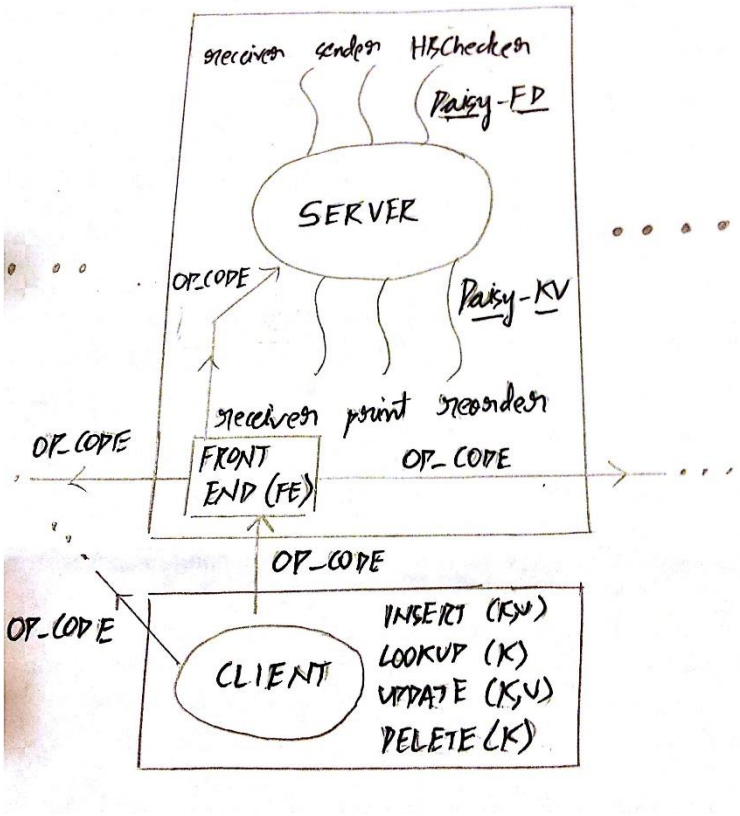
Helenus

Introduction

The *Helenus* is an extension of the *Daisy-KV* which is the *Daisy Distributed System's* key-value store. Like the other component, *Helenus* is a multi-threaded, fault-tolerant, eventually-consistent, key-value store that also supports consistency. The high-level architecture of the *Helenus* is shown by the diagram below:



Design

- *Basics*

The client receives the respective command from the user and creates a message called the op-code, together with the key and/or value. The client then can contact the local front end thread in the current host or it can also contact a front end thread of a remote host directly if it has the awareness of the remote node. The FE thread running in the server queues the requests if there are multiple requests and then checks the timestamp of the request. Based on the key it's routing on the chord logical ring is determined. It is either a local operation or a peer node routing. If it is a peer node routing it is forwarded to the peer node thread running on the peer server. If it is a local operation, the *Replica Manager* is kicked off which

Rajath Subramanyam
Karthik Rajashekar Gooli

replicates or performs the necessary operations on the replicas. The servers to maintain the replicas are chosen based on a consistent algorithm applied on the virtual chord ring.

- *System Churn*

Whenever a node joins or leaves the *Daisy Distributed System*, there is a thread to perform a reorg on the local key value store which contain both original entries and replicas of other nodes' original entries. Whenever a new node joins the distributed system or when a remote node leaves the distributed system this is recognized by this thread and reorgs its local key value store based on the current membership. An algorithm automatically takes care of re-replicating replicas, finding new owners in case of failures. If any node voluntarily leaves the system, its key value store is routed to different peer nodes before exiting the distributed system. There is also a thread running that supports printing the local key value store and the membership list. The *Daisy-KV* uses reliable TCP for exchanging operation codes, results, etc and uses UDP for membership protocol. It has been implemented in C programming language with multithreading using pthreads and it significantly increases performance.

