DISTRIBUTED NOSQL STORAGE FOR EXTREME-SCALE SYSTEM SERVICES IN SUPERCOMPUTERS AND CLOUDS

BY

TONGLIN LI

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate College of the
Illinois Institute of Technology

Approved _____
Advisor

Chicago, Illinois
December 2015

ACKNOWLEDGMENT

This dissertation could not have been completed without the consistent encouragement and support from my advisor Dr. Ioan Raicu and my family. I am grateful to Dr. Kate Keahey and Dr. Lavanya Ramakrishnan for advising my research work at Argonne National Laboratory and Lawrence Berkeley National Laboratory, respectively. There are many more people who help me succeed through my PhD study, particularly my dissertation committee: Dr. Zhiling Lan, Dr. Boris Glavic, and Dr. Ashfaq Khokhar.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

As supercomputers gain more parallelism at exponential rates, the storage infrastructure performance is increasing at a significantly lower rate due to relatively centralized management. This implies that the data management and data flow between the storage and compute resources is becoming the new bottleneck for large-scale applications. Similarly, cloud based distributed systems introduce other challenges stemming from the dynamic nature of cloud applications. This dissertation addresses several challenges on storage systems at extreme scales for supercomputers and clouds by designing and implementing a zero-hop distributed NoSQL storage system (ZHT), which has been tuned for the requirements of high-end computing systems. ZHT aims to be a building block for scalable distributed systems. The goals of ZHT are delivering high availability, good fault tolerance, light-weight design, persistence, dynamic joins and leaves, high throughput, and low latencies, at extreme scales (millions of nodes). We have evaluated ZHT's performance under a variety of systems, ranging from a Linux cluster with 64-nodes, an Amazon EC2 virtual cluster up to 96-nodes, to an IBM Blue Gene/P supercomputer with 8K-nodes. This work also presents several real systems that have adopted ZHT as well as other NoSQL systems, namely ZHT/Q, FusionFS, IStore, MATRIX, Slurm++, Fabriq, FREIDA-State, and WaggleDB, all of these real systems have been significantly simplified due to NoSQL storage systems, and have been shown to outperform other leading systems by orders of magnitude in some cases. Through our work, we have shown how NoSQL storage systems can help on both performance and scalability at large scales in such a variety of environments.

CHAPTER 1

INTRODUCTION

Today's science is generating datasets that are increasing exponentially in both complexity and volume, making their analysis, archival, and sharing one of the grand challenges of the 21st century. As supercomputers gain more parallelism at exponential rates, the storage infrastructure performance is increasing at a significantly lower rate. This implies that the data management and data flow between the storage and compute resources is becoming the new bottleneck for large-scale applications. The support for data intensive computing is critical to advancing modern science as storage systems have experienced a gap between capacity and bandwidth that increased more than 10-fold over the last decade. There is an emerging need for advanced techniques to manipulate, visualize and interpret large datasets. Many domains (e.g. astronomy, bioinformatics, and financial analysis) share these data management challenges, strengthening the potential impact from generic solutions.

"A supercomputer is a device for turning compute-bound problems into I/O bound problems" [1]. The quote from Ken Batcher reveals the essence of modern high performance computing and implies an ever-growing shift in bottlenecks from compute to I/O. For exascale computers, the challenges are even more radical [2], as the only viable approaches in next decade to achieve exascale computing all involve extremely high parallelism and concurrency. Up to 2015, some of the biggest systems already have more than 3 million general-purpose cores that are connected with high speed network such as torus [3] or fat tree [4]. Many experts predict that exascale computing will be a reality by the end of the decade; an exascale system is expected to have millions of nodes, billions of threads of execution, hundreds of petabytes of memory, and exabyte of persistent storage [5].

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 [6], PVFS [7] and Lustre [8]). 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.

One of the major bottlenecks in current state-of-the-art storage systems is metadata management. Metadata operations on parallel file systems can be inefficient at large scale. Experiments on the Blue Gene/P system at 16K-core scales show the various costs (wall-clock time measured at remote processor) for file/directory create on GPFS. Ideal performance would have been constant, but we see the cost of these basic metadata operations (e.g. create file) growing from tens of milliseconds on a single node (four-cores), to tens of seconds at 16K-core scales; at full machine scale of 160K-cores, we expect a file create to take over two minutes for the many directory case, and over 10 minutes for the single directory case. Previous work shows these times to be even worse, putting the full system scale metadata operations in the hour range, but the test bed as well as GPFS might have been improved over the last several years. Whether the time per metadata operation is minutes or hours on a large-scale HEC system, the conclusion is that 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, reaching saturation at only 4 to 32 core scales (on a 160K-core machine).

Other distributed file systems (e.g. Google's GFS and Yahoo's HDFS for

Hadoop) that have centralized metadata management make the problems observed with GPFS even worse from the scalability perspective. Future storage systems for high-end computing should support distributed metadata management, leveraging distributed data-structure tailored for this environment. The distributed data-structures share some characteristics with structured distributed hash tables, having resilience in face of failures with high availability; however, they should support close to constant time inserts/lookups/removes delivering the low latencies typically found in centralized metadata management (under light load). Metadata should be reliable and highly available, for which replication (a widely used mechanism) could be used.

HPC is not the only area that suffers the storage bottleneck. Similar with the HPC scenarios, cloud based distributed systems also have to face storage bottleneck. Furthermore, due to the dynamic nature of cloud applications, a suitable storage system needs to satisfy more requirements.

As an attempt to meet these needs, we propose and build ZHT (zero-hop distributed hash table [9, 10]), an instance of NoSQL database [11], 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 as well as clouds. 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 [12] and Memcached [13] and found it to offer superior performance for the features and portability it supports, at large scales up to 16K-nodes. We also compared it to DynamoDB[14] in the Amazon AWS Cloud, and found that ZHT offers significantly better performance and economic cost than DynamoDB.

This work also presents several real systems that have adopted ZHT as well as other NoSQL systems, namely FusionFS [15] (distributed metadata management and data provenance capture/query), ZHT/Q (a flexible QoS fortified distributed key-value storage system for the cloud), IStore [16] (data chunk metadata management), MATRIX (distributed scheduling), Slurm++ (distributed HPC job launch), Fqbriq (distributed message queue management), FREIDA-State [17] (state management for scientific applications on cloud), and WaggleDB [18] (a Cloud-based interactive data infrastructure for sensor network applications); all of these real systems have been simplified due to NoSQL storage systems, and have been shown to outperform other leading systems by orders of magnitude in some cases. It's important to highlight that some of these systems are rooted in HPC systems from supercomputers, while others are rooted in clouds and ad-hoc distributed systems; through our work, we have shown how versatile NoSQL storage systems can be in such a variety of environments.

The contributions of this work are as follows:

- Design and implementation of a NoSQL storage system named ZHT and optimized for high-end computing; Verified ZHT's scalability on 32K-cores scale;
- Design and implementation of ZHT/Q, a flexible QoS fortified distributed key-value storage system based on ZHT. The new system is optimized to satisfy QoS on latency while achieving high throughput;
- Application of ZHT in real systems, namely FusionFS, IStore, MATRIX, Slurm++, Fabriq, Graph/Z;
- Design and implementation of FRIEDA-State, a NoSQL-based state management system for scientific applications running in cloud environments, with lightweight capturing, efficient storage and vector clock-based event ordering;
- Design and implementation of WaggleDB, a NoSQL-based dynamically scalable cloud data infrastructure for sensor networks;
- Both real systems and simulations were used to evaluate NoSQL at extreme scales, up to thousands of real nodes, and millions of simulated nodes;
- Prove that Distributed NoSQL storage systems that are light-weight, dynamic, resilient, portable, supporting both low latency and high throughput, are a excellent and fundamental building block for more complex distributed systems;

These contributions have led to 15 peer reviewed publications, and one publications that is under review.

- **Tonglin Li**, Ke Wang, Shiva Srivastava, Dongfang Zhao, Kan Qiao, Iman Sadooghi, Xiaobing Zhou, Ioan Raicu, *A Flexible QoS Fortified Distributed Key-Value Storage System for the Cloud*, IEEE International Conference on Big Data (IEEE BigData 2015)

- **Tonglin Li**, Xiaobing Zhou, Ke Wang, Dongfang Zhao, Iman Sadooghi, Zhao Zhang, Ioan Raicu, *A Convergence of Distributed Key-Value Storage in Cloud Computing and Supercomputing*, Journal of Concurrency and Computation Practice and Experience (CCPE) 2015.

- **Tonglin Li**, Kate Keahey, Ke Wang, Dongfang Zhao, Ioan Raicu, *A Dynamically Scalable Cloud Data Infrastructure for Sensor Networks*, ScienceCloud 2015

- **Tonglin Li**, Chaoqi Ma, Jiabao Li, Xiaobing Zhou, Ke Wang, Dongfang Zhao and Ioan Raicu, *GRAPH/Z: A Key-Value Store Based Scalable Graph Processing System*, IEEE Cluster 2015

- **Tonglin Li**, Ioan Raicu, Lavanya Ramakrishnan, *Scalable State Management for Scientific Applications in the Cloud*, BigData 2014

- **Tonglin Li**, Kate Keahey, Rajesh Sankaran, Pete Beckman, Ioan Raicu, *A Cloud-based Interactive Data Infrastructure for Sensor Networks*, ACM/IEEE Supercomputing Conference Regular Research Poster, SC2014

- **Tonglin Li**, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Adithya Rajendran, Zhao Zhang, Ioan Raicu, *ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table*, 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2013.

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

- Iman Sadooghi, Jess Hernndez Martin, **Tonglin Li**, Kevin Brandstatter, Ketan Maheshwari, Tiago Pais Pitta de Lacerda Ruivo, Gabriele Garzoglio, Steven

Timm, Yong Zhao, Ioan Raicu, *Understanding the Performance and Potential of Cloud Computing for Scientific Applications*, IEEE Transactions on Cloud Computing (TCC), 2015

- Ke Wang, Kan Qiao, Iman Sadooghi, Xiaobing Zhou, **Tonglin Li**, Michael Lang, Ioan Raicu, *Load-balanced and locality-aware scheduling for data-intensive workloads at extreme scales*, Journal of Concurrency and Computation Practice and Experience (CCPE) 2015.

- Ke Wang, Ning Liu, Iman Sadooghi, Xi Yang, Xiaobing Zhou, **Tonglin Li**, M Lang, Xian-He Sun, I Raicu, *Overcoming Hadoop scaling limitations through distributed task execution*, IEEE Cluster 2015

- Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, **Tonglin Li**, Ke Wang, Dries Kimpe, Philip Carns, Robert Ross, and Ioan Raicu. *FusionFS: Towards Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems*, IEEE International Conference on Big Data 2014

- Ke Wang, Xiaobing Zhou, **Tonglin Li**, Dongfang Zhao, Michael Lang, Ioan Raicu, *Optimizing Load Balancing and Data-Locality with Data-aware Scheduling*, IEEE International Conference on Big Data 2014

- Ke Wang, Kan Qiao, Iman Sadooghi, Xiaobing Zhou, **Tonglin Li**, Michael Lang, Ioan Raicu, *Loadbalanced and localityaware scheduling for dataintensive workloads at extreme scales*, Journal of Concurrency and Computation Practice and Experience (CCPE) 2015.

- Iman Sadooghi, Ke Wang, Shiva Srivastava, Dharmit Patel, Dongfang Zhao, **Tonglin Li**, Ioan Raicu, *FaBRiQ: Leveraging Distributed Hash Tables towards Distributed Publish-Subscribe Message Queues*, IEEE/ACM International Symposium on Big Data Computing (BDC) 2015

These contributions have also lead to 13 additional [19–32] peer-reviewed publications which have used the work from this dissertation as a building block towards more complex distributed systems.

The rest of this dissertation is organized as follows: chapter 2 describes ZHT, a Light-weight reliable persistent dynamic Scalable zero-hop distributed key-value store for supercomputers and clouds, and proved that distributed key-value storage systems can be light-weight, dynamic, resilient, portable, supporting both low latency and high throughput. Chapter 3 describes a series of real distributed systems that have been built based on ZHT. Chapter 4 describes ZHT/Q, a flexible QoS (Quality of Service) fortified distributed key-value storage system for clouds and data centers. Chapter 5 describes two real cloud-based scientific application systems that used NoSQL databases to boost the performance, scalability and to simplify the design. Chapter 6 summaries the related work. Chapter 7 describes the conclustion and future work.

