ISSN (Online) 2456 -1304

International Journal of Science, Engineering and Management (IJSEM)
Vol 2, Issue 12, December 2017

Zero hop Distributed Hash Table

[1] Sruthi Lakshmi.G, [2] Saranya.J, [3]Berlin Femina.J
[1][2][3] III,B.SC Computer Science, Holycross Home Science College,Thoothukudi,TamilNadu,India

Abstract:-- **Hash tables use more structured key-based routing in order to attain both the decentralization of Freenet and gnutella, and the efficiency and guaranteed results of Napsterlevels, and in future comes with accurate measure of delivering millions of nodes and billions of threads of execution. A distributed hash Distributed table (DHT) is a class of a Storage decentralized distributed system that provides a lookup service similar to ahash table: (key, value) pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. DHTs form an infrastructure that can be used to build more complex services, such as anycast, cooperative systems. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures. Distributed hash tables use a more structured key-based routing in order to attain both the decentralization of Freenet and gnutella, and the efficiency and guaranteed results of Napster. The primary goal of ZHT is to provide excellent storage systems have availability, fault tolerance, high throughput, and low latencies.**

Keywords – Distributed hash tables, key/value stores, high-end computing

INTRODUCTION

Exascale computers (e.g. capable of 1018 ops/sec) [1], with a processing capability similar to that of the human brain, will enable the unraveling of significant scientific mysteries and present new challenges and opportunities. Major scientific opportunities arise in many fields (such as weather modeling, understanding global warming, national security, drug discovery, and economics) and may rely on revolutionary advances that will enable exascale computing.
In the current decades-old architecture of HPC systems, storage is completely segregated from the compute resources and are connected via a network interconnect (e.g. parallel file systems running on network attached storage, such as GPFS [2], PVFS [3], and Lustre [4]). This approach is not able to scale several orders of magnitude in terms of concurrency and throughput, and will thus prevent the move from petascale to exascale. If we do not solve the storage problem with new storage architectures, it could be a "show-stopper" in building exascale systems. The need for building efficient and scalable distributed storage for high-end computing (HEC) systems that will scale three to four orders of magnitude is on the horizon. Other distributed file systems (e.g. Google's GFS [7] and Yahoo's HDFS [8]) that have centralized metadata management make the problems observed with GPFS even worse from the scalability perspective.

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.

FEATURES:

ZHT has several important features 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.

ADVANTAGES:

• Design and implementation of ZHT, a light-weight, high performance, fault tolerant, persistent, dynamic, and highly scalable distributed hash table, optimized for high-end computing.
• Support for unconventional operations, such as append allowing data to be incrementally added to an existing value, delivering lock-free concurrent modification on key/value pairs.
• Micro-benchmarks up to 32K-core scales, achieving latencies of 1.1ms and throughput of 18M ops/sec.
• Integration and evaluation with three real systems (FusionFS, IStore, and MATRIX), managing distributed storage metadata and distributed job scheduling information.

DISADVANTAGES:

The distributed metadata management in GPFS does not have enough degree of distribution, and not enough emphasis was placed on avoiding lock contention. GPFS's metadata performance degrades rapidly under concurrent operations,

ISSN (Online) 2456 -1304

International Journal of Science, Engineering and Management (IJSEM)
Vol 2, Issue 12, December 2017

reaching saturation at only 4 to 32 core scales (on a 160K-core machine).

ZHT DESIGN AND IMPLEMENTATION:

1. OVERVIEW:

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.

2. SERVER ARCHITECTURE:

We explored various architectures for ZHT server. Since typical Key-Value store operations are very short but frequent, we designed ZHT to be able to respond fast with little resource consumption. In early prototypes, we explored a multithreading design, in which each request had a separate thread, but the overheads of starting, managing, and stopping threads was too high in comparison to work each thread was performing. We eventually converged on a much more streamlined architecture, an event-driven model server architecture based on epoll. The current epoll-based ZHT outperforms the multithread version 3X.

