Distributed Key-Value Store on HPC and Cloud Systems

Tonglin Li[1], Xiaobing Zhou1, Kevin Brandstatter[1], Ioan Raicu[1,2]
Department of Computer Science, Illinois Institute of Technology[1]
Mathematics and Computer Science Division, Argonne National Laboratory[2]

ABSTRACT
ZHT is a zero-hop distributed hash table, which has been tuned for the requirements of high-end computing systems. ZHT aims to be a building block for future distributed systems. The goals of ZHT are delivering high availability, good fault tolerance, high throughput, and low latencies, at extreme scales of millions of nodes. ZHT has some important properties, such as being light-weight, dynamically allowing nodes to join and leave, fault tolerant through replications, persistent, scalable, and supporting unconventional operations such as append. ZHT scaled up to 32K-cores with latencies of 1.1ms and 18M operations/sec throughput on IBM Blue Gene/P supercomputer, and 96 nodes on Amazon EC2 cloud with 800ns latency and 1.2M ops/s throughput. In previous work we proved ZHT's excellent performance and scalability on supercomputers, and in this work we show that it also works great on cloud environment from both performance and cost perspective.

General Terms
Management, Measurement, Performance, Design, Reliability, Experimentation.

Keywords
Distributed Key-Value store, Distributed Gash Table, High-End Computing, Cloud Computing.

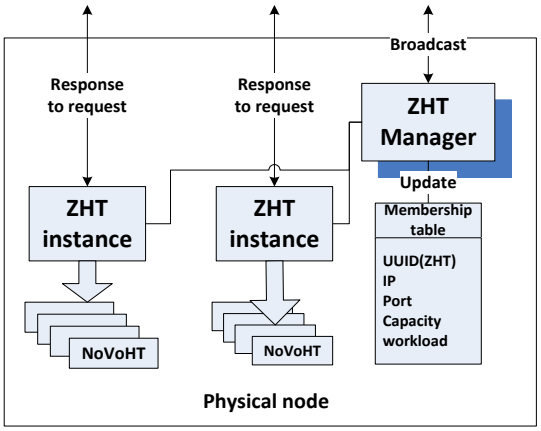
1. INTRODUCTION
This work presents a zero-hop distributed hash table (ZHT), which has been tuned for the specific requirements of high-end computing (e.g. trustworthy/reliable hardware, fast networks, non-existent "churn", low latencies, and scientific computing data-access patterns). ZHT aims to be a building block for future distributed systems, with the goal of delivering excellent availability, fault tolerance, high throughput, scalability, persistence, and low latencies. ZHT has several important features making it a better candidate than other distributed hash tables and key-value stores, such as being light-weight, dynamically allowing nodes join and leave, fault tolerant through replication and by handling failures gracefully and efficiently propagating events throughout the system, a customizable consistent hashing function, supporting persistence for better recoverability in case of faults, scalable, and supporting unconventional operations such as append (providing lock-free concurrent key/value modifications) in addition to insert/lookup/remove. To provide ZHT a persistent back end, we also created a fast persistent key-value store that could be easily integrated and operated in lightweight Linux OS typically found on today's supercomputers.

We have evaluated ZHT's performance under a variety of systems, ranging from a Linux cluster with 512-cores, to an IBM Blue Gene/P supercomputer with 160K-cores. Using micro-benchmarks, we scaled ZHT up to 32K-cores with latencies of only 1.1ms and 18M operations/sec throughput. We compared ZHT against two other systems, Cassandra and Memcached and found it to offer superior performance for the features and portability it supports, at large scales up to 16K-nodes. We also conducted experiments on Amazon EC2 cloud to compare ZHT against Amazon DynamoDB on up to 96-nodes scale, in both performance and economical perspective.

2. Design and Implementation
The primary goal of ZHT is to get all the benefits of DHTs, namely 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.

The application programming interface (API) of ZHT is kept simple and follows similar interfaces for hash tables. The four operations ZHT supports are 1. int insert(key, value); 2. value lookup(key); 3. int remove(key), and 4. int append(key, value). Keys are typically a variable length ASCII text string. Values can be complex objects, with varying size, number of elements, and types of elements. Integer return values return 0 for a successful operation, or a non-zero return code that includes information about the error that occurred.

In static membership, every node at bootstrap time has all information about how to contact every other node in ZHT. In a dynamic environment, nodes may join (for system performance enhancement) and leave (node failure or scheduled maintenance) any time, although in HEC systems this "churn" occurs much less frequently than in traditional DHTs. ID Space and Membership Table are put in a ring-shaped key name space. The node ids in ZHT can be randomly distributed throughout the network, or they can be closely correlated with the network distance between nodes. The correlation can generally be computed from information such as MPI rank or IP address. The random distribution of the ID space has worked well up to 32K-cores, but we will explore a network aware topology in future work.



The hash function maps an arbitrarily long string to an index value, which can then be used to efficiently retrieve the communication address (e.g. host name, IP address, MPI-rank) from a membership table (a local in-memory vector). Depending

on the level of information that is stored (e.g. IP - 4 bytes, name - <100 bytes, socket - depends on buffer size), storing the entire membership table should consume only a small (less than 1%) portion of available memory of each node. On 1K-nodes scale, one ZHT instance has a memory footprint of only 10MB (from an available 2GB memory), achieving our desired sub 1% memory footprint. The memory footprint consists of ZHT server binary in memory, entries in hash table, membership table and ZHT server side socket connection buffers. Among them, only membership table and socket buffers will increase with the scale of nodes. Entries in hash table will be flushed to disk finally. But membership is very small, it takes 32 bytes per entry (for each node), 1million nodes only need 32MB memory. By tuning the number of KeyValue pairs that are allowed stay in memory, users can achieve the balance between performance and memory consumption.