CHAPTER 2

KEY-VALUE STORAGE SYSTEM FOR SUPERCOMPUTERS AND CLOUDS

This chapter presents a convergence of distributed Key-Value storage systems in supercomputers and clouds. It specifically presents ZHT, a zero-hop distributed key-value store system, which has been tuned for the requirements of high-end computing systems. ZHT aims to be a building block for future distributed systems, such as parallel and distributed file systems, distributed job management systems, and parallel programming systems. ZHT has some important properties, such as being light-weight, dynamically allowing nodes join and leave, fault tolerant through replication, persistent, scalable, and supporting unconventional operations such as append, compare and swap, callback in addition to the traditional insert/lookup/remove. We have evaluated ZHT's performance under a variety of systems, ranging from a Linux cluster with 64-nodes, an Amazon EC2 virtual cluster up to 96-nodes, to an IBM Blue Gene/P supercomputer with 8K-nodes. We compared ZHT against other key/value stores and found it offers superior performance for the features and portability it supports. This chapter also presents several real systems that have adopted ZHT, namely FusionFS (a distributed file system), IStore (a storage system with erasure coding), MATRIX (distributed scheduling), Slurm++ (distributed HPC job launch), Fabriq (distributed message queue management); all of these real systems have been simplified due to Key-Value storage systems, and have been shown to outperform other leading systems by orders of magnitude in some cases. It's important to highlight that some of these systems are rooted in HPC systems from supercomputers, while others are rooted in clouds and ad-hoc distributed systems; through our work, we have shown how versatile Key-Value storage systems can be in such a variety of environments.

2.1 Introduction

Today's science is generating datasets that are increasing exponentially in both complexity and volume, making their analysis, archival, and sharing one of the grand challenges of the 21st century [33]. As supercomputers gain more parallelism at exponential rates, the storage infrastructure performance is increasing at a significantly lower rate [34]. This implies that the data management and data flow between the storage and compute resources is becoming the new bottleneck for large-scale applications. The support for data intensive computing is critical to advancing modern science as storage systems have experienced a gap between capacity and bandwidth that increased more than 10-fold over the last decade. There is an emerging need for advanced techniques to manipulate, visualize and interpret large datasets. Many domains (e.g. astronomy, bioinformatics [35], security [36–46], micro electro mechanical systems [47–50], GIS(Geographic Information System) [51–55] and financial analysis) share these data management challenges, strengthening the potential impact from generic solutions.

"A supercomputer is a device for turning compute-bound problems into I/O bound problems" [1]. The quote from Ken Batcher reveals the essence of modern high performance computing and implies an ever-growing shift in bottlenecks from compute to I/O. For exascale computers, the challenges are even more radical, as the only viable approaches in the next decade to achieve exascale computing all involve extremely high parallelism and concurrency [25]. Up to 2015, some of the biggest systems already have more than 3 million general-purpose cores. Many experts predict that exascale computing will be a reality by the end of the decade; an exascale system is expected to have millions of nodes, billions of threads of execution, hundreds of petabytes of memory, and exabyte of persistent storage.

In the current decades-old architecture of HPC systems, storage(e.g. parallel file systems, such as GPFS [6], PVFS [7] and Lustre [8]) is completely separated from

compute resources. The connection between them is a high speed network. 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. The unscalable storage architecture could be a "show-stopper" in building exascale systems [25]. Although there are works such as burst-buffer [56, 57] to alleviate the parallel file system bottleneck, in the long run the need for building efficient and scalable distributed storage for high performance computing (HPC) systems that will scale three to four orders of magnitude is on the horizon.

One of the major bottlenecks in current state-of-the-art storage systems is metadata management. Metadata operations on most of parallel and distributed file systems can be inefficient at large scales. Our previous work(Fig.2.1) on a Blue Gene/P supercomputer with 16K-cores shows the various costs for file/directory creating(metadata operation of file systems) on GPFS. GPFS's metadata performance degrades rapidly under concurrent operations, reaching saturation at only 4 to 32 core scales (on a 160K-core machine). Ideal performance would have been constant at different scales, but we see the cost of these basic metadata operations (e.g. create file) growing exponentially, from tens of milliseconds on a single node (four-cores), to tens of seconds at 16K-core scales; at full machine scale of 160K-cores, we expect one file creation to take over two minutes for the many directory case, and over 10 minutes for the single directory case. Previous work shows these times to be even worse, putting the full system scale metadata operations in hours range, although GPFS might have been improved over the last several years. On a large scale HPC system, whether the time per metadata operation is minutes or hours, the conclusion is that the metadata management in GPFS does not have enough degree of distribution, and not enough emphasis was placed on avoiding lock contention.

Other parallel or distributed file systems (e.g. Google's GFS and Yahoo's

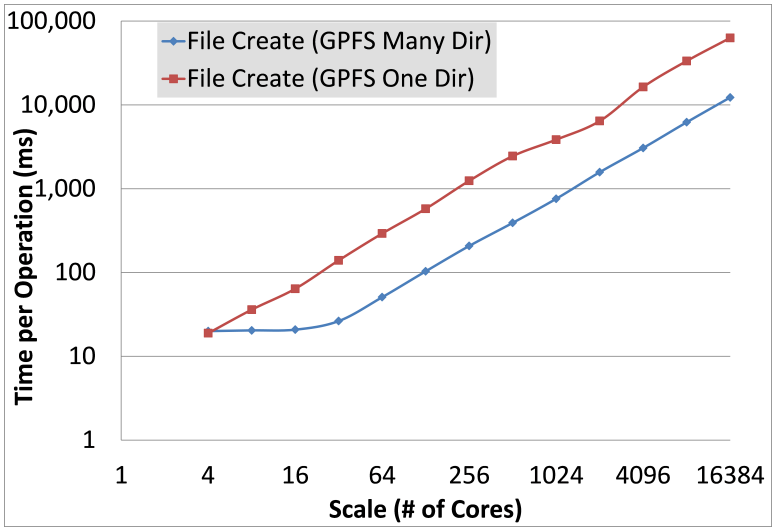


Figure 2.1. Time per operation (touch) on GPFS on various numbers of processors on a IBM Blue Gene/P

HDFS) that have centralized metadata management make the problems observed with GPFS even worse from the scalability perspective. Future storage systems for high-end computing should support distributed metadata management, leveraging distributed data-structure tailored for this environment. The distributed data-structures share some characteristics with structured distributed hash tables, having resilience in face of failures with high availability; however, they should support close to constant time operations and deliver the low latencies typically found in centralized metadata management (under light load).

HPC storage is not the only area that suffers the storage bottleneck. Similar with the HPC scenarios, cloud based distributed systems also have to face storage bottleneck. Furthermore, due to the dynamic nature of cloud applications, a suitable storage system needs to satisfy more requirements, such as being able to handle dynamic nodes join and leave on the fly and the flexibility to run on different cloud instance types simultaneously.

As an initial attempt to meet these needs, we propose and build ZHT(zero-

hop distributed hash table [9, 10, 58, 59]), an instance of NoSQL database [11]. ZHT has been tuned for the specific requirements of high-end computing (e.g. trustworthy/reliable hardware, fast networks, non-existent "churn", low latencies). 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. Highlighted features include being lightweight, dynamically allowing nodes join and leave, fault tolerant through replication, handling failures gracefully, efficiently propagating events throughout the system, a customized consistent hashing mechanism. Unlike conventional key-value store, ZHT implemented new operations such as append,compare_swap/ and state_change_ callback in addition to insert/lookup/remove. To provide ZHT a persistent back end, we also created a fast persistent single node data store that could be easily integrated and operated in lightweight Linux OS typically found on today's supercomputers as well as clouds. 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 [12] and Memcached [13] and found it to offer superior performance for the features and portability it supports, at large scales up to 16K-nodes. We also compared it to DynamoDB [14] in the Amazon AWS Cloud, and found that ZHT offers significantly better performance and economic cost than DynamoDB.

It also covers five real systems (FusionFS, IStore, MATRIX, Slurm++, and Fabriq) at a high-level. They have been integrated with ZHT, and evaluated at modest scales. 1) ZHT is used in the FusionFS distributed file system to deliver distributed meta-data management and data provenance capture/query. On a 512-nodes deployment at Los Alamos National Lab, FusionFS reached 509kops/s metadata per-

formance. 2) ZHT is used in the IStore, an erasure coding enabled distributed object storage system, to manage chunk locations delivering more than 500 chunks/sec at 32-nodes scales. 3) ZHT is also used as a building block to MATRIX, a distributed task scheduling system, delivering 13k jobs/sec throughput at 4K-core scales. 4) Slurm++, a distributed job launch system that avoids the centralized gateway nodes in Slurm. 5) ZHT is used in Fabriq, a distributed message queue, to store messages reliably, and load balance the resource requirements. All of these real systems have been simplified due to NoSQL storage systems, and have been shown to outperform other leading systems by orders of magnitude in some cases. It's important to highlight that some of these systems are rooted in HPC systems from supercomputers, while others are rooted in clouds and ad-hoc distributed systems; through our work, we have shown how versatile NoSQL storage systems can be in such a variety of environments.

The contributions of the original conference paper [9] that chapter has extended are:

- Design and implementation of ZHT, a light-weight, high performance, fault tolerant, persistent, dynamic, and highly scalable distributed hash table, optimized for supercomputers and clusters.
- Support for unconventional operations, such as append, in order to reduce lock contention.
- Extensive system micro benchmarks conducted on up to 8K real nodes. Simulations used to evaluate ZHT and some competitors at up to millions of simulated nodes.
- Integration and evaluation with three real systems (FusionFS, IStore, and MATRIX), managing distributed storage metadata and distributed job scheduling

information.

This chapter has extended the original conference paper [9] through the following significant contributions:

- Prove that Distributed NoSQL key/value storage systems that are light-weight, dynamic, resilient, portable, supporting both low latency and high throughput, are a reality.
- Extended primitive operations with new operations(cswap and callback), which can significantly simplify the design of upper level applications.
- Extended evaluations of ZHT to the Amazon AWS cloud for both performance and economics.
- Showcased two new real system that have been built with ZHT, namely Slurm++ (HPC job launch system) and Fabriq(distributed message queue system)

2.2 ZHT Design and Implementation

Most high performance computing environments are batch oriented, in which an allocation is configured in the beginning at run time. Such an allocation generally has information about the available hardware and software resources, and the amount of resources (e.g. number of nodes) generally would not change until the allocation is terminated. The only possible reason to decrease the allocation is hardware(nodes, hard drive or network) or low level software system (such as monitoring [60, 61] and scheduling [**?**, 62, 63] systems)failure. Because nodes in HPC systems are generally reliable and have predicable uptime (from the start of an allocation, to shut down on de-allocation), it implies that node "churn" in HPC occurs much less frequently than in traditional DHTs. In ZHT's static membership (for HPC), every node at bootstrap time knows how to contact each other.

However in some dynamic environments, the system properties are different. In dynamic environments such as clouds or data centers, nodes may join (for system performance boosting) and leave (node failure or scheduled maintenance) at any time. We believe that dynamic membership would be important for such environments, especially for cloud computing systems, and hence have made efforts to support it without affecting primitive operations' time complexity. This principle guided our design of the proposed dynamic membership management in ZHT.

The node ID space and membership table are treated as a ring-shaped name space. The node IDs in ZHT can be randomly distributed throughout the network. The random distribution of the ID space has worked well up to 32K-cores. A hash function maps arbitrarily long strings to index values, which can then be used to efficiently retrieve the communication address (e.g. host name, IP address, port, MPI-rank) from a local in-memory membership table . Depending on the volume of information that is stored, storing the entire membership table consumes only a small (less than 1%) portion of available memory on each node.

With a 1K-nodes scale allocation on Intrepid, an IBM BlueGene/P supercomputer, one ZHT instance's memory footprint is less than 8MB. The memory footprint consists of ZHT server binary, 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 periodically. The membership table is very small, each entry(presenting a node) only takes 32 bytes, 1 million nodes only need 32 MB space. By tuning the number of key/value pairs that are cached in memory, users can reach the balance between performance and memory consumption.

2.2.1 Primitive operations. Similar to other key-value stores, ZHT offers conventional operations, namely `insert`, `lookup` and `remove`. These three operations

are implemented efficiently to achieve low latency. Based on the requirements that we have faced during developing large-scale distributed systems, we abstract three extra operations. These operations can significantly simplify the system design for many scenarios.

insert operation. Insert a string type key-value pair into data store; return execution status.

lookup operation. Lookup a given key string and return a value string, if the key is ever modified by `append` operation, it will return a list of value.

remove operation. Remove a key-value pair and return execution status.

append operation. This allows user assigning multiple value to a same key (algorithm 2.1). This is not a feature that many hash maps do, and is especially rare in persistent ones as well. We found the append operation critical in supporting lock-free concurrent modification in ZHT (eliminating the need for a distributed system lock); using append, we were able to implement a highly efficient metadata service for a distributed file system, where certain metadata (e.g. directory lists) could be concurrently modified across many clients. Consider a typical use case in distributed and parallel file systems, creating 10K files from 10K processes in one directory; the concurrent metadata modification occurs usually via distributed locks, which is known to be inefficient. Append primitive looks like multi-version concurrency control (MVCC) appeared in some data stores like Voldemort, Riak and HBase, but it's implemented differently for different purpose. In MVCC, only one version of data is marked as current and active, while in append primitives all data fields are current and are treated equally.

cswap (compare and swap) operation. In some applications when a client reads a value and sets it to another value that depends on the read value, it will need

to lock the keyvalue record. If another client wants to access the same key, there comes the lock contention problem. A naive way to implement is to add a global lock for each queried key in the key-value stores, which is apparently not scalable. A better approach is used in ZHT to implement an atomic operation that is executed on the key-value store server side, which finish the value update before return to the client (algorithm 2.2). This primitive is similar with "Check-and-Set" in Memcached and Couchbase Server [64].

callback operation. Callback operation is used to notify a client upon a specified value change (algorithm 2.3). Sometimes an application (such as a state machine) needs to wait on specific state change in the key-value store before moving on. A simple way to do this is letting the client to pull the data server periodically (e.g. every 1 sec) until the state changes, which brings too much unnecessary communication. To solve this problem, we move the value checking from the client side to the server side and introduce a new operation called **state_change_callback** (abbreviated to callback). The data server creates a dedicated thread for all state change callback requests, and the main thread keeps processing other requests. Within a given period of time (time_out), if the server finds the value being changed to expected value, it returns success signal to the client; otherwise returns failure signal to the client.

