Gregory Wint

MyStore: A Raft-Based Distributed Key-Value Store

Fault-tolerance, the ability of a group to withstand the failure of some subset of its parts, must be a high priority for any mission-critical system. The redundancy that allows for this fault-tolerance in computer systems must be achieved in a way that accounts for the issues that can prevent a cluster of machines from behaving as a single, coherent unit. These challenges range from random machine failure, to network communication problems, to clock skew and mostly stem from the inherent unreliability present in certain aspects of computer systems. These issues must be addressed along the path to achieving distributed consensus, a state where a group of machines are in agreement regarding what data to store, and the operations to perform on that data. Leslie Lamport's Paxos, a popular distributed consensus algorithm, is often viewed as complex and unclear regarding needed extensions by those responsible for implementing systems based on the algorithm. Raft, originally published in *In Search of an Understandable Consensus Algorithm* by Diego Ongaro and John Ousterhout, was created as a more easily understood alternative to Paxos.

Raft is a leader-based consensus algorithm consisting of two main pieces, leader election and log replication. A leader in Raft is the machine responsible for servicing client requests and making sure that those requests are registered securely within a cluster. Leaders maintain control over the cluster by periodically sending messages to the other servers in the cluster, reminding them of its current role. When an amount of time known as the election timeout passes without a particular server receiving a message from the current leader, that server initiates an election where a majority vote-share is needed to win. The period of time between successive elections is known as a term, and these terms serve as a time-keeping mechanism in Raft. Once a leader is elected, that server begins to remake the cluster in its image through the log replication process. The leader, through the periodic communication it has with the other servers in the cluster, forces servers to remove entries from their log that conflict with its own. Once client requests have been successfully replicated across a majority of follower logs they are eventually applied to the state machine present on each server, and an affirmative response is returned to the client.

During the development of MyStore it was occasionally appropriate to deviate from what was laid out in *In Search of an Understandable Consensus Algorithm*. One such decision was making remote procedure calls sequentially, rather than in parallel as was prescribed. This decision stemmed from a wish to avoid the additional complexity that would be introduced by the need to wait for the contacted servers to respond. Another difference between the implementation of Raft in MyStore and that laid out in *In Search of an Understandable Consensus Algorithm* is that in MyStore election timers are reset after elections conclude rather than before they begin. Functionally, both options are the same because MyStore pauses the election timer for a particular server when that server initiates an election. This is accomplished by placing the code for election initiation in the same thread as the timer. Another difference between Raft as described by Ongaro and Ousterhout and MyStore's implementation is that rather than retry requests indefinitely, MyStore dedicates a particular amount of time to retrying a given request, and then relies on mechanisms built into the protocol if further retries are needed. If an AppendEntry request fails, a job is added to a retry queue serviced by a dedicated retry thread. This thread spends a specified total amount of time waiting for a response to the relevant request, but gives up if that time period expires with no response. At that point, the AppendEntry request is retried as part of the heartbeat mechanism: log entries missing from a particular server's log are sent as part of the heartbeat messages that flow from the leader to the other servers. The “nextIndex” data structure maintained by each server tracks which entries need to be replicated on which servers. This approach minimizes the number of messages that must be handled by the retry thread and reduces complexity by allowing for greater separation of responsibility between the heartbeat mechanism and the retry thread: the latter can be dedicated only to the most recently failed requests, while the former can handle longer term failures. Failed RequestVote rpc’s are not retried at all. If a particular election fails to produce a leader due to failed RequestVote rpc’s, the term will end with no leader and another election will be conducted. Again, this choice avoided the addition of new mechanisms where existing ones could be relied upon. Another implementation decision made in MyStore was the choice to not cache request responses. Instead, servers track request identifiers so that they can determine what work must be done to service a particular request based on whether the request is new or old. Since key-value pair creation is idempotent when the same mapping is performed consecutively, there is no harm in allowing the same request to be made to two different servers. This could potentially happen in a situation where a current leader successfully replicates a log entry on a majority of the cluster, but dies before a response can be made to the client, causing the request to be retried with the new leader.

The read and write performance of MyStore was measured against that of etcd, a popular, open-source distributed key-value store used in products such as Kubernetes and Openstack. The tables below show the raw data for time per read and time per write for both MyStore and etcd across a series of cluster sizes.

