desired. One major advantage of Dynamo is allowing users to tune a few parameters including the number of replicas, the number of read confirmations and the number of write confirmations so that the trade-offs are always in control. For example, increasing the number of write confirmations, although increases consistency, increases the latency at the same time. In CRAQamo, we adopt the parameter tuning idea in Dynamo to control the consistency.

## 4.4 Eiger

Eiger is a geo-replicated storage system which provides low latency, causal consistency, read-only and write-only transactions. All client requests can be satisfied in the local data center. Eiger supports useful data model abstractions such as column-families and counter columns which provides rich structure allowing programmers to represent and query data efficiently. Eiger provides causal consistency and allows clients to access data with non-blocking read-only and write-only transactions. The problem of Eiger is that all the objects applies a single consistency model. In practice, some objects require strong consistency while others do not. CRAQamo solves this problem by combing hybridizing both strongly consistent operations and eventually consistent operations.

# 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] M. Krohn, E. Kohler, and M. Kaashoek. Events can make sense. In *USENIX Annual Technical Conference*, 2007.

[3] D. Mazierez, M. Kaminsky, M. Kaashoek, and E. Witchel. Separating key management from file system security. In *Symposium on Operating Systems Principles*, 1999.

[4] 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.