2.2.2 Terminologies. In this section, we briefly introduce the terms used in the this work.

Physical node. A physical node is an independent physical machine. Each physical node may run several ZHT instances that are differentiated with a combination of IP address and port.

Instance. A ZHT instance is a ZHT server process that handles the requests

Algorithm 2.1 Append

```
1: procedure APPEND_CLIENT(key, appended_value)
2:     pack ← pack(key, appended_value)
3:     send(serialize(pack))
4: end procedure
5: procedure APPEND_SERVER_RESPONSE(key, appended_value)
6:     record_p ← lookup(key)                          ▷ return an entry pointer
7:     while cursor_p.value.next! = Null do
8:         cursor_p ← record_p.value.next
9:     end while
10:    cursor_p.next.value ← appended_value
11: end procedure
```

Algorithm 2.2 Compare_and_swap

```
1: procedure COMPARE_SWAP_SERVER_RESPONSE(key, update())
2:     record_p ← lookup(key)
3:     record_p.lock()
4:     seenValue ← record_p.value
5:     newValue ← update(seenValue)
6:     record_p.value ← newValue
7:     record_p.unlock()
8: end procedure
```

Algorithm 2.3 Callback

```
1: procedure CALLBACK_SERVER_RESPONSE(key, expectedValue)
2:     while record_p.value! = expectedValue do
3:         record_p ← lookup(key)
4:         record_p ← record_p.value.next
5:     end while
6:     notify_client()
7: end procedure
```

from clients. Each instance takes care of some partitions. By adjusting the number of instance per physical node, ZHT can fit in heterogeneous computing systems with various storage capacities and performance. A ZHT instance can be identified by a combination of IP address and port. Therefore the partitions can be many more than the instances and physical nodes.

Partition. A partition is a contiguous range of the key address space; a file on disk is associated with each partition for persistence. We developed a single-node persistent key-value store (NoVoHT) as ZHT's back-end, which also takes care of each partition.

Manager. A Manager is a service process running on each physical node and takes charge of starting and shutting down ZHT instances, managing membership table and partition migration. As traditional consistent hashing does, initially we assign each of the k physical nodes a manager and one or more ZHT instances, each with a universal unique id (UUID) in the ring-shaped space. The entire name space N (a 64-bit integer) is evenly distributed into n partitions where n is a fixed big number indicating the maximal number of nodes that can be used in the system. It is worth noting that while n (the number of partitions, also the maximal number of physical nodes) cannot be changed without potentially rehashing all the key/value pairs stored

in ZHT, i (the number of ZHT instances) as well as k (the number of physical nodes) is changeable with changes only to the membership table. Each physical node has one manager, holds n/k partitions, with each partition storing N/n key-value pairs and i/k ZHT instances serving requests. Each partition (which can be persisted to disk) can be moved across different physical nodes when nodes join, leave, or fail.

For example, in an initial system of 1000 ZHT instances (typically running on 1000 nodes), where each instance contains 1000 partitions, the overall system could scale up to 1 million instances with 1 million physical nodes. Experiments validate this approach by showing that there is little impact (0.73ms vs. 0.77ms per request when scaling from 1 partition to 1000 partitions respectively) on the performance as we increase the number of partitions per instance. This design allows us to avoid a potentially expensive rehash of many key-value pairs when the need arises to migrate partitions.

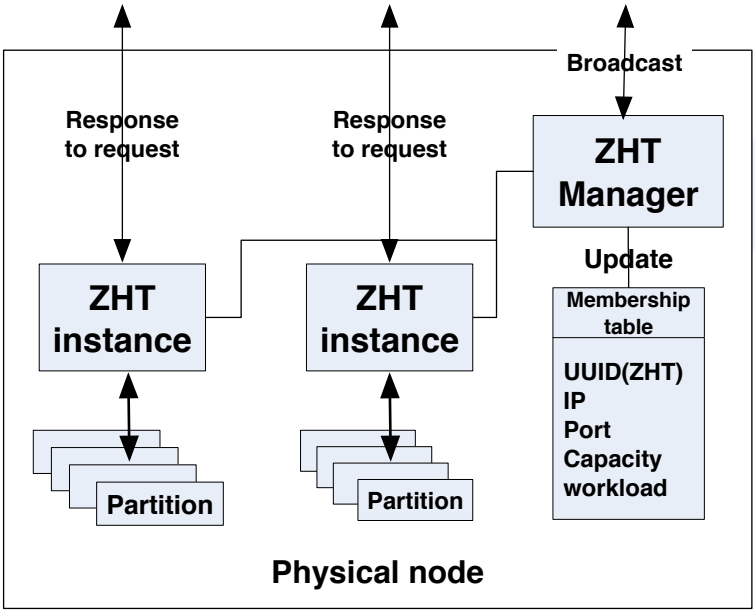


Figure 2.2. ZHT server single node architecture. Each physical can run multiple ZHT instances, which further manages multiple partitions. Each partition stores a contiguous key space.

2.2.3 Membership management. ZHT supports both static and dynamic node membership. In static membership, clients and servers fetch neighbor list from the batch job scheduler or a given file during the bootstrapping phase. Once the membership is established, no new nodes is allowed to join. Nodes could leave the system due to failures; we assume failed nodes do not recover. For the dynamic membership, nodes are allowed to join and leave the system dynamically. Many DHTs and key-value stores support dynamic membership, but typically deliver this through logarithmic routing. They use consistent hashing which sacrifices performance for scalability under dynamic environment. We address this issue with an improved consistent hashing mechanism that requires constant-number-hop (typically 1, at most 2) routing. With this novel design, we offer the desired flexibility of dynamic membership while maintaining very low latency through constant time routing.

Data migration and membership update. The design goal is to ensure that only minimal impact on performance and scalability would be posed by adopting dynamic membership. With dynamic membership, there comes the need to potentially migrate data from one physical node to another. In order to achieve this, ZHT organizes its data in partitions, and migrates a partition as a whole instead of many individual key-value pairs. This avoids rehashing all effected key-value pairs, as most DHTs that adopt consistent hashing. Moving an partition altogether is remarkably more efficient than rehashing individual key/value pairs. When migration is in progress, the partition state is locked. All incoming requests during this time are queued, until the migration is completed. In the meanwhile because the partition state won't change, corresponding replicas also won't change. This keeps the entire system state consistent. If failure occurs during migration, simply don't apply the changes to corresponding partitions and replicas (discard the queued requests and report error to clients and administrators), this will eventually bring the system to roll back to a consistent earlier state.

Node Joins. Upon a node joins the network, it firstly checks out a copy of membership table from a ZHT Manager on a random physical node. Based on this table, the new node can find the physical nodes with the most partitions, then join the ring as one of the heavily loaded node's neighbor and migrates some of the partitions from the "busiest" node to itself. Migrating a partition is as easy as moving a file, without having to rehash the key/value pairs in the partition.

Node departures. On planned node departures (e.g. during system maintenance), the administrator fetches current membership table from a random physical node, modify it accordingly, then broadcast the incremental table to other ZHT managers to update their local tables. The departing managers firstly migrate their data partitions to neighboring nodes, and then proceed to depart. For an unplanned departure (e.g. due to a node failure), it will be firstly detected by the client that sends a request and doesn't get response within a given timeout, or due to another ZHT instance initiates a server-to-server operation and fails (e.g. migration, replication, etc.). Upon a certain number of try-and-fails on a certain server, the client marks the corresponding physical node down on its local membership table and informs a random ZHT manager about this failure. The client then sends the request to the first available replica of the failed node. At the same time, the manager updates its local membership table and broadcasts the change to the whole network, and initiates a rebuilding of the replicas, specifically increasing replicas for all partitions that are stored on the failed physical node in order to maintain the specified replication level.

Client Side State. In case that clients and servers are not on the same physical nodes, it's necessary to keep the client-side membership table updated. Since the node joining and departure changes the number of partitions covered by a ZHT server, clients might send requests to wrong nodes if it's local membership table is not updated. To address this issue, we lazily update clients' membership table. Only

when the requests are sent mistakenly, the ZHT server will send back a copy of latest membership table to the clients.

2.2.4 Server architecture. We have explored various architectures for ZHT server. Since typical Key-Value store operations are very small but frequent, we optimize ZHT more for small requests. In early prototypes, we adopted a multi-threading design, in which a server throw a thread for each request, but the overheads of starting, switching, and tearing down threads was too high compare to the work to be done for a request. We eventually converged on a notably more streamlined architecture, an event-driven server architecture based on *epoll*[65]. The current epoll-based ZHT outperforms the multithread version by at least 3X. We'll discuss the performance difference in detail in the evaluation section.

2.2.5 Fault tolerance. ZHT handles failures gracefully by lazily tagging non-responding nodes failed. ZHT uses replication to ensure data persistence in face of failures. Newly inserted or modified key/value data will be replicated asynchronously to secondary replicas that have closer hashed location. By communicating only with near neighbors, this approach ensures that replicas only consume less network resources when we succeed in implementing the topology-aware and locality-aware protocols (similar approach can be found in [66,67]). Despite the lack of topology-aware in the current ZHT, the asynchronous replication only adds relatively small overhead when adding more replicas at modest scales (up to 4K-cores).

ZHT is fully distributed, and single node failures do not affect the functionality. The key/value pairs that were stored on the failed node can be found on replica nodes. Upon failures, the replicas answers the requests for data that were originally stored on the failed node.

When ZHT is shut down due to hardware maintenance or system reboot pur-

poses, the entire state of ZHT could be dumped to local persistent storage; note that every change to the in-memory data structure is dumped to disk periodically, ensuring the entire state of the data store can be recovered if needed. Considering the increasing size of memory and SSDs, as well as I/O performance improvements in the future, it is expected that a multi-gigabyte of state could be retrieved in just seconds.

Once ZHT is bootstrapped, the system verification time should not be related to the size of the system. In the event that a fresh new ZHT instance is to be bootstrapped, the process is very efficient with its current static membership table, as there is no global communication required between nodes (see Fig.2.3). Nevertheless, we expect the time to bootstrap ZHT to be insignificant compare to the batch scheduler's overheads on a high-end computing system, which could potentially include node provisioning, OS booting, starting of network services, and perhaps the mounting of some parallel file systems. At 1K-node scale on IBM Blue/Gene machine, the time to start the batch scheduled job is about 150 seconds, after which the ZHT bootstrap takes another 8 seconds at 1K-node scale and 10 seconds to bootstrap it at 8K-node scale. Fig.2.3 shows the bootstrap time increase with the scale.

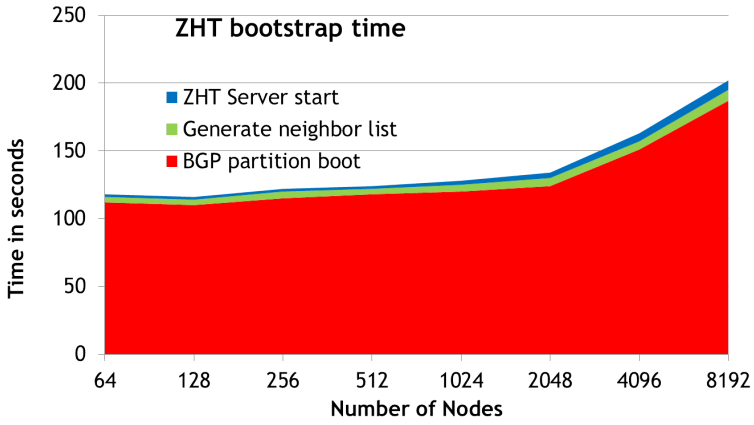


Figure 2.3. ZHT Bootstrap time on Blue Gene/P from 64 to 8K nodes

2.2.6 Socket level thread safety and connection caching. In a dynamic network environment, many multi-threaded problems are related to socket. In other words, if the sockets are thread safe, many message transferring out-of-order related issues are smoothed away. The previous version of ZHT client is thread-safe in operation level, such as insert, delete, etc. It relies on a shared mutex to avoid any contention problem. This was not as efficient as it could be. We have made ZHT client as thread safe not only in operation level but also in socket level.

In ZHT client, sockets are stored in a LRU cache. The key is combination of IP and port of ZHT server that client talks to. When the client needs to communicate with a server it will first try to find the key as mentioned. If not found, the socket is initialized and bound to IP and port of the server, and then put into the cache for reuse. In the same time, a mutex is initialized for that socket for protection. We removed the single mutex shared by ZHT API(s) and only rely on socket level thread-safe to realize overall ZHT client thread-safe.

