CS 550 Project

An Empirical Evaluation of Distributed Key/Value Storage Systems — Memcached,Rick and Redis

Ji Tin Justin Li A20423037

Subowen Yan A20430537

Yuhan Li A20431466

1. Introduction ZHT

The differences between chosen 3 systems to ZHT

	Latency	Through	Persistence	Scalability	fault tolerance	Impl.	dynamically noes	light-weight key-value
ZHT	Middle	Middle	High	High	Replication mechanism	C++	Yes	Yes
Memcached	Very low	Very large	Low	High	No Replication	Java	No	Yes
Redis	Low	Large	High(disk)/low	High	Replication Master-slave	python	No	Yes
Riak	High	Very low	High	High	Replication mechanism	python	Yes	No

ZHT has excellent availability and fault tolerance, but concurrently achieve the benefits minimal latencies normally associated with idle centralized indexes. The data-structure is kept as simple as possible for ease of analysis and efficient implementation. A Manager is a service running on each physical node and takes charge of starting and shuting down ZHT instances. The manager is also responsible for managing membership table, starting/stopping instances, and partition migration. In each physical node, there is one ZHT manager with multiple ZHT instances. And each instance have multiple Non-Volatile Hash Table (NoVoHT) in order to withstand failures and restarts, it also supports persistence.

2. Memcached

Brief description:

Memcached is a free and open-source, high-performance, distributed memory object caching system. It is designed to speed up dynamic web applications by caching chunks of arbitrary data in a distributed fashion, with the intent of alleviating database load. Memcached allows quick and easy aggregation of memory resources across multiple server nodes, which better utilizes resources on nodes, as well as facilitates system scalability.

Comparison to ZHT:

The 3 major difference between ZHT and memcached is that memcached does not support data persistence, replication, and dynamic membership. Out-of-the-box, Memcached does not provide functionalities that support data persistence - all the data is stored in memory. ZHT, however, supports persistence by providing configurations that periodically checkpoints the system. ZHT also has built-in data replication to enhance data availability, in which primary and secondary replicas are strongly consistent, while other replicas are asynchronously updated following weak consistency. For dynamic membership, memcached server locations has to be specified upon starting a memcached client, and the memcached client handles all the communication with the specified memcached servers. I.e. clients are statically configured, and they do not support the churn of server nodes. On the other hand, ZHT optionally allows dynamic membership, where client and server nodes can interleave without affecting the complexity of basic operations.

3. Riak:

For the Riak system, it states as a distributed, decentralized data storage system. The Riak system, a distributed database is designed to perform maximum data availability by distributing data across multiple servers. It works as a horizontal scale system. Users can easily add or delete additional nodes to a cluster. The ZHT can also achieve dynamically allowing noes to join and leave.

The Riak system uses Bitcask as backends storage algorisms. The keys stores at in-memory hash-table. The values locate on disk, are directly pointed by keys. The shorthand will be that the the Riak system cannot handle an extremely large number of keys. However, the ZHT system use light-weight key-value stores. It will outperform the RIak systems. The ZHT system fault-tolerant through replication and by handling failures gracefully and efficiently propagating events throughout the system. Due to the importance of fault tolerance, The ZHT uses replication mechanism. As well, the replication default setting for the Riak system is three. It means that, for one data enter, the system will store the same data three times across the system, which would increase insert, lookup or remove time. The ZHT systems support unconventional operations such as append (providing lock-free concurrent key/value modifications) in addition to insert/lookup/remove. On the contrary, the Riak systems use a lock

when doing insert, lookup or remove.

4. Redis

Redis is famous for its quick speed, its ability to store data types such as lists or sets, and because it offers persistence, unlike other in-memory key value stores.

Redis works with an in-memory dataset. Depending on our use case, we can persist it either by dumping the dataset to disk every once in a while, or by appending each command to a log. In this condition we plan to disable the persistence to get a feature-rich, highspeed, in-memory cache. Example: *save " " # save 900 1 # save 300 10 # save 60 10000*

So it is very flexible to choose if users want to save the data on disk or not, as well as decided how often save it once.

How the Redis cluster work

Redis Cluster is an active-passive cluster implementation that consists of master and slave nodes. The cluster uses hash partitioning to split the key space into 16,384 key slots, with each master responsible for a subset of those slots. Each slave replicates a specific master and can be reassigned to replicate another master or be elected to a master node as needed. Replication is completely asynchronous and does not block the master or the slave. Masters receive all read and write requests for their slots; slaves never have communication with clients.