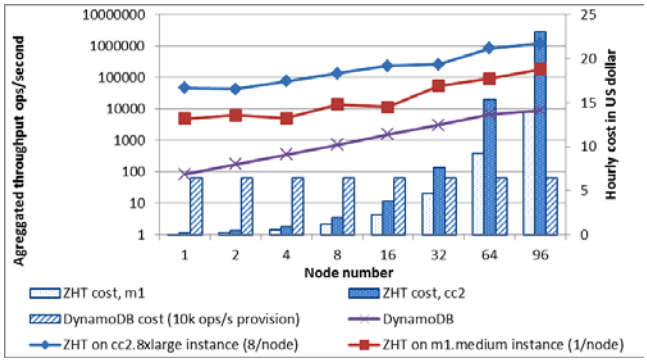
3. Evaluation

3.1 Performance and running cost

We conduct micro benchmark on Amazon EC2 cloud as well to compare against Amazon DynamoDB. The EC2 instance type we used are m1.medium and cc2.8xlarge, the details are below.

Since the interference between m1.medium instances, ZHT shows mild fluctuation in throughput. On 2cc.8xlarge instances, the fluctuation closes to disappear and the throughput close to be linear. Although DynamoDB seems to stay with a linear growth, the absolute throughput is quite low. Comparing with ZHT, DynamoDB was more than 20 times slower at all scales.

For different EC2 instance types, we tried with various numbers of ZHT servers and clients on each instance so as to explore the aggregated throughput. In our experiments, on larger instance type such 2cc.8xlarge, running multiple ZHT server/client won't influence latency. Thus the aggregated throughput may have a linear growth as long as there is still CPU and network bandwidth resource. On 96 nodes scale with 2cc.8xlarge instance type, ZHT offers 1215.0 K ops/s while DynamoDB failed the test since it saturated the capacity. The measured maximum throughput of DynamoDB is 11.5K ops/s which is found at 64 node scale. For a fair comparison, both DynamoDB and ZHT have 8 clients per node.



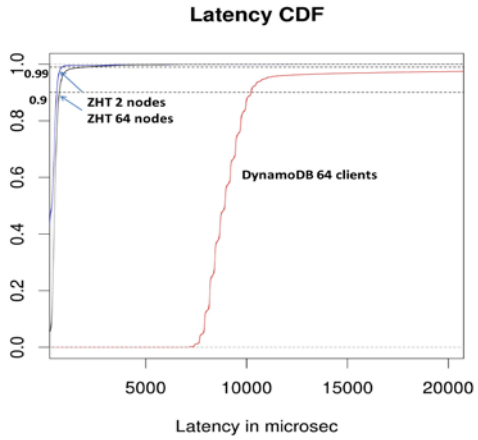
It's worth noting that DynamoDB has a maximum throughput which is provisioned (namely capacity) by the users. When the throughput is beyond provisioned capacity, DyndmoDB will saturate and give errors, requests start to fail.

When discussing cloud, the cost is always a big concern. We calculated hourly cost for both ZHT and DynamoDB on different scales. Since DynamoDB has a fixed cost, the average cost reduces with the client increasing. On 2-node scale DynamoDB cost 423 times more than ZHT; on largest scale that DynamoDB can support, it still cost 9 times more than ZHT for a same throughput. Note the cost for DynamoDB doesn't include the EC2 instances for running clients, it will cost even more if include the client cost.

3.2 Latency distribution

As expected, DynamoDB has much longer latency on all scales. On 4-node (32 clients) scale it is 22 times slower than ZHT. In the CDF comparison DynamoDB shows that its 90% latencies fall into a 20x wider time window than ZHT. When we ran 8 clients on 64 nodes, DynamoDB started to give errors which complain that we used too much throughput so we can't continue to run the benchmarks on larger scales. The slowest 5% requests latency increased by 3 times.

It is worth noting that DynamoDB latencies don't vary much with the system scales. It seems to show an excellent scalability and a better aggregated throughput. However considering that Amazon only guarantees the maximum throughput, instead of latency, users won't get faster response when they only use low throughput. In other words, DynamoDB with more clients doesn't work as fast as it with fewer clients; instead, with fewer clients it works as slow as with many clients. This characteristic prevents the users from reaching the provisioned capacity by lowering down the latency when they only have fewer clients.



4. Conclusion

ZHT has shown excellent performance and scalability. It's been used as building blocks of several distributed systems. Beside being highly effective on HPC environment, it also shows versatility on commercial cloud. ZHT is more than 20 times faster than Amazon DynamoDB while costing less than 1/10 of the premium (spent on running VMs), which make it a great candidate for both a building block of distributed HPC systems and a general-purpose key-value store on cloud.

5. REFERENCES

[1] Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang, Ioan Raicu. "ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table", IEEE IPDPS, 2013

[2] Tonglin Li, Raman Verma, Xi Duan, Hui Jin, Ioan Raicu. "Exploring Distributed Hash Tables in High-End Computing", ACM Performance Evaluation Review (PER), 2011