ZHT: a Zero-hop DHT for High-End Computing Environment

Tonglin Li[1], Antonio Perez de Tejada[1], Kevin Brandstatter[1], Zhao Zhang[3], Ioan Raicu[1,2]
Department of Computer Science, Illinois Institute of Technology[1]
Mathematics and Computer Science Division, Argonne National Laboratory[2]
Department of Computer Science, University of Chicago[3]

ABSTRACT

One critical component of future file systems for high-end computing is meta-data management. This work presents ZHT, a zero-hop distributed hash table, which has been tuned for the requirements of HEC systems. ZHT aims to be a building block for future distributed file systems to implement distributed metadata management. The goals are delivering availability, fault tolerance, high throughput, and low latencies. ZHT has some important properties, such as being light-weight, fault tolerant using replication and persistence. We have evaluated ZHT's performance under a variety of systems, ranging from a Linux cluster to an IBM BlueGene/P supercomputer. We scaled ZHT up to 16K processes and achieved 4M operations/sec throughput. Latencies have scaled similarly well, with sub-milliseconds latencies at 4K-core scales. We compared ZHT against other systems and found it offers superior performance for the features and portability it supports.

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 file systems, with the goal of delivering excellent availability, fault tolerance, high throughput, and low latencies. ZHT has several important features making it a better candidate than other distributed hash tables, such as being light-weight, fault tolerant by handling failures gracefully and efficiently propagating events throughout the system, a customizable consistent hashing function, supporting replication to guard against data loss, and supporting persistence for better recoverability in case of faults. We have evaluated ZHT's performance under a variety of systems, ranging from a modest 64-node Linux cluster to a 1024-node IBM BlueGene/P supercomputer with up 16K ZHT instances. We compared ZHT against two other systems, Cassandra [37] and Memcached [20] and found it to offer superior performance for the features and portability it supports at modest scales of thousands of nodes.

The contributions of this work are as follows:

- Design and implementation of ZHT, a light-weight, high performance, fault tolerant, persistent, and highly scalable distributed key-value store, optimized for high-end computing.
- Zero-hop routing with constant time properties, achieving latencies between 0.5ms~0.7ms at up to 1K nodes scales.Low cost asynchronous replication for low overhead fault tolerance.
- Performance evaluation at up to 4K-cores and 16K instances comparing ZHT to Memcached and Cassandra, on many test beds, including clusters and supercomputers, up to 16K-processes, and achieving 4 millions of operations/sec throughput.

2. Design and Implementation

The primary goal of ZHT is to get all the benefits of distributed hash tables, namely excellent availability and fault tolerance, but concurrently achieve the benefits of a centralized index where latencies are minimal. The data-structure is kept as simple as possible for ease of analysis and efficient implementation. In a static membership, every node at bootstrap time has all information about how to contact every other node in ZHT, which is a valid assumption because of the batch-scheduled HEC environment. ID Space and Membership Table Figure 2: ZHT architecture design, in the ring-shaped key name space, replicas are set to each nodes’ neighbors. 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 and IP address. Take BlueGene/P as an example, each node in BGP has a coordinate (x,y,z) within the allocation range(X, Y, Z), the rank of the node is computed as z*X*Y+y*X+x, the ip address is 12.x.y.z+1. Given the range(X, Y, Z) and any one of the coordinates, ip or rank, we could compute the other two. In the case of the network correlated node ID space, nodes can make decisions based on some distance metrics to determine the closest node to communicate with. This network topology aware approach is critical to making ZHT scalable by ensuring that communication is kept localized when performing 1-to-1 communication. For efficient 1-many communication, we have adopted a minimum spanning tree approach which spreads the communication load across all nodes with minimal latencies (expected to be logarithmic in the number of nodes).

The hash function maps an arbitrarily long string directly to an index value, which can then be used to efficiently retrieve the communication address (e.g. host name, IP address, MPIrank) 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 1024 nodes scale, ZHT has a memory foot print of only 15MB. By tuning the number of Key-Value pairs that are allowed stay in memory, user can achieve the balance between performance and memory consumption.

ZHT uses event-driven model server architecture. The current version ZHT has an epoll-based single thread server, but works 3 times faster than the previous version which used multithreading. We'll discuss the performance difference between these two architectures in evaluation section.