Comparison to ZHT:

I think ZHT is really similar to redis Storage Systems, for they have similar throughput and latency. However, with the increase nodes, the ZHT will have a better performance. They both have fault-tolerance mechanism that Replication mechanism.ZHT gracefully handle failures, by lazily tagging nodes that do not respond to requests repeatedly as failed. ZHT uses replication to ensure data will persist in face of failures. Newly created data will be proactively replicated asynchronously to nodes in close proximity of the original hashed location. The redis use master and slave so it can distribute both on other nodes or local.

5. Data form

Throughput vs. Scale (operations per second)

System\Scale	1	2	4	8
ZHT (Opts/s)	4117	5524	9813	18680
Memcached (paper)	7385	7961	14480	40995

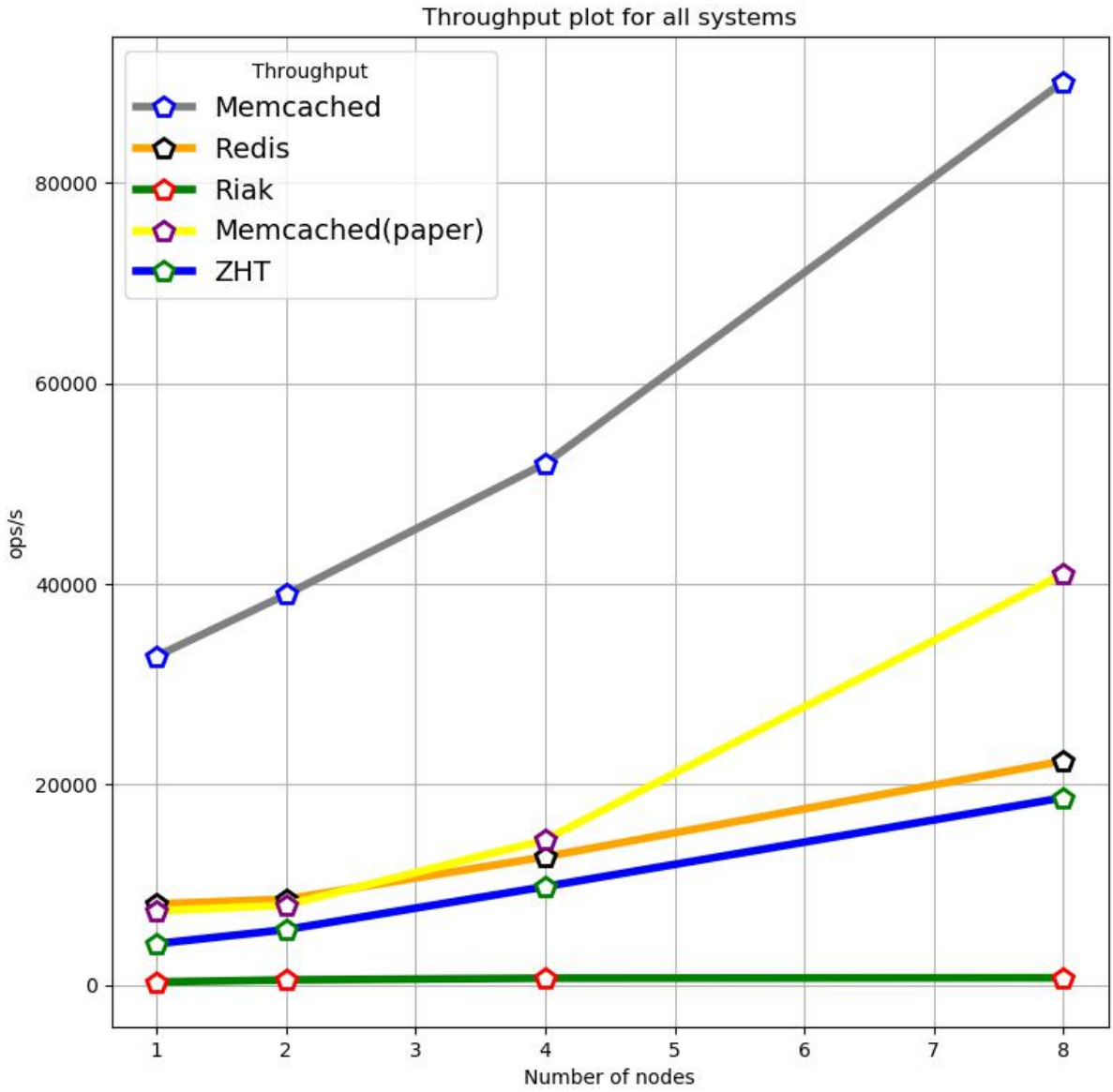
Memcached	32794	38938	51998	90099
Redis	8088	8558	12792	22336
Riak	309	507	683	723