Cluster Size – 3

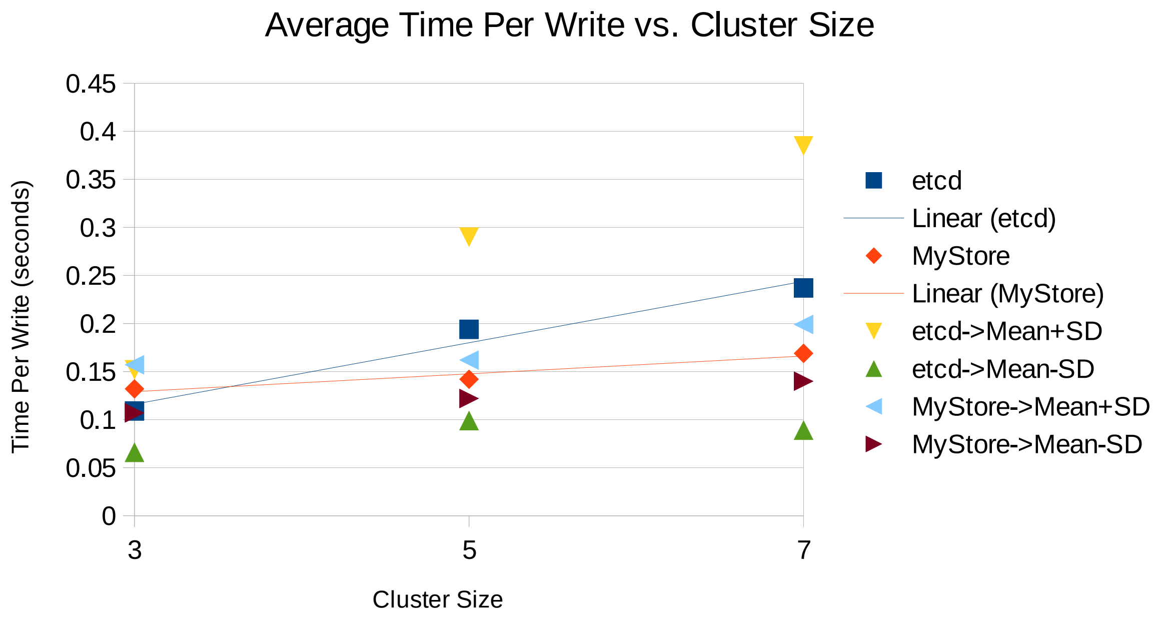
Writes (s)		Reads (s)	
MyStore	etcd	MyStore	etcd
0.101	0.117	0.098	0.044
0.106	0.139	0.108	0.052
0.135	0.164	0.105	0.056
0.124	0.137	0.144	0.064
0.108	0.187	0.102	0.063
0.122	0.191	0.077	0.055
0.136	0.077	0.088	0.045
0.167	0.143	0.088	0.042
0.134	0.109	0.087	0.046
0.109	0.099	0.092	0.038
0.145	0.075	0.097	0.042
0.162	0.069	0.089	0.045
0.196	0.066	0.099	0.056
0.16	0.097	0.097	0.043
0.127	0.074	0.097	0.048
0.124	0.062	0.117	0.039
0.11	0.137	0.092	0.04
0.114	0.067	0.098	0.073
0.128	0.055	0.105	0.05