3. HASHING FUNCTIONS:

There are many good hashing functions in practice [31]. Each hashing function has a set of properties and designed goals, such as:

1) minimize the number of collisions
2) distribute signatures uniformly
3) have an avalanche effect ensuring output varies widely from small input change, and
4) detect permutations on data order. Hash functions such as the Bob Jenkins' hash function, FNV hash functions, the SHA hash family, or the MD hash family all exhibit the above properties [32, 33].

We have explored the use of Bob Jenkins' and FNV hash functions, due to their relatively simple implementation, consistency across different data types (especially strings), and the promise of efficient performance [49].

4.LIGHT-WEIGHT1-1 COMMUNICATION:

We implemented ZHT with both TCP (with server returned result state) and UDP (acknowledge message based, which means every time a message is sent, the sender is waiting for an acknowledge message) protocols. In previous work [14], we showed that UDP offered some performance advantage at modest scales of nearly 6K cores. We anticipate that UDP's advantages will become more prevalent with even larger scales as connectionless communication protocols will be preferred to avoid having expensive connection establishments among many nodes. In ZHT, we implemented a LRU cache for TCP connections, which makes TCP works almost as fast as UDP does. We expect to extend the communication protocols in future iterations of ZHT, such as BMI [41], and perhaps even MPI if we are willing to sacrifice fault tolerance for potentially improved performance and accessibility to certain HEC systems that do not support the IP protocol.

5.COMPLEX STRUCTURES SUPPORT

In order to support complex structures as values in ZHT, we adopted the Google protocol buffer [37] project, which serializes complex structures into a stream of bytes. The indicators for four basic operations (insert, lookup, remove, and append) are defined in the message prototype and compiled with Google Protocol Buffers. They are encapsulated with the key-value pair into a plain string and transferred through network. When a server receives a request, it just unpacks the message, read the indicator and execute the operation request.

ZHT gracefully handle failures, by lazily tagging nodes that do not respond to requests repeatedly as failed (using exponential back off). 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 (according to the UUID) of the original hashed location. By communicating only with neighbors in close proximity, this approach will ensure that replicas consume the least amount of shared network resources when we succeed in implementing the network-aware topology (see future work section). Despite the lack of network-aware topology in the current ZHT, the asynchronous nature of the replication adds relatively little overhead with increasing numbers of replicas at modest scales up to 4K-cores. ZHT is completely distributed, and the failure of a single node does not affect ZHT as a whole. The (key, value) pairs that were stored on the failed node were replicated on other nodes in response to the failure, and queries asking for data that were on the failed node will be answered by the replicas.

ISSN (Online) 2456 -1304

International Journal of Science, Engineering and Management (IJSEM)
Vol 2, Issue 12, December 2017

6. PERSISTENCE:

ZHT is a distributed in-memory data-structure. In order to withstand failures and restarts, it also supports persistence. We evaluated several existing systems, such as KyotoCabinet HashDB [39] and BerkeleyDB, but low performance and missing features prompted us to implement our own solution. We designed and implemented a Non-Volatile Hash Table (NoVoHT) which uses a log-based persistence mechanism with periodic checkpointing. NoVoHT was designed to address several limitations of KyotoCabinet, specifically to enable specifying a size (to control memory footprint), re-size rate (how often to increase or decrease the size of the table), and garbage collection (how often to reclaim unused space on persistent storage). Since all key-value pairs are kept in memory, it lends itself to low latency in lookups when compared to other persistent hash maps such as KyotoCabinet's HashDB[39], which are disk-based and any lookup must hit disk.

7. CONSISTENCY:

ZHT uses replication to enhance reliability. Replicas have distinct ordering in terms of which ones are accessed by 0 50 100 150 200 250 64 128 256 512 1024 2048 4096 8192 Time (sec) Number of Nodes ZHT bootstrap time ZHT Server start Generate neighbor list BGP partition boot clients. This means that clients will generally be interacting with a single replica (e.g. primary replica), and consistency is straightforward to be maintained, at the cost of potential loss of performance advantages if we allowed multiple replicas to be concurrently modified.In the event that the primary replica becomes temporarily inaccessible, a secondary replica will interact directly with clients (which would cause modifications to happen concurrently on both the primary and secondary replicas). The ZHT primary replica and secondary replica are strongly consistent, other replicas are asynchronously updated after the secondary replica is complete, causing ZHT to follow a weak consistency model. Using this approach, ZHT achieves high throughput and availability while maintains reasonable consistency level.