Latency vs. Scale (milliseconds per operation)

System\Scale	1	2	4	8
ZHT (ms)	0.243	0.362	0.408	0.428
Memcached (paper)	0.122	0.324	0.272	0.278
Memcached	0.0301	0.0584	0.0698	0.0908
Redis	0.1331	0.2560	0.3232	0.3706
Riak	3.182	3.771	5.843	10.516

6. Conclusion of graph

The following plot shows the latency (microsecond) vs scale:

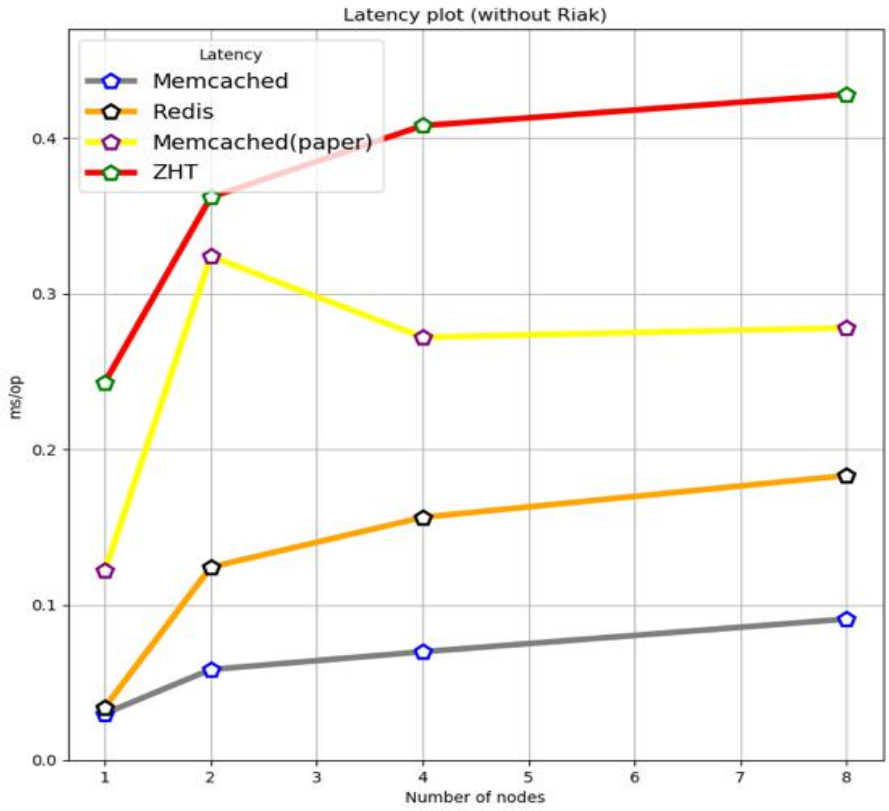


As we can see, latency of the Riak system is definitely the worst one. It highly causes by using Bitcask as backends storage algorithms. The keys stores at in-memory hash-table. The values located on disk, are directly pointed by keys.

For memcached, our experiments were comparable to the results as suggested in the paper, where our plots follows the shape of the paper's plot, with an upscaled-factor of approximately 2-2.5x, for the scaling from 1 node to 8 nodes.