2.2.7 Consistency. ZHT uses replication to handle server failures. Current version only allows clients interact with a single primary replica for write operations (insert/remove/append). This decision is based on the fact that if we allowed multiple replicas to be concurrently modified, a more complicated consistency mechanism such as Paxos protocol has to be maintained, which may cause serious performance loss. If enabling all replicas to accept read request, the read performance in terms of throughput will gain some boost, however, to ensure consistent results for all read requests to different replicas will need version checking and updating, which may increase the request latency significantly. Upon a primary replica failure, a corresponding secondary replica will take place of primary and directly talk to clients. Strong consistency is maintained between ZHT primary and secondary replica, operation completion thus will be acknowledged to clients right after the second replica is

updated. Other replicas are asynchronously updated after the secondary replica updating is complete, bringing ZHT to eventual consistency. By this hybrid consistency approach, ZHT attains high throughput, good availability and reasonable consistency level at the same time.

2.2.8 Persistence. ZHT is a distributed in-memory data-structure. In order to withstand failures and restarts, it also supports persistence. We designed and implemented a Non-Volatile Hash Table (NoVoHT [68]) for ZHT, which uses a log-based persistence mechanism with periodic checkpointing. We evaluated several existing systems, such as KyotoCabinet HashDB and BerkeleyDB, but performance and missing features prompted us to implement our own solution.

NoVoHT is a custom-built lightweight hash table at the core, with added features built on top. The design of the map structure is an array of linked lists. This structure makes collision handling efficient. Also, it helps lookup time, by eliminating the worst case of iterating the entire array in the case of it being full. Finally, it allows the application to overfill the map, with more keys than buckets. While this would impact time of insert and remove, it keeps the space used for the array lower. It also allows the key/value store to allow lock-free read operations. When a key/value pair is inserted, it writes the key-value pair to the file specified, and records where it was written with the key-value pair in the map. By recording the location in the file, removal is efficient. When an element is removed it removes the pair from the map, and marks the spot in the file. By marking the file, if the application crashes, that pair will not be inserted into the map when the file state is recovered. NoVoHT allows a customized threshold, which determines how many removes to do before the file is rewritten with the pairs in the map (effectively eliminating the pairs that were marked for removal from the file). NoVoHT also supports periodic garbage collection to reclaim free space at timed intervals.

2.2.9 Implementation. ZHT is implemented in C/C++, and has very few dependencies. It consists of 14900 lines of code, and is an open source project accessible at GitHub. The dependencies of ZHT are NoVoHT (discussed in the next section) and Google Protocol Buffers [69].

2.3 Performance Evaluation via Synthetic benchmarks

In this section, we evaluate ZHT's performance, in terms of including request latencies, system throughput, performance of hashing functions, persistence overhead, and replication cost. Firstly we describe the configuration of test beds and benchmark setup. Secondly we presented a comprehensive performance evaluation. Two popular NoSQL systems (Memcached and Cassandra) that offer similar functionality or features are compared against ZHT, along with a cloud database service, DynamoDB on EC2 [70] cloud.

2.3.1 Testbeds, Metrics, and Workloads. We used several platforms to evaluate ZHT's performance.

- Kodiak, a Parallel Reconfigurable Observational Environment (PROBE) [71] at Los Alamos National Laboratory, it has 1024 nodes, and each node has two 64-bit AMD Opteron processors at 2.6GHz and 8GB memory.
- Intrepid, an IBM Blue Gene/P [72] supercomputer at Argonne Leadership Computing Facility [73]. 8K physical nodes (32K cores) are used, each of which has a 4-core PowerPC 450 processor and 2GB of RAM. Intrepid was used to compare ZHT to Memcached. Note that this system does not have persistent local node disks so the RAM-based disks were used as persistence option.
- HEC-Cluster, a 64-node (512-core) cluster at IIT, each node has two quad-core processors, 8GB RAM, 200GB HDD, it's used to compare ZHT to Cassandra.

- DataSys, an 8-core x64 server at IIT. Two Intel Xeon quad-core processors with HT, 48 GB RAM, used to compare KyotoCabinet, BerkeleyDB and NoVoHT.
- Fusion, a 48-core x64 server at IIT. Four AMD Opteron 12-core processors, 256GB RAM, used to compare KyotoCabinet, BerkeleyDB and NoVoHT.
- Amazon EC2 Cloud, up to 96 cc.8xlarge VMs.

On each node, one or more ZHT client-server pairs are deployed, namely ZHT instances. Each instance is configured with one partition known as NoVoHT. Test workload is a set of key-value pairs where the key is 15 bytes and value is 132 bytes. Clients sequentially send all of the key-value pairs through a ZHT Client API for insert, then lookup, and then remove. The additional operations such as append are evaluated separately due to their different nature of the operation. Since the keys are randomly generated, the communication pattern is All-to-All, with same number of servers and clients.

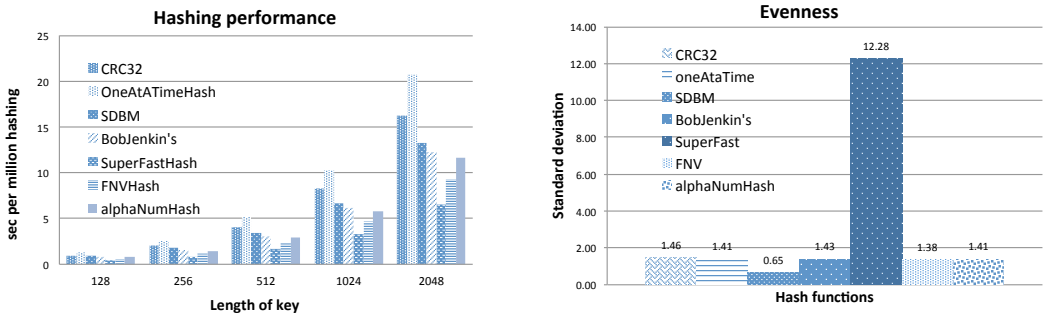
The metrics measured and reported are:

- Latency. Latency presents the time taken from a request to be submitted from a client to a response to be received by the client, measured in milliseconds (ms). Since various operations (insert/lookup/remove) latencies are quite close, we use average of the three operations to simplify the results presentation. Note that the latency consists of round trip network communication, system processing, and storage access time. Since Blue Gene/P doesn't have persistent storage for each work node, ram-disks are used for the experiment, while regular spinning drives are used in experiments on cluster.
- Throughput. The number of operations (insert/lookup/remove) the system can handle over some period of time, measured in Kilo Ops/s.

- Ideal throughput. Measured throughput between two nodes times the number of nodes.
- Efficiency. Ratio between measured throughput and ideal throughput.

2.3.2 Hash Functions. Similar with what we observed, the time spent on hashing keys to nodes is not major of the total cost, but with the time passed, the accumulation could be observed. We investigate some of the usual hash functions for figuring out the trade-off between performance and evenness.

As shown in Fig.2.4(a) that some hash functions are faster than others, but a more important concern rather than performance is the evenness (Fig.2.4(b)). Since the worst hash function we investigated has performance of 0.02ms/hash, and thus negligible compared to other overhead. Meanwhile the evenness is essential to the entire performance. An ideal hash function should be able to spread keys evenly do as to provide a natural load balancing mechanism.



(a) Hash functions performance

(b) Key distribution evenness

Figure 2.4. Hash function comparison

2.3.3 Individual data store synthetic benchmark. We compared KyotoCabinet [74] to NoVoHT with persistence. We used identical workloads of 1M, 10M, and 100M operations (insert/get/remove), operating on fixed length key value pairs (see Fig.2.5). When comparing NoVoHT with KyotoCabinet or BerkeleyDB, we observed

significantly better capacity scalability on NoVoHT. Although BerkeleyDB has some advantages such as memory usage, it does this at the cost of higher latency. When comparing NoVoHT persistence to non-persistence, we noticed that most of the overhead of the operations is on the hard disk I/O.

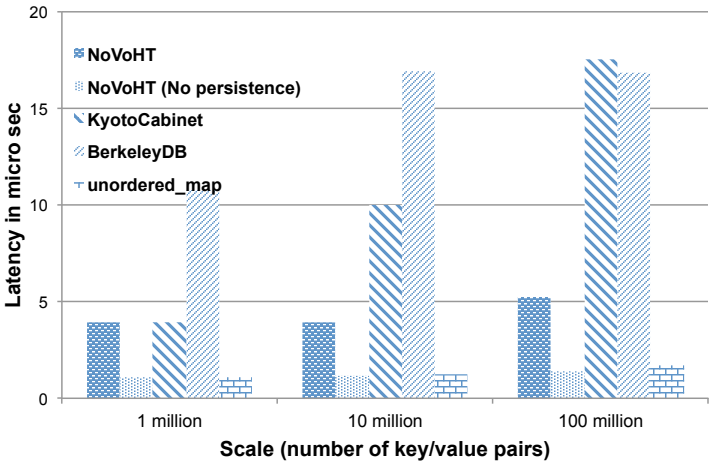


Figure 2.5. Average latency of NoVoHT, KyotoCabinet and BerkeleyDB

2.3.4 System Latencies.

2.3.4.1 Synthetic benchmarks On Supercomputers and clusters. We evaluated the latency on both the Blue Gene/P supercomputer and HEC-Cluster. We evaluated different implementations of ZHT with various communication protocols, such as UDP, TCP with connection caching, and compare them to Cassandra and Memcached.

Since ZHT latency is mostly dominated by cross node communication overhead, in-node operations mostly happened in memory, the latency difference between insert and lookup is minimal (Fig.2.6). Because of this performance characteristic of ZHT, we use average latency of insert, lookup and remove operations to simplify presentation of the figures in rest of the chapter.

ZHT shows great scalability at up to 8K-node scale. As shown in Fig.2.7, on

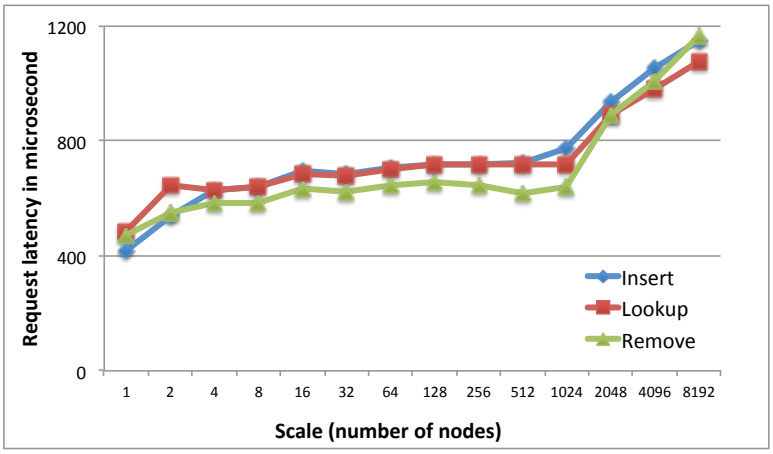


Figure 2.6. Basic operation latency comparison on Blue Gene/P. Note that insert/lookup operation latencies are extremely close, because majority of the latency are communication overhead, which are same for insert and lookup.

single node, the latency of both TCP with connection caching and UDP are extremely low (0.5ms). When scaling up, ZHT shows slowly increased latency, up to 1.1ms at 8K-node scales. We see that TCP with connection caching delivers practically same performance of that of UDP, at all the scales we measured. Memcached scaled well too, the latencies ranging from 1.1 ms to 1.4 ms from single node to 8K nodes (note that this represents a 25% to 139% increased latency, depending on the scale, implying some scalability issue). Note that IBM Blue Gene/P uses a 3D Torus network [75] for communication, which means the routing needs increasing number of hops at larger scales to send messages cross compute nodes. This also explains why the latency start to increase on large scale – one rack of Blue Gene/P has 1024 nodes, any scale larger than 1024 involves 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 racks. If running on a Fat-tree network [76], we expect more constant latency (before the network is saturated) due to the constant routing hops.

The CDF plot (Fig.2.8) shows very similar trends for different scales that imply excellent scalability. On 64 node-scale 90% requests finish in 853us and 99% requests

finish in 1259us. When scaling up to 1024 node-scale, the latencies are only slightly increased, 90% finishes in 1053us and 99% finishes in 3105us (table 2.1).