8. IMPLEMENTATION:

ZHT has been under development for 2 years with 4.5 years of man-hours. It is implemented in C/C++, and has very few dependencies. It consists of 6700 lines of code, and is an open source project accessible at [45]. The dependencies of ZHT are NoVoHT and Google Protocol Buffers [37]. NoVoHT itself has no dependencies other than a modern gcc compiler.

APPLICATIONS:

This work provides three real systems that have integrated with ZHT, and evaluates them at modest scales.

1) ZHT was used in the FusionFS distributed file system to deliver distributed meta-data management at over 60K operations (e.g. file create) per second at 2K-core scales.

2) ZHT was used in the IStore [50, 65], an information dispersal algorithm enabled distributed object storage system, to manage chunk locations delivering more than 500 chunks/sec at 32-nodes scales.

3) ZHT was also used as a building block to MATRIX, a distributed job scheduling system, delivering 5000 jobs/sec throughputs at 2K-core scales.

PERFORMANCE EVALUATIONVIA MICROBENCHMARKS:

In this section, we describe the performance of ZHT, including hashing functions, persistence, throughput, latencies, and replication.

1. NOVOHT PERSISTENCE:

We compared NoVoHT with persistence to KyotoCabinet with identical workloads for 1M, 10M, and 100M inserts, gets, and removes, operating on fixed length key value pairs. The results (see Figure 6) show NoVoHT scales nearly perfect in terms of time per operation; experiments not shown in this figure also show that memory overheads follow the same near perfect trends. It is interesting to note that persistency of writing key/value pairs to disk only adds about 3us of latency on top of the in-memory implementation. Figure 6: Performance evaluation of NoVoHT, KyotoCabinet and BerkeleyDB plotting latency vs. scale (1M to 100 million key/value pairs) on Fusion When comparing NoVoHT with KyotoCabinet or BerkeleyDB, we see much better scalability properties for NoVoHT. Although BerkeleyDB has some advantages such as memory usage (not shown in the figure), it does this at the cost of performance. 0 5 10 15 20 1 million 10 million 100 million Latency (microseconds) Scale(number of key/value pairs) NoVoHT NoVoHT (No persistence) KyotoCabinet BerkeleyDB unordered_map C

2. LATENCIES:

We evaluated the latency metric on both the Blue Gene/P and HEC-Cluster testbeds. We evaluated several communication variations, such as UDP/IP, TCP/IP without connection caching, TCP/IP with connection caching, and compare them with Memcached and Cassandra. Performance evaluation of ZHT and Memcached plotting latency vs. scale (1 to 8K nodes on the Blue Gene/P) At 8K-node scale, ZHT shows great scalability.

ISSN (Online) 2456 -1304

International Journal of Science, Engineering and Management (IJSEM)
Vol 2, Issue 12, December 2017

When scaling up, ZHT shows low latency, up to 1.1ms at 8K-node scales. We see that TCP with connection caching can deliver essentially the same performance as UDP, for all the scales measured. Memcached also scaled well, with latencies ranging from 1.1ms to 1.4ms from 1 node to 8K nodes (note that this represents a 25% to 139% slower latency, depending on the scale). Note the IBM Blue Gene/P network for communication is a 3D Torus network, which does multi-hop routing to send messages among compute nodes. That means the number of hops will increase when communicate across racks. This explains the performance slow down on large scale, since one rack of Blue Gene/P has 1024 nodes, any larger scale than 1024 will involve more than one rack. We found the network to scale very well up to 32K-cores, but there is not much we can do about the multi-hop overheads across rack.