Helenus Goals

- *Fault-tolerance: Helenus* uses *active replication* with a replication level of 3 i.e. for every entry in the key value store two other copies are maintained on remote peer nodes. When the FE receives an insert op code, it firstly inserts the value in the local key value store and then spawns the *replica manager* which based on a consistent algorithm chooses two other nodes from the membership table and sends them replicas. The hash table has metadata information to differentiate between an original local entry and remote replica entry. Once the remote peer node is done successfully with the operation, it sends an acknowledgement back to the FE it received the request from. The FE waits only for specified *consistency level* of acknowledgements.
- *Concurrency: Helenus* supports three *consistency levels* of: ONE, QUORUM and ALL. *Helenus* also uses *NTP time synchronized clocks* and appends timestamps to every request sent by the client. The FE queues every concurrent request and serves request based on timestamp and thus it obeys Last Writer Wins (LWW*).*

"Cool" Application

We have built a simple "*Dictionary*" application to insert, update, delete and lookup words in the English dictionary. The key is the word and the value is the meaning.

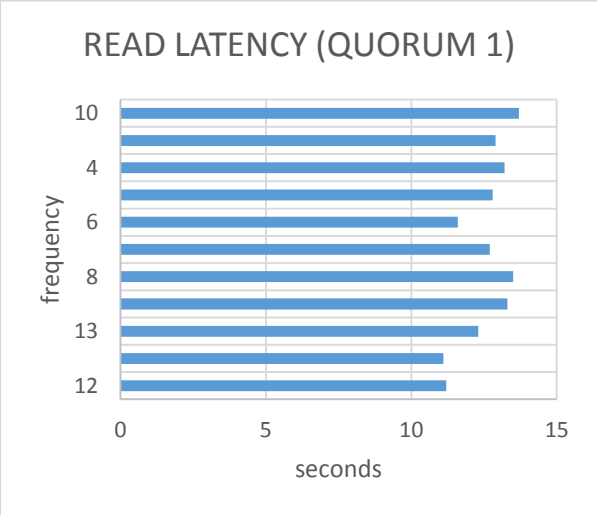
How useful was MP1 ?

The MP1 i.e. *Daisy-L* was extensively used for debugging purposes. The *Daisy-KV* follows a multithreaded model and hence there are several threads running simultaneously doing different things. It becomes increasingly difficult to debug issues if any with a conventional debugger. The *Daisy Distributed Systems* follows the mechanism used by most distributed systems i.e. logging. The MP1 was used to get important debugging information from across several nodes in a single place.

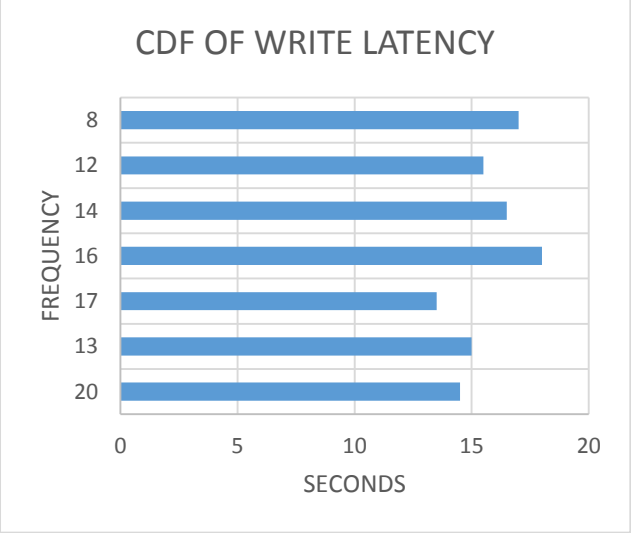
Rajath Subramanyam
Karthik Rajashekar Gooli

Performance Graph and Extra credit:

Read Latency When Quorum is 1:

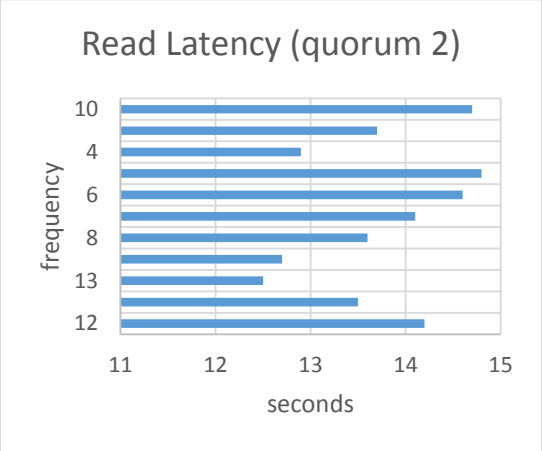


Write Latency when quorum is 1:

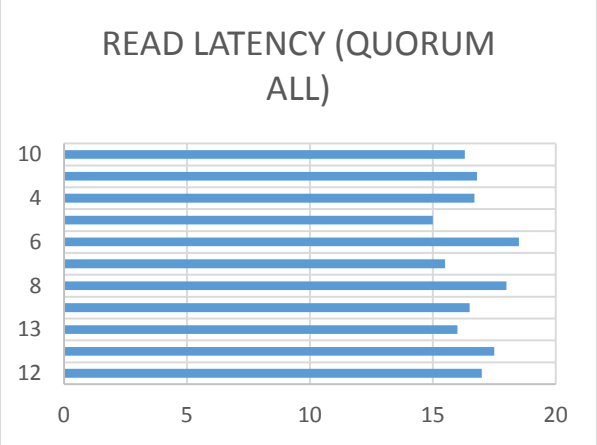


CDF for READ LATENCY when quorum 2

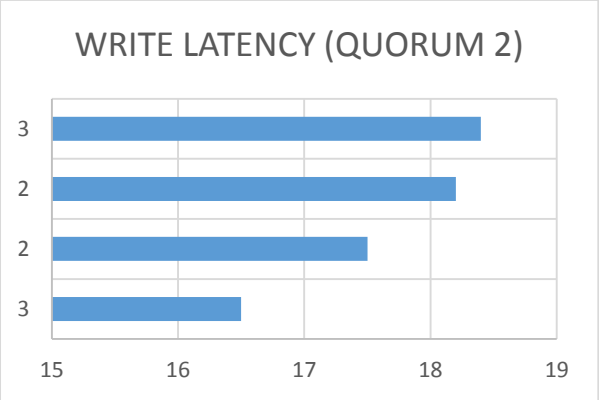
Rajath Subramanyam
Karthik Rajashekar Gooli



CDF for READ LATENCY when quorum is All

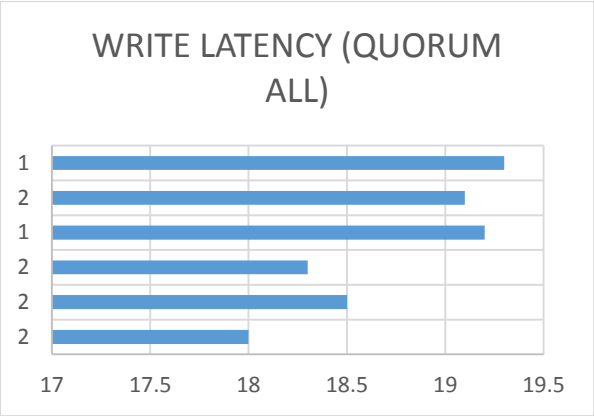


CDF FOR WRITE LATENCY WHEN quorum is 2

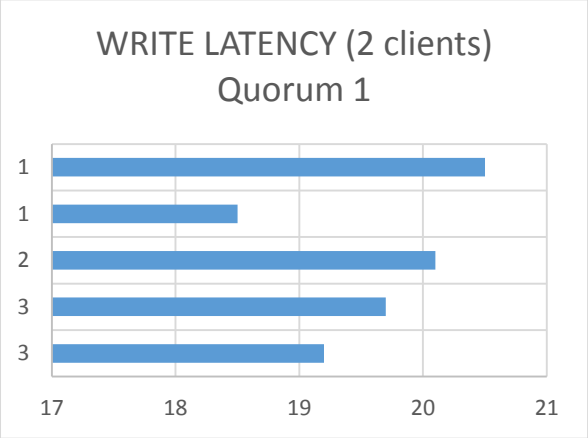


CDF for WRITE latency when quorum is All

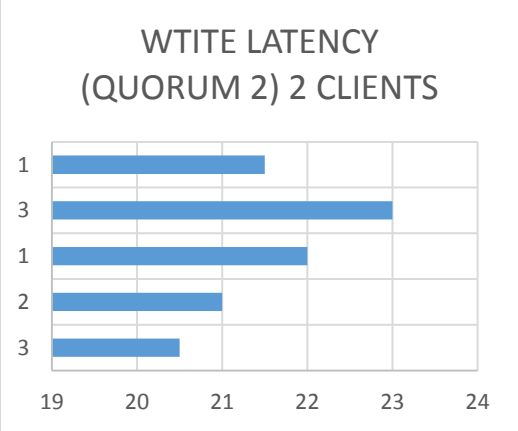
Rajath Subramanyam
Karthik Rajashekar Gooli