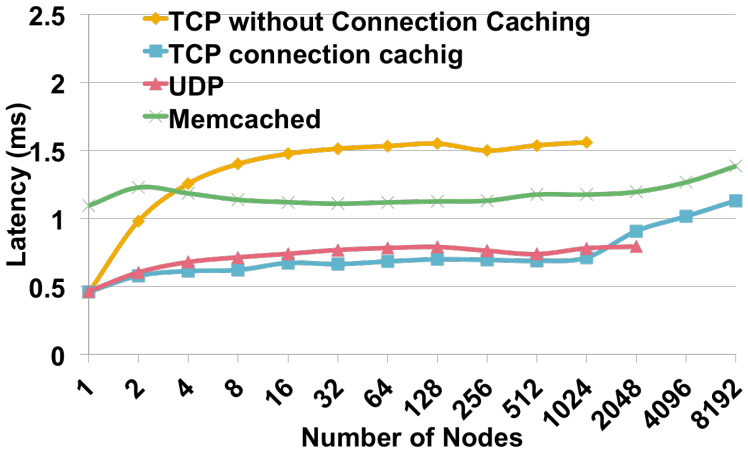


Figure 2.7. Performance evaluation of ZHT and Memcached plotting latency vs. scale (1 to 8K nodes on the Blue Gene/P)

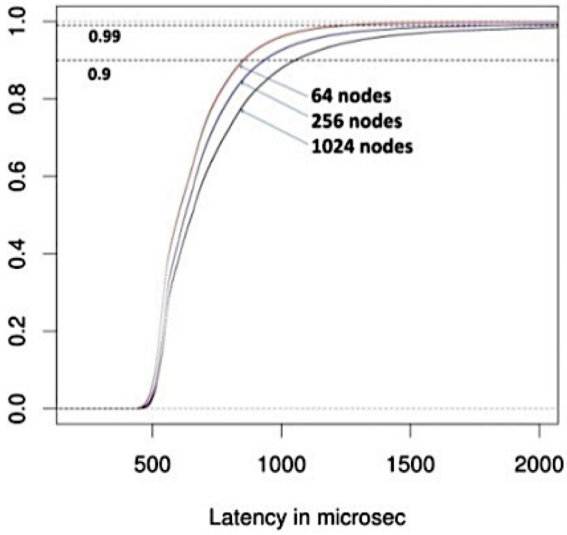


Figure 2.8. CDF of Benchmark on Blue Gene/P CDF

Because of Cassandra's implementation in Java, and the the Blue Gene/P machine lacks of support for Java , we evaluated Cassandra, Memcached, and ZHT on a conventional Linux cluster, the HEC-Cluster. Not surprisingly, as shown in 2.9, ZHT's latency is notably lower than that of Cassandra. ZHT also shows superior scalability over Cassandra. This is mainly because Cassandra adopts a logarithmic

Table 2.1. ZHT Latencies ON Blue Gene/P in Microsecond

Scales	75%	90%	95%	99%	Average	Throughput
64	713	853	961	1259	676	90632
256	755	933	1097	1848	748	356137
1024	820	1053	1289	3105	1007	1316942

routing algorithm and ZHT uses constant routing. Interestingly, Memcached only shows slightly better performance than ZHT at up to 64-node scales. We attributed ZHT's slight increment in latency to the fact that ZHT must write to disk, while Memcached's data stayed in memory completely.

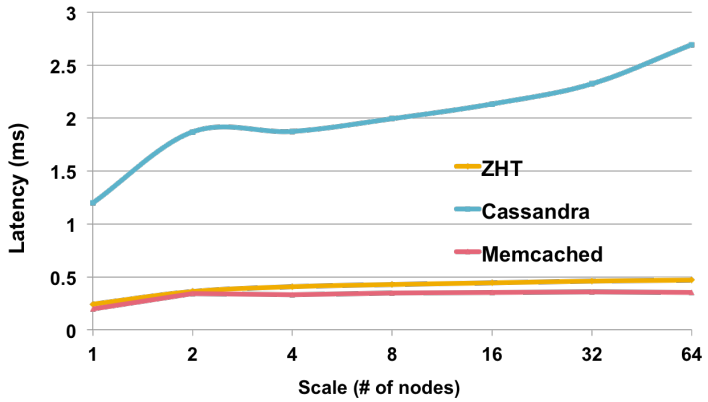


Figure 2.9. Performance evaluation of ZHT, Memcached and Cassandra plotting latency vs. scale (1 to 64 nodes on an AMD Cluster)

2.3.4.2 Synthetic benchmarks On Cloud. 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 shown in table 2.2.

Different from the result that we got on supercomputers, the results on EC2 cloud reveal interesting inconsistency on various scales. Ideally the request latencies on large scale should fall into a narrow window like they do on smaller scale. On smaller instances such as m1.medium, ZHT latency CDF plots are quite different

(Fig.2.10). Although 90% requests are still finished within similar time (600 to 800us), 99% requests latency doubled when scales increase by 4 times. In other words, on EC2, latencies have much longer tails in larger scales than in smaller scales.

If the experiments are conducted on smaller instances, the long tail will be even longer (see Fig.2.10(b)). On larger instances such as cc2.8xlarge, ZHT latency CDF plots are closer than they are on smaller instances. Fig.2.10 shows that the latency trends of 4, 16 and 64 nodes scales are quite similar. This difference confirms a fact that smaller EC2 instances (such as m1.medium) share more hardware resources (include CPU and network bandwidth) than larger instances (like cc2.8xlarge). Therefore small instances have more interference than large ones, so the application performance will also be influenced. Because of the less shared resource, big EC2 instance types may act more like regular cluster or supercomputer, since little interference exists. Note that on each cc2.8xlarge instance we start 8 ZHT servers and clients to better utilize the resource.

We also conducted micro benchmarks for Amazon DynamoDB as a comparison. Since DynamoDB default maximum throughput is 10K/s, all benchmarks are under that provision. There is no information released about how many nodes are used to offer a specific throughput. Since we have observe that the latency of DynamoDB doesn't change much with scales, and the value is around 10ms, we have to use many clients to saturate the capacity. We deployed clients for DynamoDB micro benchmarks on cluster computing instance, namely cc2.8xlarge. 8 clients were started on each instance.

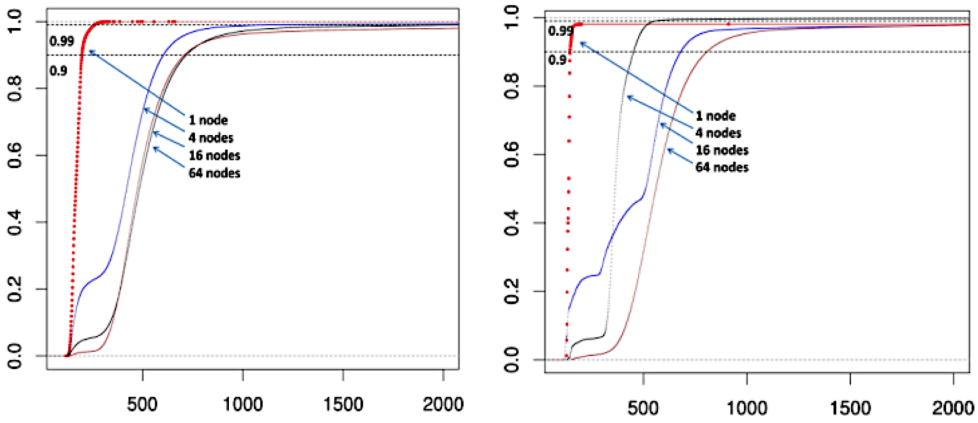
As expected, DynamoDB has much longer latency on all scales. On 4-node (32 clients) scale it is 22 times slower than ZHT. In the CDF comparison DynamoDB shows that its 90% latencies fall into a 20x wider time window than ZHT. When we ran 8 clients on 64 nodes, DynamoDB started to give errors that complain about

Table 2.2. Profile of EC2 Instances Used in Experiments

Instance type	m1.medium	cc2.8xlarge
CPU	2 EC2 Compute Unit	88 EC2 Compute Units
Memory	3.75GB	60.5
Storage	160GB	3370GB
I/O Performance	Moderate	High (10 Gb/s Ethernet)
Cost	$0.112/hour	$2.4/hour

Table 2.3. ZHT Latencies on cc2.8xlarge EC2 Instance in Microsecond

Scales	75%	90%	95%	99%	Average	Throughput
8	186	199	214	260	172	46421
32	509	603	681	1114	426	75080
128	588	717	844	2071	542	236065
512	574	708	865	3568	608	841040



(a) cc2.8xlarge instance (b) m1.medium instance

Figure 2.10. Latency distribution comparisons

over used throughput so we can't continue to push experiments 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 limited maximum throughput,

Table 2.4. ZHT Latencies on m1.medium EC2 Instance in Microsecond

Scales	75%	90%	95%	99%	Average	Throughput
1	142	146	154	4887	229	4892.4
4	591	680	767	12500	760	4978.5
16	369	452	482	556	388.3	11351
64	665	807	970	3880	711.5	91201

Table 2.5. DynamoDB Latencies With Clients ON EC2 cc2.8xlarge Instance, 8 Clients/Instance

Scales	75%	90%	95%	99%	Average	Throughput
8	11942	13794	20491	35358	12169	83.39
32	10081	11324	12448	34173	9515	3363.11
128	10735	12128	16091	37009	11104	11527
512	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED

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. When we tried with scales larger than 128 clients for DynamoDB, more than half request failed, because the throughput was beyond the provisioned one.

2.3.5 System Throughput.

2.3.5.1 Synthetic benchmarks On Supercomputers and clusters. We conducted several experiments to measure the throughput. The throughput of ZHT (TCP with connection caching) as well as that of Memcached increases near-linearly with scaling, reaching nearly 7.4M ops/sec at 8K-node scale in both cases.

On the HEC-Cluster, ZHT has higher throughput than Cassandra as expected. We expect the performance gap between Cassandra and ZHT to grow as system scales

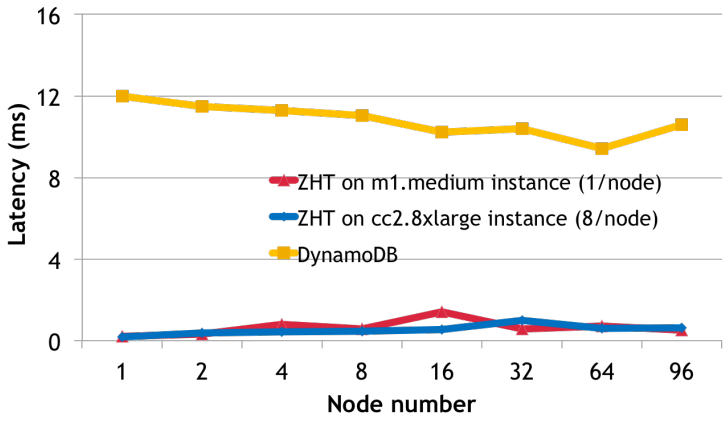


Figure 2.11. Latency comparison, ZHT v.s DynamoDB on EC2

grows due to ZHT's faster routing algorithm. Fig.2.13 shows the nearly 7x throughput difference between Cassandra and ZHT. As expected, Memcached performed better as well, with 27% higher total throughput.

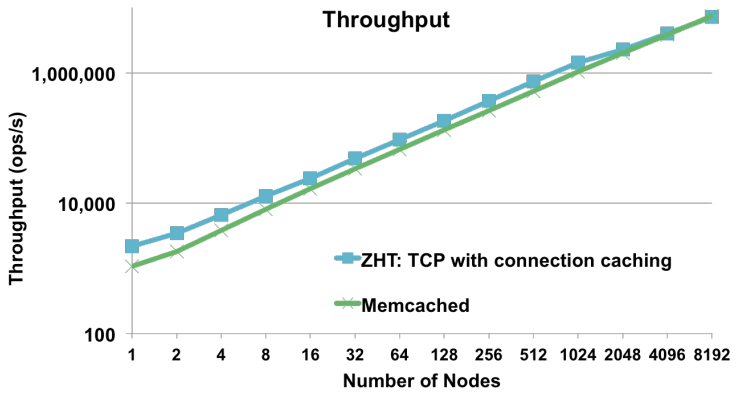


Figure 2.12. Performance evaluation of ZHT and Memcached plotting throughput vs. scale (1 to 8K nodes on the BLUE GENE/P)

2.3.5.2 Synthetic benchmarks On Cloud.

Throughput. In Fig.2.14, due to the interference between m1.medium instances, ZHT shows mild fluctuation in throughput. On 2cc.8xlarge instances, the fluctuation almost disappears and the throughput is close to linear. Although DynamoDB seems to stay with a linear growth, the absolute throughput is quite low.

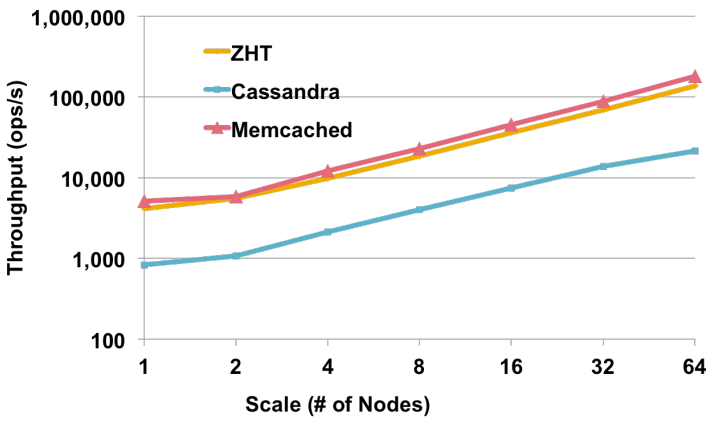


Figure 2.13. Performance evaluation of ZHT, Memcached and Casandra plotting throughput vs. scale (1 to 64 nodes on the HEC-Cluster)

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 that 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 that is provisioned (namely capacity) by the users. When the throughput is beyond provisioned capacity, DyndmoDB will saturate and give errors, requests start to fail.