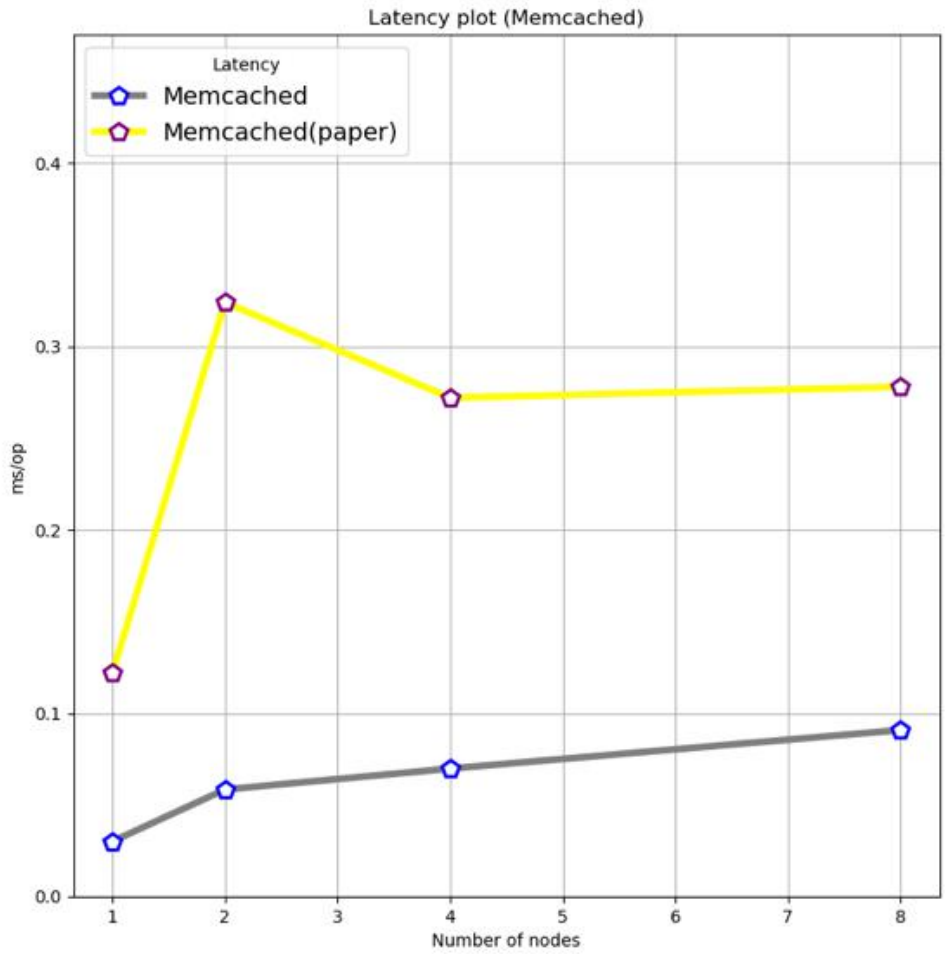
The redis have the similar throughput with Memcached(paper).

Without the Riak system, the following plot shows the latency (microsecond) vs scale:



Below is a comparison latency plot for memcached as given in the paper verses our experiment results.

On the single nodes the Redis and Memcached have the same low latency. The ZHT latency is highest except Riak.



There are a few reasons that may contribute to this difference:

(1) Hardware difference: HEC-Cluster vs ChameleonHardware Diff:

A.HEC-Cluster nodes - dual processor quad-core, 8GB RAM

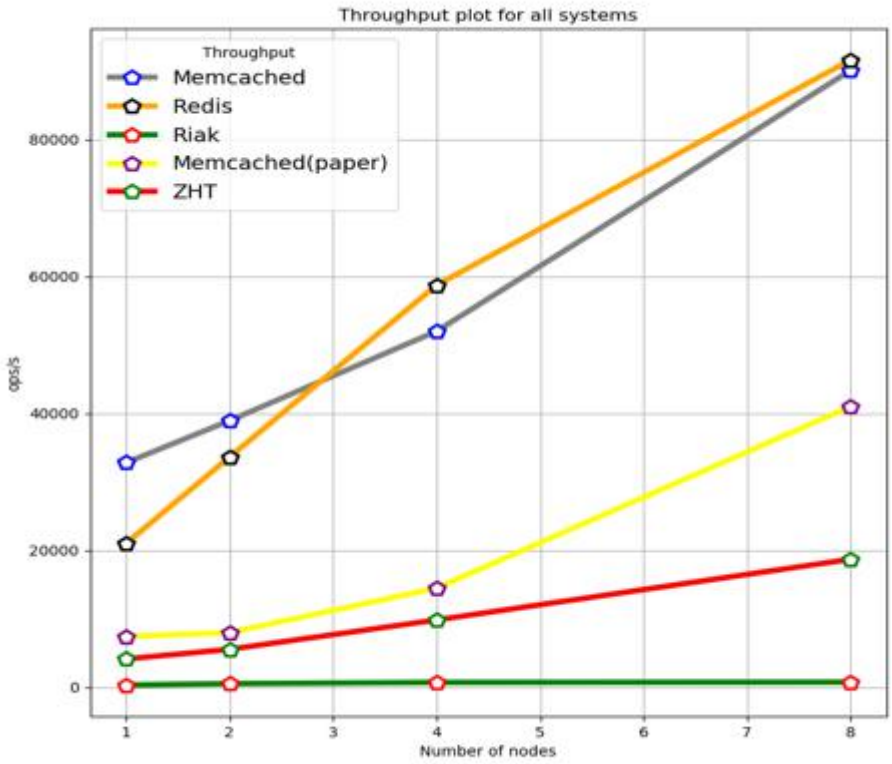
B.m1.medium instances - 2 vcpus, 4GB RAM

(2) Software:

A.Memcached server configs was not specified in the paper. Our experiments uses the default configurations for server nodes.