Because of Cassandra's implementation in Java, and the lack of support for Java on the Blue Gene/P, we evaluated Cassandra, Memcached, and ZHT on the HEC-Cluster (a traditional Linux cluster). Not surprisingly, as shown in Figure 8, ZHT has much lower latency than Cassandra. ZHT also shows superior scalability over Cassandra. This is mainly because Cassandra has to take care of a logarithmic-routingtime dynamic member list and ZHT use constant routing. Surprisingly, Memcached only shows slightly better performance than ZHT up to 64-node scales. We attributed the slight loss in performance to the fact that ZHT must write to disk, while Memcached's data stayed completely in-memory.

3. THROUGHPUT:

We conducted several experiments to measure the throughput (see Figure 9). The throughputs of ZHT (TCP with connection caching) as well as that of Memcached increases near-linearly with scale, reaching nearly 7.4M ops/sec at 8Knode scale in both cases.

On the HEC-Cluster, as expected, ZHT has higher throughput than Cassandra. We expect the performance gap between Cassandra and ZHT to grow as system scales grows. Figure 10 shows the nearly 7x throughput difference between ZHT and Cassandra. Memcached performed as expected better as well, with a similar 27% higher overall throughput.

4. EFFICIENCY:

Efficiency is simply the measured throughput divided by the ideal throughput. In Figure 11, we show that ZHT and Memcached achieve different levels of efficiency (51%~100% for ZHT and 42%~53% for Memcached) up to 8K-node scales. Memcached's worse efficiency is attributed to having lower performance (higher latency) overall. Efficiency was computed by comparing ZHT and Memcached performance against the ideal latency/throughput (which was taken to be the better performer at 2-node scale – ZHT).

BUILDING BLOCK FOR DISTRIBUTED SYSTEMS:

1. FUSION FS: DISTRIBUTED DATAMANAGEMENT SYSTEM

FusionFS: Distributed Metadata Management We have an ongoing project to develop a new highly scalable distributed file system, called FusionFS [13]. FusionFS is optimized for a subset of HPC and many-task computing (MTC) [12, 59, 62, 63] workloads, and it is designed for extreme scales [61]. These workloads are often extremely data-intensive [56, 58, 60], and optimizing data locality [55] becomes critical to achieving good scalability and performance. In FusionFS, every compute node serves all three roles: client, metadata server, and storage server. The metadata servers use ZHT, which allows the metadata information to be dispersed throughout the system, and allows metadata lookups to occur in constant time at extremely high concurrency. Directories are considered as special files containing only metadata about the files in the directory. FusionFS leverages the FUSE kernel module to deliver a POSIX compatible interface as a user space filesystem. In order to measure the metadata performance of FusionFS (which in turn is based on ZHT), we built a benchmark that creates 10K files per node, across N directories, where N was equal to the number of nodes, ranging from 64 to 512. In the case of FusionFS, it could use the simple insert/lookup API of ZHT, as every node/client could modify metadata information of different directories. We compared the performance of metadata management of FusionFS with that of GPFS which is commonly deployed in production large-scale HEC systems.

1. ISTORE:

Large-scale storage systems require fault-tolerance mechanisms to handle failures, which are a norm rather than an exception. To deal with this, a new trend other than replication, includes the information dispersal algorithms [47, 48]. By implementing erasure coding, these algorithms encode the data into multiple blocks among which only a portion is necessary to recover the original data. IStore is a simple yet high-performance Information Dispersed Storage System that makes use of erasure coding and distributed metadata management with ZHT [50]. IStores' metadata performance throughput on 8 to 32 nodes in the HEC-Cluster. The workload consisted of 1024 files of different sizes ranging from 10KB to 1GB. The workload performed read and write operations on these files through the IStore. The IStore uses ZHT to manage metadata about file chunks. At each scale of N nodes, the IDA algorithm was configured

ISSN (Online) 2456 -1304

International Journal of Science, Engineering and Management (IJSEM)
Vol 2, Issue 12, December 2017

to chunk up files into N chunks, and storing this information in ZHT for later retrieval and the N chunks would be sent to or read from N different nodes. The smaller the files, the more metadata intensive IStore became, requiring as many as 500 metadata operations per second at 32 node scales.