Running Cost. When discussing cloud, the cost is always a big concern[77, 78]. We calculated hourly cost for both ZHT and DynamoDB on different scales. We calculate the ideal cost for DynamoDB, assuming the user always provisions the same throughput to fit their need, then according to Amazon's pricing policy, for 1k

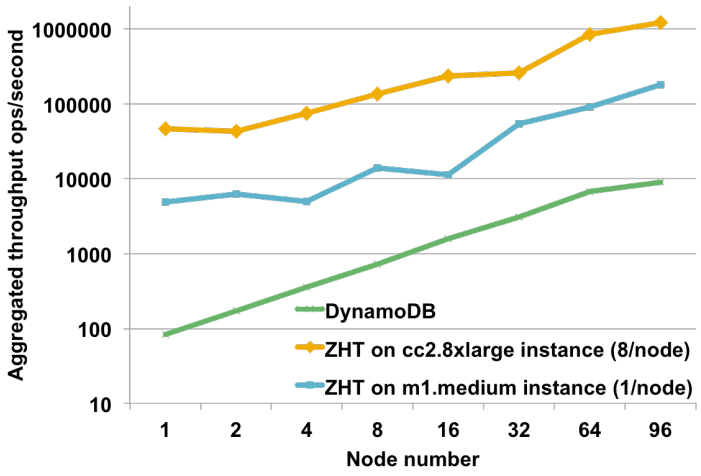


Figure 2.14. Aggregated throughput of ZHT and DynamoDB on EC2

ops/sec throughput, the cost is 0.65 cents per hour.

On 2-node scale DynamoDB cost 65 times more than ZHT; on largest scale that DynamoDB can support, it still cost 32 times more than ZHT for a same throughput (Fig.2.15). Note the cost for DynamoDB doesn't include the EC2 instances for running clients, it will cost even more if include the client cost. These are huge cost savings applications could have by running their NoSQL distributed key/value stores on their own, at the expense of managing their own NoSQL setup.

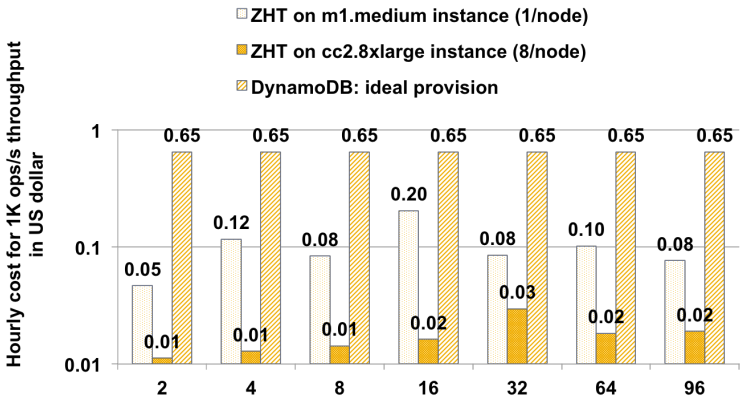


Figure 2.15. Running cost comparison

2.3.6 Scalability and efficiency. Although the throughput achieved by ZHT is impressive at many millions of ops/sec, it is important to investigate the efficiency of the system. 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, the smallest test that involving the network communication). In Fig.2.16, we show that Memcached and ZHT achieve different levels of efficiency at up to 8K-node scales. The reason why the performance over 1K-nodes degrades more sharply is because on Blue Gene/P system, 1K-nodes form a rack, and communication across the rack is more expensive (at least this is the case for TCP/UDP.

Although we were not able to run experiments at more than 8K-node scales due to time allocation on Intrepid, we have simulated ZHT on a PeerSim-based simulator. It was interesting that the simulator results were able to closely match the results up to 8K-node scales (where we achieved 8M ops/sec), giving on average only 3% of difference. The simulation showed efficiency drop to 8% at exascale levels (1M nodes). This sounds like ZHT would not scale to an exascale system, but a closer look at what 8% really means is worthy. 100% efficiency implies a latency of about 0.6ms per operation (ZHT latency at 2 node scales). 51% efficiency means 1.1ms latency (this is the performance of ZHT at 8K-node scales). 8% efficiency means the latency is as low as 7ms, at 1M node scales which is still extremely low. At 1M node scales and with latencies of 7ms, ZHT would achieve 150M ops/sec throughput.

2.3.7 Aggregated performance. Each Blue Gene/P compute node has 4 cores, to fully utilize the compute resource, we conduct experiments with various numbers of ZHT instances on each node and measure the request throughput and latency. We expect to achieve higher aggregate throughput by running multiple ZHT instances per node. The experiment results implies that the best resource utilization and efficiency

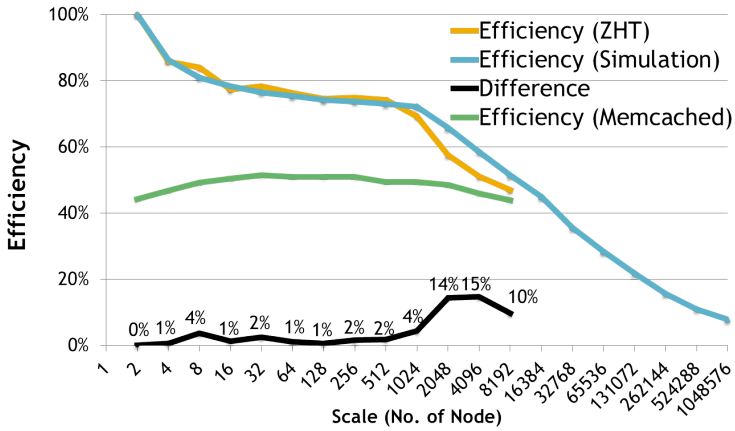


Figure 2.16. Performance evaluation of ZHT plotting measured efficiency and simulated efficiency vs. scale (1 to 8K nodes on the Blue Gene/P and 1 to 1M nodes on PeerSim)

can be achieved by assigning one instance to each core. In a setting with up to 4 instances per node, the aggregated throughput is the compelling (16.1M ops/sec as opposed to 7.3M ops/sec for 1 instance per node, a 2.2X increase), and the latency is still extremely low (2.08ms on 8K-nodes scale with 32K-instances).

2.4 Summary

ZHT is optimized for high-end computing systems and is designed and implemented to serve as a foundation to the development of fault-tolerant, high-performance, and scalable storage systems. We have used mature technologies such as TCP, UDP, and an epoll-based event-driven model, which makes it easier to deploy. It offers persistency with NoVoHT, a persistent high performance hash table. ZHT can survive various failures while keeping overheads minimal. It's also flexible, supporting dynamic nodes join and departure. We have shown 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 AMD Opteron cluster, an IBM BlueGene/P supercomputer, to the Amazon cloud.

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 HPC, and open the door to a 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.

CHAPTER 3

USING ZHT AS BUILDING BLOCKS FOR LARGE SCALE DISTRIBUTED SYSTEMS

In the era of Big Data and Cloud, distributed key-value stores are increasingly used as building blocks of large scale applications. Comparing to traditional relational databases, key-value stores are particularly compelling due to their low latency and excellent scalability. This section presents some real systems that have adopted ZHT as a building block. It also lead to additional publications [17, 18, 31, 79–84].

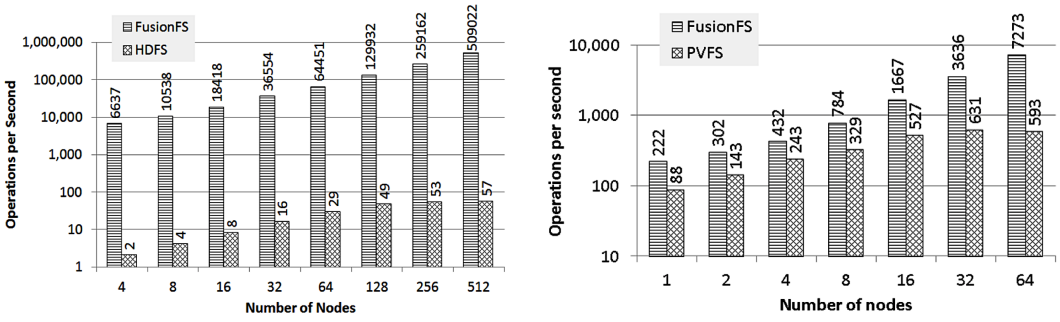
3.1 FusionFS: a Distributed File System with Distributed Metadata Management

We have an ongoing project to develop a new highly scalable distributed file system, called FusionFS [15, 85, 86]. FusionFS is optimized for a subset of HPC and many-task computing (MTC) workloads. 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 [87] kernel module to deliver a POSIX compatible interface as a user space filesystem.

We compare the metadata performance between FusionFS and HDFS on Kodiak. Both storage systems have FUSE/POSIX disabled. We have each node create (i.e. "touch") a large number of empty files (with unique names), and we measure the number of files created per second. In essence, each touched file indicates a metadata operation. The aggregate metadata throughput of different scales is reported in Fig.3.1(a). The gap between FusionFS and HDFS is about more than 3 orders of magnitude. Note that, HDFS starts to flatten out from 128 nodes, while FusionFS

keeps doubling the throughput all the way to 512 nodes, ending up with almost 4 orders of magnitude speedup (509022 vs. 57).

We then compare the metadata performance between FusionFS and PVFS on Intrepid. The result is reported in Fig.3.1(b). FusionFS outperforms PVFS on a single node, which justifies that our metadata optimization for big directory (i.e. append vs. update) is quite efficient. FusionFS shows a linear scalability, where PVFS is saturated at 32 nodes.



(a) Throughput on Kodiak

(b) Throughput on Intrepid

Figure 3.1. FusionFS: metadata performance comparison

3.2 IStore: an Erasure Coding Distributed Storage System

IStore is a simple yet high-performance Information Dispersed Storage System that makes use of erasure coding [88–90], and distributed metadata management with ZHT. Fig.3.2 shows 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. At each scale of N nodes, the IDA algorithm was configured 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.

3.3 MATRIX: a Distributed Many-Task Computing Scheduling Frame-

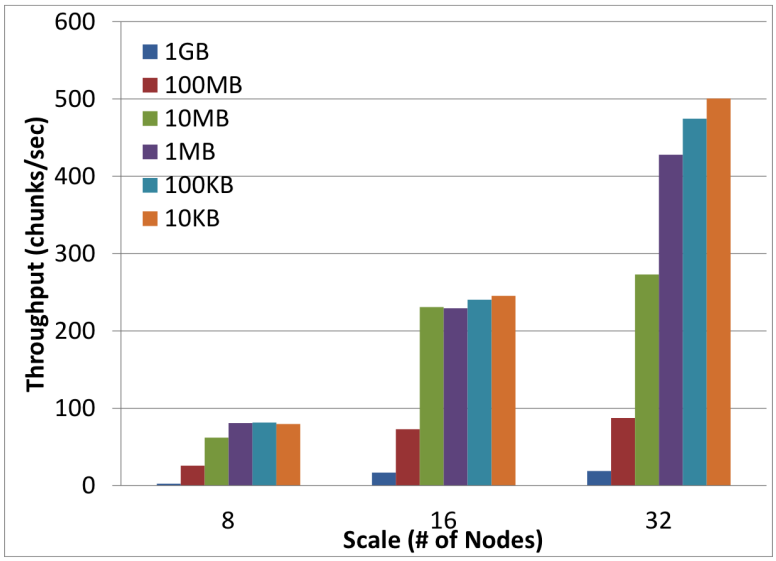


Figure 3.2. IStore metadata performance on HEC-Cluster

work

MATRIX [91,92] is a distributed many-task computing execution framework, which utilizes the adaptive work stealing algorithm to achieve distributed load balancing, and uses ZHT to submit tasks and monitor the task execution progress by the clients. Similar scheduling methods can be found in [93,94]. 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 [95] lightweight task execution framework. Fig.3.3 shows the results from a study of how efficient we can utilize up to 2K-cores with varying size tasks using both MATRIX and the distributed version of Falkon (which used a nave hierarchical distribution of tasks). We see MATRIX outperform Falkon across the board with across all size tasks, achieving efficiencies starting at 92% up to 97%, while Falkon only achieved 18% to 82%.

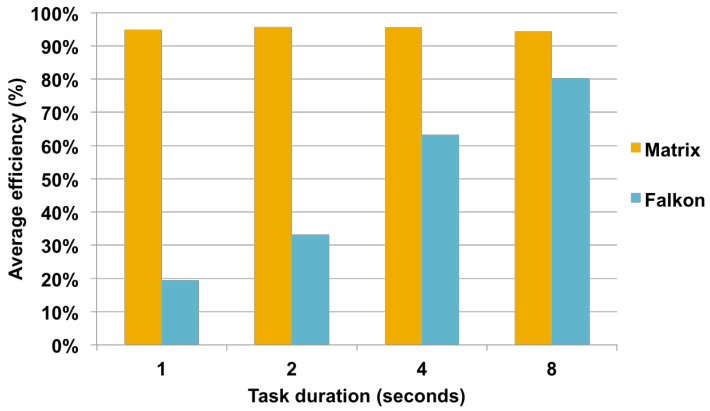


Figure 3.3. Comparison of MATRIX and Falkon average efficiency (between 256 and 2048 cores) of 100K sleep tasks of different granularity (1 to 8 seconds)

3.4 Slurm++: a Distributed HPC Job Launch

We have developed a distributed job launch prototype, SLURM++ based on SLURM [96], which serves as a core part for distributed job management system to do resource allocation and job launching. We used the ZHT as the data storage system to keep the job and resource metadata information in a globally accessible system.