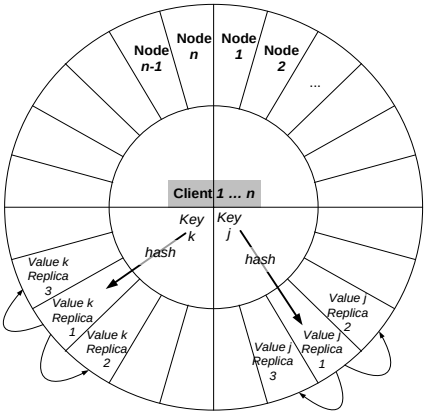
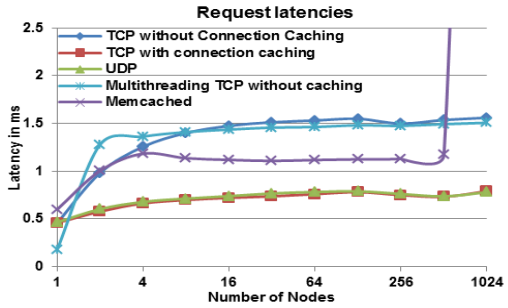


Figure 1: ZHT architecture design

3. Performance Evaluation

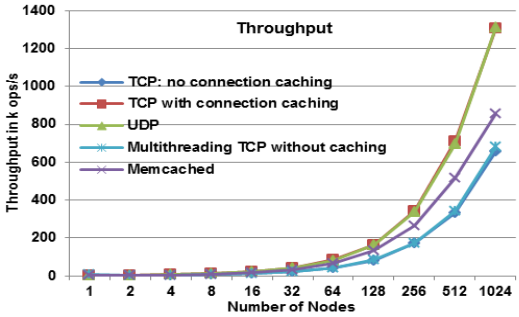
3.1 Latency

We extensively tested ZHT on BlueGene/P supercomputer. On 1024-node scale, ZHT shows great scalability. As shown in the Figure 7, on one node, the latency of both TCP with connection caching and UDP are extremely low. When scaling up, ZHT shows nearly constant latency, almost all within 0.75 ms, even at scale of 1024 nodes, the average latency is still 0.79 ms. By this observation, we conclude that the major cost of TCP is to create and close connections. With all the connections keep open, TCP can work as fast as UDP. The implementation with standard TCP, on the other hand, takes 2x more time to finish the same operations. An old version of ZHT which implemented with multiple thread and standard TCP shows the same performance as event-driven version. As a comparison, Memcached also shows low latency within 1.2 ms until 1024 nodes. It drastically increases the insert latency to 47ms. After several run of experiments, we're sure it's not an exception.



3.2 Throughput

We conducted several experiments to measure the throughput of ZHT as well as Memcached and Cassandra. The throughputs of ZHT(TCP with connection caching and UDP) in operations per second increases near-linearly with scale, reaching 1.3M ops/sec at 1024 nodes while Memcached can reach 900K ops/sec; Figure 9 shows the comparison between different implementation of ZHT and Memcached throughput.



Because of the importance of fault tolerance, ZHT ses replication mechanism. It will certainly introduce some overhead. As shown in Figure 13, replication does increase the operation latency, but it is not a significant increase. 1 replica adds around 20% and 2 replica adds around 30% overhead compare with no replica latency. It is worth noting that the choice of doing the replicas asynchronously likely helped keep the overheads low.

4. Conclusion

ZHT optimized for high-end computing systems is architected and implemented as a foundation in the development of fault-tolerant, high-performance, and scalable storage systems. We performed an extensive performance evaluation of ZHT on a modest scale up to 1K nodes and 16K instances on an IBM BlueGene/P.

We achieved more than 4M operations/sec throughput. The latency is as low as 0.78ms at 1K node scale. We hope to extend the performance evaluation to significantly larger scales, as the machine we tested on has 40K nodes. We believe that ZHT could transform the architecture of future storage systems in HEC, and open the door to a much broader class of applications that would have normally not been tractable. Furthermore, the concepts, data-structures, algorithms, and implementations that underpin these ideas in resource management at the largest scales can be applied to new emerging paradigms, such as Cloud Computing. The work presented in this paper addresses the fundamental technical challenges that will become increasingly harder to address with existing solutions due to a declining MTTF of future HEC systems. Our work will benefit the "Many-Task Computing" paradigm that bridges the gap between highthroughput computing and high-performance computing, generally producing both compute-intensive and data-intensive workloads, and has been shown to contain a large set of scientific computing applications from many domains.

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

[2] Cassandra http://cassandra.apache.org/, 2012

[3] B. Fitzpatrick. "Distributed caching with Memcached." Linux Journal, 2004(124):5, 2004