2. MATRIX:

MATRIX is a distributed many-task computing [12] execution framework, which utilizes the adaptive work stealing algorithm to achieve distributed load balancing [51], and ZHT to submit tasks and monitor the task execution progress by the clients. We have a functioning prototype implemented in C, and have scaled this prototype on a BLUE GENE/P supercomputer up to 512 nodes (2K-cores) with good results. By using ZHT, the client could submit tasks to arbitrary node, or to all the nodes in a balanced distribution. The task status is distributed across all the compute nodes, and the client can look up the status information by relying on ZHT. We performed several synthetic benchmark experiments to evaluate the performance of MATRIX, and how it compares to the state-of-the-art Falkon [53] light-weight task execution framework (see Figure 18). The workload consisted of 100K tasks of various lengths, ranging from 0 seconds (NO-OP) to 1, 2, 4, and 8 seconds. It might be difficult to compare MATRIX with Falkon running on the SiCortex or the Linux Cluster, as MATRIX was run on the BLUE GENE/P. However, when comparing MATRIX with Falkon on the BLUE GENE/P for peak throughput, we see Falkon saturate at 1700 tasks/sec at 256-core scales (similarly as all the other test beds also saturate eventually). Falkon has a centralized architecture, and hence had limited scalability. MATRIX shows excellent growth in throughput, starting with 1100 tasks/sec at 256-core scales, up to almost 4900 tasks/sec at 2048-core scales. What is even more important is that there was no obvious sign of saturation, and the growth tracked well the increase in ZHT performance.

FUTURE WORK:

We have many ideas on how to improve ZHT. There are also many possible use cases where ZHT could make a significant contribution in performance or scalability. Network-aware topology: Given the popularity of multidimensional Torus networks on HEC systems, we believe that making ZHT network topology aware is critical to making ZHT scalable by ensuring that communication is kept localized when performing 1-to-1 communication. Broadcast primitive: We believe that a broadcast primitive (in addition to insert/lookup/remove/append) would be beneficial to transmit the key/value pairs efficiently to all nodes (potentially via a spanning tree). Data Indexing: We will explore the possibility of using ZHT to index data (not just metadata) based on its content. For this indexing to be successful, some domain specific knowledge regarding the data to be indexed will be necessary. MosaStore: MosaStore [30] is an experimental storage system under development at the University of British Columbia. MosaStore has a centralized manager to handle metadata, just like most other filesystems available. ZHT will be used to implement a distributed metadata manager for MosaStore. Swift: Swift [27, 57] is a system for the rapid and reliable specification, execution, and management of large-scale science and engineering workflows on clusters, grids, supercomputers, and clouds [64]. It supports applications that execute many tasks coupled by disk-resident datasets. We will work with the Swift team to integrate ZHT into Swift in order to achieve scalable data management.

CONCLUSION:

ZHT's performance and scalability are excellent up to 8K-node and 32K instances. On the 32K-core scale we achieved more than 18M operations/sec of throughput and 1.1ms of latency at 8K-node scale. The experiments were conducted on various machines, from a single node server, to a 64-node cluster, and an IBM Blue Gene/P supercomputer. On all these platforms ZHT exhibits great potential to be an excellent distributed key-value store, as well as a critical building block of large scale distributed systems, such as job schedulers and file systems. In future work, we expect to extend the performance evaluation to significantly larger scales, as well as involve more applications. 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 not normally 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 emerging paradIgms, such as Cloud Computing, Many-Task Computing, and High-Performance Computing.

REFERENCES:

1.H. Shen, C. Xu, and G. Chen. Cycloid: A Scalable Constant-Degree P2P Overlay Network. Performance Evaluation, 63(3):195-216, 2006

2. DeCandia,D.Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels. "Dynamo: Amazon's Highly Available Key-Value Store." SIGOPS Operating Systems Review, 2007

3. T. Li, R. Verma, X. Duan, H. Jin, I. Raicu, "ZHT: Zero-Hop Distributed Hash Tab for High-End Computing", ACM Performance Evaluation Review (PER), 2012