We see that the average per-job ZHT message count shows decreasing trend (from 30.1 messages / job at 50 nodes to 24.7 messages at 500 nodes) with respect to the scale. This is likely because when adding more partitions, each job that needs to steal resource would have higher chance to get resource, as there are more options. This gives us intuition about how promising the resource stealing and compare and swap algorithms would solve the resource allocation and contention problems of distributed job management system towards exascale ensemble computing.

3.5 Fabriq: a Distributed Message Queue

We propose Fast, Balanced and Reliable Distributed Message Queue (FaBRiQ [97]), a persistent reliable message queue that aims to achieve high throughput and low latency while keeping the near perfect load balance even on large scales. Fabriq

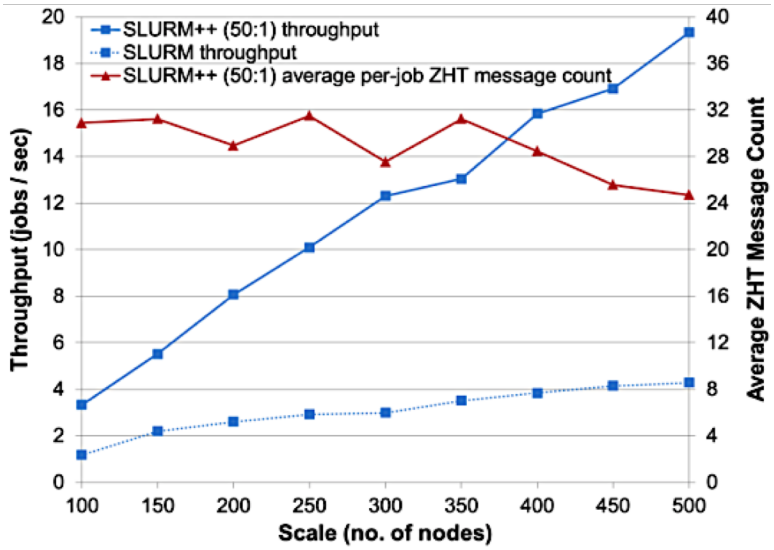
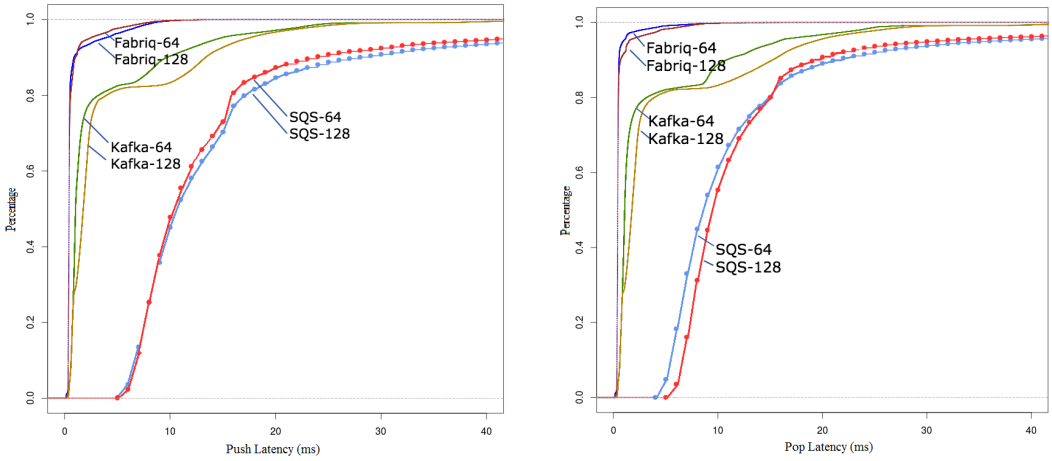


Figure 3.4. Throughput comparison for Slurm and Slurm++

uses ZHT as its building block. Fabriq leverages ZHT components to support persistence, consistency and reliable messaging. Another unique feature of Fabriq is the guarantee of exactly once delivery. To our best knowledge, no other DMQ provides such guarantee. Most of the DMQs either provide no delivery guarantee or at least once delivery guarantee. The fact that Fabriq provides low latency makes it a good fit for HPC and MTC workloads that are sensitive to latency and require high performance. Furthermore, providing high throughput on larger scales and persistence makes Fabriq a good option for HTC applications.

At 128 nodes scale, Fabriqs throughput was as high as 1.8 Gigabytes/sec for 1 Megabytes messages, and more than 90,000 messages/sec for 50 bytes messages. At the same scale, Fabriqs latency was less than 1 millisecond. Our framework outperforms other state of the art systems including Kafka and SQS in throughput and latency. According to Fig.3.5(a), at the 50 percentile, the push latency of Fabriq, Kafka, and SQS are respectively 0.42ms, 1.03ms, and 11ms. However, the problem with the Kafka is having a long tail on latency. At the 90 percentile, the push latency of Fabriq, Kafka, and SQS are respectively 0.89ms, 10.4ms, and 10.8ms. We can no-



(a) CDF: push latency comparison

(b) CDF: pop latency comparison

Figure 3.5. Comparison for Fabriq vs. SQS, IronMQ, HDMQ, and Windows Azure Service Bus

tice that the range of latency on Fabriq significantly shorter than the Kafka. At the 99.9 percentile, the push latency of Fabriq, Kafka, and SQS are respectively 11.98ms, 543ms, and 202ms. Similarly, Fig.3.5(b) shows a long range on the pop operations for Kafka and SQS. The maximum pop operation time on the on Fabriq, Kafka, and SQS were respectively 25.5ms, 3221ms, and 512ms.

CHAPTER 4

VERSATILITY & AND QOS IN KEY-VALUE STORAGE SYSTEM

4.1 Introduction

Distributed key-value stores are known for their ease of use and attractive performance. Major technology companies such as Facebook, Amazon and Google have built their data infrastructure with key-value stores to accommodate their fast growing businesses. Due to the complexity of system design, deployment and maintenance, along with the running cost, more and more companies choose to share a single key-value storage system between different applications and services. Take Facebook as an example, workloads from user accounts information, web-app object metadata and system data on service location, etc., run on one Memcached deployment [98]. Most of applications have their unique performance requirements. Some applications may prefer lowest latency, some prefer high total throughput, while others may like to have a well-balanced performance profile. These potentially conflicting requirements can be very different from the design goals of conventional key-value stores, which mostly focus on low-latency.

How to choose a good solution that meets many applications' needs is still an open question. The choice is even not obvious for latency – one of the most commonly used metrics. Different applications can tolerate very different latency ranges. For example, a shopping cart application can satisfy customers with 50 ms latency; instant messaging users are fine with 500 ms while a metadata service for databases or file systems requires as low as possible latency, ideally no longer than 5 ms [99]. Giving all applications same efforts and optimizing on the same aspect (single request latency) is not necessarily appropriate, as sometimes it might harm the total throughput provision of the system, and lower the resource utilization. This is especially true when key-value stores are delivered as cloud services that need to

serve many different applications and users [100].

In this chapter we present ZHT/Q, a flexible QoS Fortified distributed key-value storage system for Clouds. It enhances a high performance key/value store with flexible QoS (Quality of Service) properties such that both configurable latency and high aggregated throughput can be achieved. It satisfies different applications' latency requirements with QoS while improves the overall performance through dynamic and adaptive request batching mechanisms. The system QoS provides guaranteed and best-effort service on latency for different scenarios. It also watches the performance change and dynamically adjusts the batching strategy to alleviate performance degradation upon traffic.

The contributions of this work include:

- We design and implement a flexible QoS fortified distributed key-value storage system on top of our previous plain key-value store [9]. The new system is optimized to satisfy QoS on latency while achieving high throughput;
- Our system supports different QoS latency on a single deployment for multiple concurrent applications, both guaranteed and best-effort services are provided;
- Extensive performance evaluation is conducted through both real system micro benchmarks (16 nodes) and simulations (512 nodes), and the comparisons show the advantages and limitations of this design.

4.2 Design and implementation

In this section we firstly describe the challenges and considerations in our design. Then we present the design and implementation of the system. Finally we analyze and model the performance.

4.2.1 Challenges and design considerations.

4.2.1.1 Configurable QoS on performance. Different applications have different performance requirements, some times even one application can have different requirements when facing different scenarios. The first and most important question we face when designing the system is how to support user-configurable QoS. For storage systems, there are many ways to deliver different performance levels. Amazon EC2 and Google Compute Engine use different hardware resource (such as SSD v.s. HDD) and network bandwidth to offer different performance; some uses software-defined network to manage performance [101], some uses different consistency model in storage replication to provide different response time [100]. Many of these solutions depend on special hardware and leave users few choice. We decide to use a pure software solution, request batching, so as to avoid hardware dependency.

4.2.1.2 Batching strategy. Request batching is not a new method to achieve better system efficiency. By aggregating individual requests, a batching system can reduce total number of messages and amortize service overhead. The key question in request batching is when to send the batch. The situation is simple when there is no time limit for request delivery (latency), within network bandwidth limits, bigger batches generally bring better throughput and efficiency. If there is an inviolate request latency limit, the system designer has to give the latency limit a higher priority over the system efficiency and throughput. Various latency limits, which are usually associated with different applications, make the situation even more complicated. With this consideration, the design goal is to provide as high as possible throughput without violating the request latency limits. Dynamic request batching is a real-time scheduling problem [102]. Some theoretical works [103] have been done on various aspects of request batching. We use a modified Earliest-Deadline First (EDF) [104] algorithm in our dynamic batching mechanism.

4.2.1.3 Dynamic system performance tuning. In dynamic environments such

as clouds and data centers, network and server workload may fluctuate all the time and impact the system performance. When the network and servers are heavily loaded, to keep trying sending more requests could make the situation worse. Therefore our system needs to be smart enough to adjust the request sending rate according to the network traffic, which requires the network/server traffic detection. Since the dynamic nature makes it difficult to predict a request latency, we design a history-based heuristic approach to detect the traffic and to tune system parameters.

4.2.2 System design. We design the new system based on our previous work, ZHT [9, 10], a zero-hop distributed key-value store. ZHT follows a Memcached-like network architecture, in which servers are organized in a logical ring and each houses a contiguous key space. Clients have the knowledge of all servers and can send requests to servers by hashing the given keys. As we observed in [9, 10], when a client is sending requests in very high rate (e.g. in a tight loop), the bottleneck is actually on the client side and is bounded by the ability to handle socket connections, which is limited by kernel and CPU performance [105]. Thus in most of scenarios the servers and network are not saturated. Additionally because the client-server communication dominates round-trip latency, it would be desired to reduce message number between clients and servers.

With these consideration in mind, we propose to add a proxy layer for dynamic batching mechanism on the client side instead of server side. The client proxy works on each client, collects and batches the requests that share a same destination server and sends to the server. The destination server unpacks the batch with a parser, executes the requests sequentially, packs the return status (including lookup results) in a batch and send back. This keeps the communication and storage layers of architecture of key-value store unchanged.

4.2.2.1 Client Proxy. The client proxy architecture is shown in Fig.4.2. The client

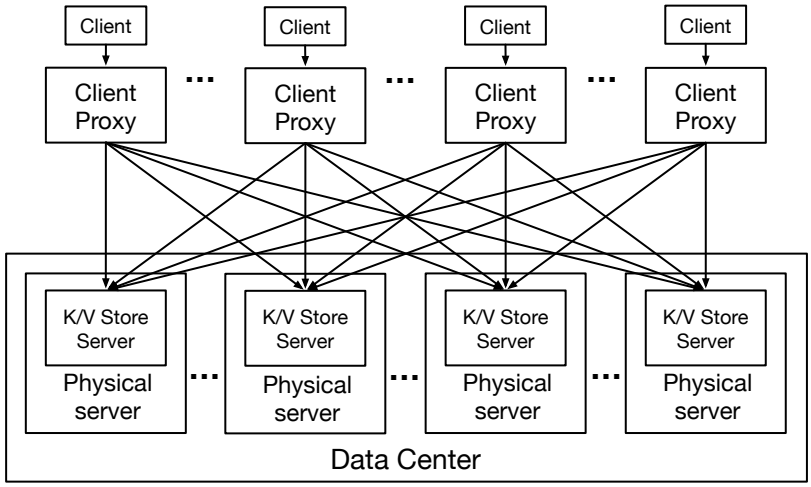


Figure 4.1. Requests are batched on client side in a proxy, which controls how and when to send a batch to a server. The servers parse batch and execute the requests sequentially, and then send batched responses back to the client.

proxy API wrapper offers the applications a set of interfaces that are compatible with conventional APIs (`put/get`). At the same time it also provides advanced controls to specify the working mode and QoS level so as to fine tune the performance.