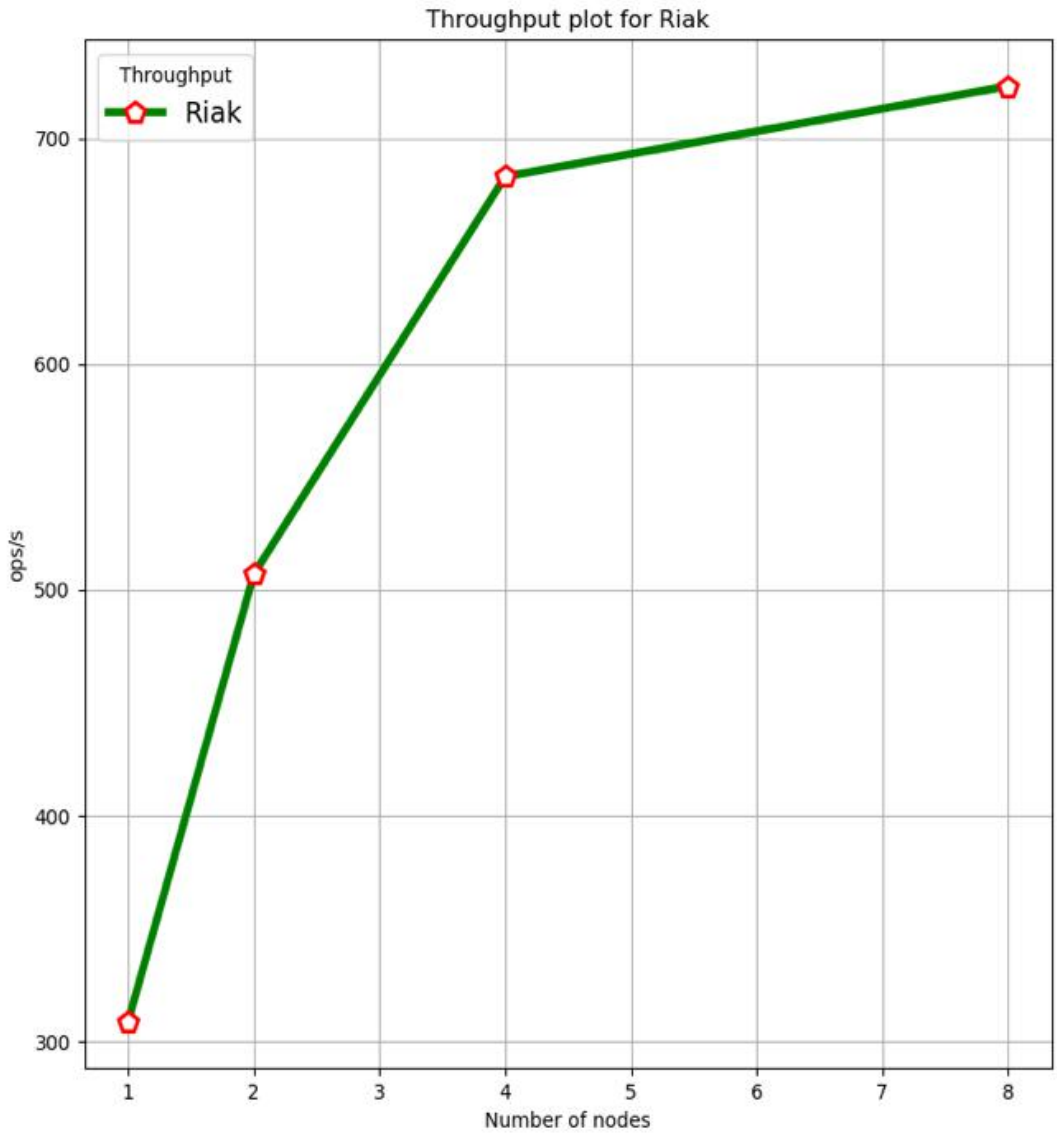
B.Memcached client API: The paper did not specify which memcached client library was used in the experiments. In our experiment, we used the open-source spymemcached (which is implemented in Java).

The following plot shows the throughput (job per second) vs scale:



As we can see, the Riak system definitely has the worst throughput. The worst throughput corresponds to high latency.

The following plot shows the throughput (job per second) vs scale for Riak:



As I mentioned earlier, the Riak system cannot handle an extremely large number of keys. From the graph, we know that as workloads increase, the throughput shows diminishing returns.