Write latency when quorum is 1 and clients are 2

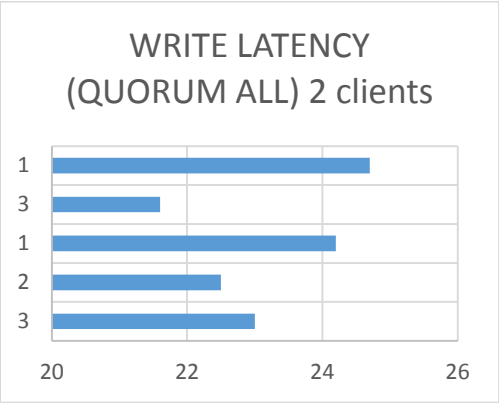


Write latency when quorum is 2 and clients are 2

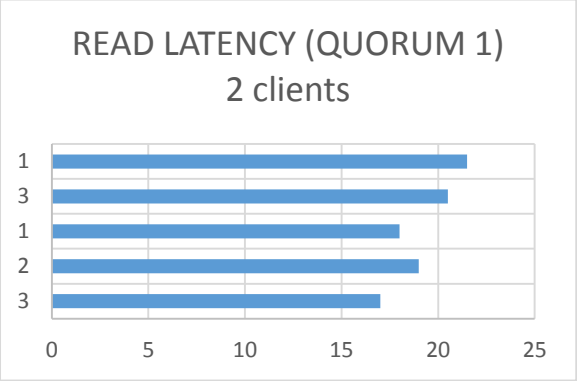


Rajath Subramanyam
Karthik Rajashekar Gooli

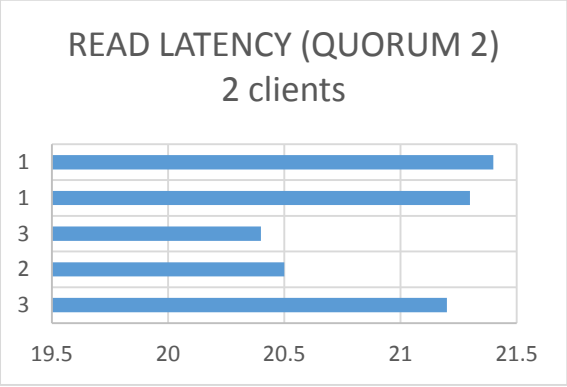
Write latency when quorum is all and 2 clients



Read latency when quorum is 1 and 2 clients

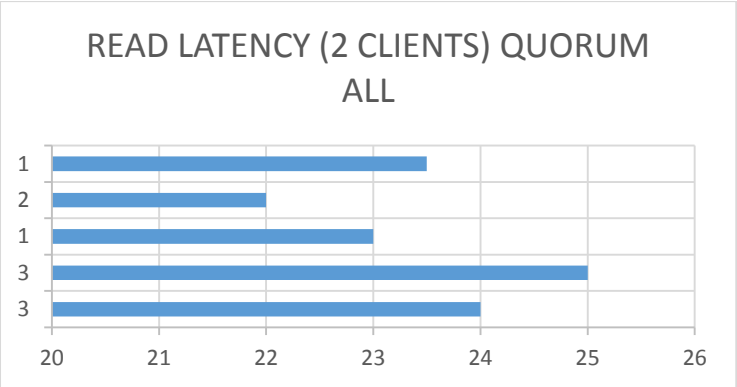


Read latency when quorum is 2 and 2 clients



Read Latency when quorum is all and 2 clients

Rajath Subramanyam
Karthik Rajashekar Gooli



CDFs above are calculated when reads happen 1000 times simultaneously for all the above to get a real latency under a good workload.

When the read and writes happen almost at the same time, staleness is seen. When the read and writes happen differing by 1-2 second staleness isn't seen (this is for quorum 1).

For quorum 2 and 3, when the read and write differs by more than 2 second staleness isn't seen.

Rajath Subramanyam
Karthik Rajashekar Gooli