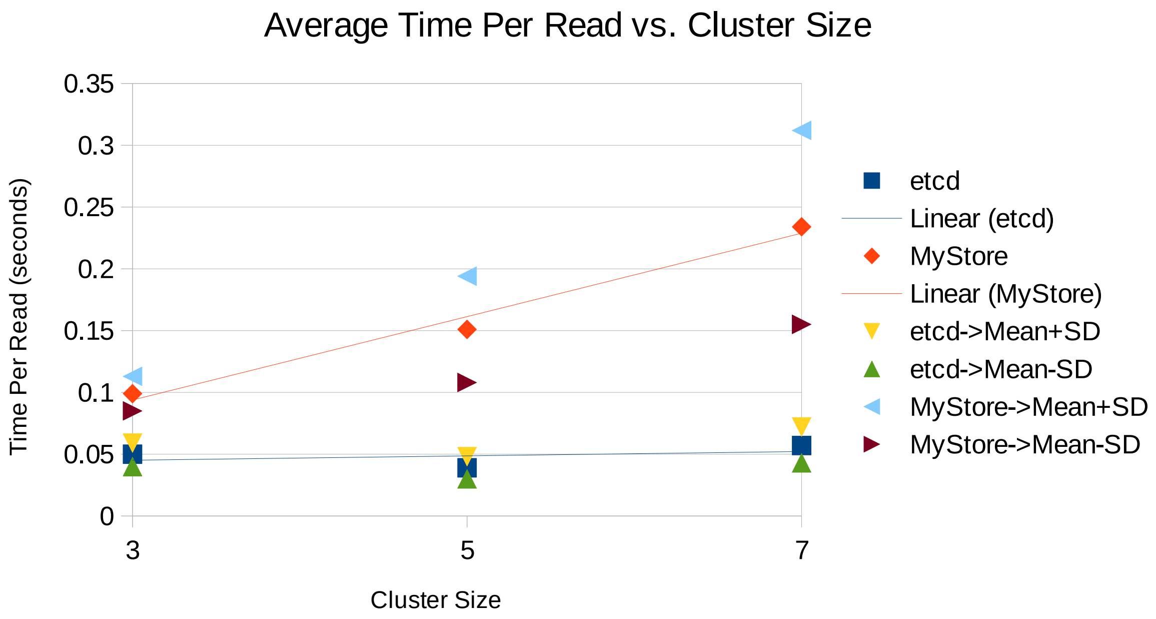
Cluster Size – 5

Writes (s)		Reads (s)	
MyStore	etcd	MyStore	etcd
0.12	0.139	0.101	0.031
0.101	0.195	0.089	0.033
0.117	0.152	0.139	0.034
0.151	0.106	0.159	0.034
0.144	0.278	0.135	0.033
0.125	0.394	0.104	0.04
0.129	0.17	0.202	0.032
0.132	0.139	0.209	0.038
0.144	0.166	0.151	0.041
0.141	0.321	0.149	0.04
0.135	0.224	0.116	0.05
0.174	0.21	0.157	0.038
0.146	0.114	0.224	0.06
0.153	0.116	0.234	0.037
0.139	0.099	0.124	0.04
0.175	0.132	0.152	0.044
0.15	0.406	0.17	0.055
0.147	0.096	0.166	0.034
0.18	0.237	0.089	0.023

Cluster Size – 7

Writes (s)		Reads (s)	
MyStore	etcd	MyStore	etcd
0.118	0.366	0.168	0.054
0.127	0.544	0.185	0.055
0.142	0.373	0.177	0.04
0.14	0.247	0.191	0.063
0.15	0.197	0.178	0.047
0.144	0.205	0.178	0.092
0.213	0.133	0.165	0.051
0.191	0.092	0.154	0.052
0.218	0.63	0.212	0.053
0.165	0.122	0.219	0.047
0.193	0.296	0.212	0.053
0.154	0.152	0.293	0.059
0.151	0.196	0.404	0.049
0.187	0.21	0.227	0.051
0.152	0.126	0.24	0.068
0.182	0.192	0.278	0.053
0.2	0.153	0.264	0.094
0.194	0.186	0.446	0.067
0.195	0.082	0.252	0.044





The overlap of the average time per write for etcd and MyStore seems to indicate that the two have comparable write performance. It appears that etcd has superior read performance relative to MyStore across all cluster sizes used during testing. One reason MyStore may lag behind etcd in read performance is MyStore's reliance on total lock ordering. Another possible cause would be that MyStore is written in Python, which uses a global interpreter lock that prevents multiple threads from utilizing the interpreter simultaneously. Go, the language in which etcd is written, has no such limitation. Both etcd and MyStore seem to exhibit some performance instability as cluster size is increased: the standard deviation on etcd's write performance and MyStore's read performance both grow considerably as cluster size is increased from 3 to 7.

Future work on MyStore will include an overhaul of the locking scheme and a rewrite of most of the code in C++ in order to achieve better performance. The command line tool will likely not be rewritten since its performance is less critical. The log compaction feature mentioned in *In Search of an Understandable Consensus Algorithm* will be implemented to truncate the log so that it does not grow uncontrollably in instances where MyStore is running for long periods of time. To make programmatic interaction with a MyStore cluster easier, a C++ client will eventually be added and then ported over to different languages. The parts of *In Search of an Understandable Consensus Algorithm* related to dynamic cluster configuration updates will be implemented to allow new nodes to be added to a MyStore cluster without having to temporarily take the system offline.