The client proxy maintains a list of batch buckets, each of which is associated to a destination server. When a request is submitted in *single* mode, the client proxy sends it directly to the server like other key-value stores do. If the request mode is *batch*, the request handler pushes it to the batch bucket that is associated to the corresponding destination server. Then it updates the condition variables of the batch bucket according to the new request. A batch monitor checks the condition variable for each batch bucket and decide to send it or not. Apparently the condition value is the key to the batching system. It is calculated through different batching policies (Alg.4.1), which work as plugins in the client proxy. We implement multiple batching policies and discuss them in section 4.2.3.

When a server receives a batch, it sequentially executes the requests and pack the results into a new batch and then send back to the client. Note that any request

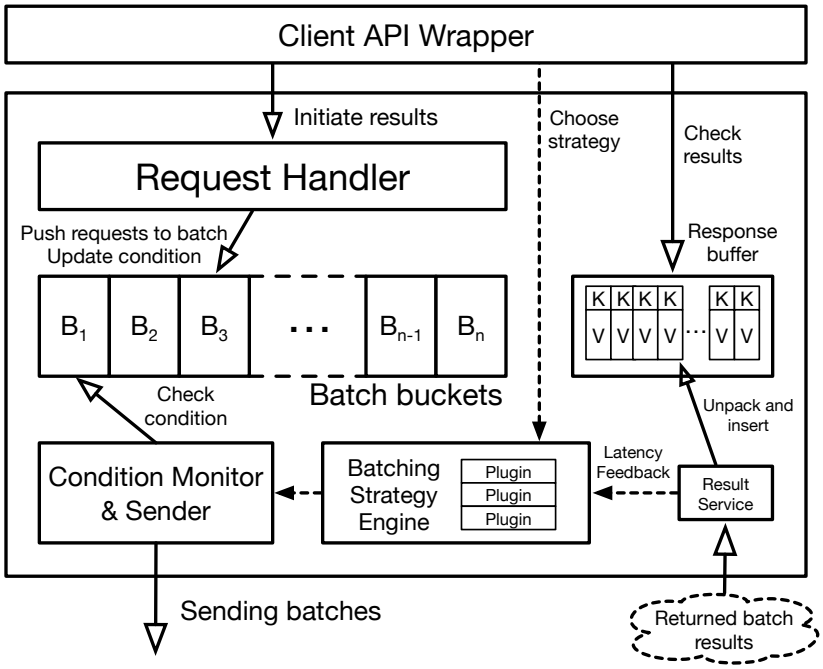


Figure 4.2. Client proxy has a list of buckets (B_1 to B_n), each of which is dedicated to one server. A monitor thread checks and sends a batch if the sending condition is satisfied. Returned response batch is unpacked and stored in a local key-value map in client proxy, the client will be notified upon the map change.

batch and response batch only involves one client/server pair.

Table 4.1. Batch request data structure

Variable Name	Description
Client_IP	For server returning results
Client_Port	Client listening port
Dest_Server	Destination server
Curr_Deadline	Current batch deadline; Condition var
Batch_num_item	# of requests; Condition var
Batch_num_limit	limit of requests #; given threshold
Batch_Size_Byte	size in bytes; Condition var
Batch_Size_limit	limit of size in bytes; given threshold
Data_Requests	List of single requests

4.2.2.2 Client APIs. Most of key-value stores' client APIs work in a synchronous manner, in which clients are blocked while waiting for the servers' response. To minimize the application change, ZHT/Q supports both synchronous and asynchronous APIs. Under the hood, in ZHT/Q's client proxy, requests are handled in non-blocking and asynchronous manner. All batch mode responses from servers are stored in a response buffer, which is a local in-memory hash map in the proxy (Fig.4.2). In this buffer map, keys and values are request keys and server responses respectively. For applications that require asynchronous access, they simply check the hash map at will, for example, to check after the QoS latency time. For synchronous applications, a dedicated thread is created in the client proxy, which blocks the application and checks the hash map within the specified QoS latency time. The application will return until response is found or a given time-out is reached.

4.2.3 Batching strategies. A batching strategy is represented by a multi-parameter trigger condition, based on which the client proxy (batcher) decides when to send a batch. ZHT/Q provides several important strategies, each of them can

work in two modes, **static** and **dynamic**. In static mode, the client proxy uses policies with user specified or system initialized parameters and thus will not change. In dynamic mode, the client proxy works with the same initial parameter set but it dynamically adjusts parameters based on currently measured performance so as to provide best-effort service when network or servers are heavily loaded (Alg.4.3).

Some applications do not have explicit QoS requirement. This type of requests can be sent in static mode with fixed batch size if the user specifies, which can fully utilize the advantage of batching. If users do not specify, these requests will be handled along with the QoS enforced requests by the system automatically with a modified Earliest-Deadline First (EDF) strategy (Alg.4.2). This strategy considers all three major parameters, namely batch latency limit (deadline), logical batch size (number of requests in a batch) and physical batch size in bytes. The batch deadline is calculated dynamically based on arrival time and QoS of every single request. The threshold values for the other two variables are given by the system administrator. A batch will be sent as soon as any one out of three conditions (batch deadline, logical batch size and physical batch size) are satisfied. In other words, the deadline for a batch is the closest deadline of all requests in that batch. EDF strategy works well to satisfy various QoS requirements. However it has a potential problem when the QoS range is very wide. The requests that have smaller QoS latency value can prevent the batching mechanism from accumulating many requests, because the system has to send batches more frequently to satisfy the smaller QoS latency.

4.2.4 QoS properties. In a request batching system, a potential problem is that a request could wait in the batch queue for a long period of time if the sending condition is not met. This could happen when the condition is not properly set or the request arrival rate is low. We avoid this problem by fortifying all batch mode requests with a maximal tolerable latency, called *QoS latency*, which can be defined in QoS or

SLA and the system is supposed to return results before that. Requests in single mode require as low as possible latency, thus they are always served immediately with the lowest possible latency (best-effort service) and no explicit QoS definition needed. ZHT/Q offers a guaranteed service when the network has no congestion and a best-effort service when network or servers are busy.

4.2.4.1 Guaranteed service. When the network and servers' processing capability are not saturated, the QoS on latency is guaranteed. Assuming a request i has a different maximal tolerable latency, denoted as l_qos_i, the request is submitted at time Tc_i, then there is a deadline d_i presented in POSIX time for the request. To ensure that the QoS of all requests in a batch are satisfied, we define the deadline of a batch $d_B$to be the closest deadline to present (T_{now}) in the batch. A given sys_cost is a threshold that is greater than the possible round-trip transferring time plus server side execution. As long as the batch is sent before $d_B + sys\_cost$, the QoS of all requests are satisfied. Then we have the lowest condition (formula 4.1) to decide when to send a batch while keeping QoS.

$$\begin{aligned} d_i &= Tc_i + l\_qos_i, \\ d_B &= \max_{i=1}^{n} d_i, \\ d_B &\leqslant T_{now} + sys\_cost \end{aligned} \tag{4.1}$$

4.2.4.2 Best-effort service with feedback based adjustment. When the network or servers are heavily loaded, the client side measured performance can degrade significantly. ZHT/Q uses passive latency detection to adjust batching parameters so as to adapt to the dynamic network environment (Alg.4.3). Latency is measured on clients for each request and compare it with an threshold value to judge if the system is running normally. The threshold latency for single and batch request mode is set in different ways. For single mode, it looks straightforward: just set to be slightly

shorter than the QoS latency. However this can cause a serious problem. On one hand, because of the delay between measured latency-based adjustment and measurable effects, and the presence of network noise, if simply using the latest measured latency as as the batching adjustment condition, the randomness of latency could lead the system to jitter. On the other hand, since individual requests are generally very small, the latency could fluctuate drastically due to the influence from client/server CPU utilization and network noise. Thus the request latency would has larger standard deviation, especially in dynamic environments such as clouds. This means there could always be a tiny part of requests are responded after the given threshold. This makes it very difficult to give a valid expected latency for threshold. To avoid this problem, we use a weighted arithmetic mean (formula 4.2b) instead of the actually measured latency, in which newer recorded latencies have higher weight. In this way, newer measured latency always play more important roles while the older latency can be used to balance the jitter.

$$\lim_{n\to\infty}\sum_{i=1}^{n}\frac{1}{2^n}=1$$

$$\bar{L_n}=\sum_{i=1}^{n}\frac{L_i}{2^n} \tag{4.2a}$$

$$\bar{L_{n+1}}=L_1/2+1/2\sum_{i=2}^{n}\frac{L_i}{2^n}$$

$$=(L_1+\bar{L_n})/2 \tag{4.2b}$$

Note that L_1 is the latest latency, L_n is the oldest latency recorded and $\bar{L_n}$ is the weighted average latency for past n requests. When n is reasonablely big, the error is negligible ($\bar{L_n}/2^n$).

In batch mode, since there is no QoS latency for batches, and the time to send a batch can not be determined before it meet the predefined sending condition, it is hard to give a reasonable threshold based on given QoS. However we can still find if

a batch is delayed. When a batch reaches the condition for sending, its physical size and logical size are known. With these sizes, we can work out a expected latency, as we have a batch latency model described in section 4.2.5.2, formula 4.7c. It is easy to derive a expected latency that we need here (formula 4.3).

$$L_{exp} = \alpha_{tr} * S_{req+res} + n * C_{exe} + C_2 \tag{4.3}$$

From formula 4.3 we can see that the batch response time is a linear function of two variables: request number (n) and total data size to transfer ($S_{req+res}$, including requests to and response from a server). Since the profiles from different network environment are different, the parameters need to be determined for all ZHT/Q deployments. Running a set of test requests with different sizes at the initial stage and measure the latency, we can calculate these parameters through linear regression.

When the measured latency (from send batch to response received, Fig.4.3) is longer than the expected duration, ZHT/Q will switch to best-effort service mode to ensure the QoS time l_qos_i is met if at all possible. In this mode, a compensatory mechanism is triggered to tune the current batching strategy to reduce latency (Fig.4.4). The predefined system cost (sys_delay) will be increased such that batches are sent more frequently. Since reducing requests in batches will benefit latency, this attempt is to sacrifice throughput for latency.

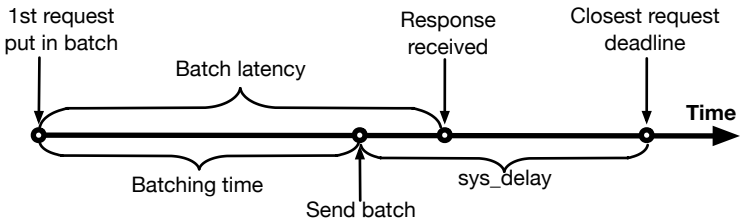


Figure 4.3. Batching events and time composition. *sys_delay* is the reserved time for batch transferring and can be automatically changed by the system to adapt traffic.

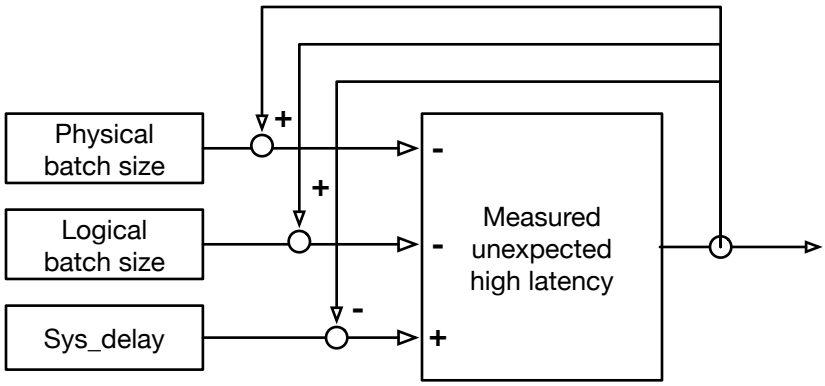


Figure 4.4. Latency feedback based parameter tuning

4.2.5 Performance model and analysis. When considering the batching problem in a dynamic environment, batching interval is the most important parameter that depends on many other parameters in the design space. It is important to understand the influence of different parameters on the system performance. For the purpose of selecting the optimal parameters in design, we formulate a model to emulate the latency and throughput for our key-value store system in both single and batch mode.

4.2.5.1 Single request processing model. In most of existing works, when running in the single request mode, each request is sent from a client to a server in a synchronous manner. Although there are works allowing write operations (`put` and `remove`) to be done in asynchronous ways, read operations (`get` or `lookup`) still need to block the clients before the results are returned. Without loss of generality, we use synchronous manner in our single request model.

The request latency consists of request and response transferring time and server side execution time. Data transferring cost can be considered as a linear function of transferring size:

$$t_{trans}(S_{trans}) = \alpha_{tr} * S_{trans} + C_{trans}$$

Table 4.2. Parameters in performance model

Parameters	Description
$t_{trans}(size)$	data transferring cost, a linear function
α_{tr}	factor of transferring cost function
t_{exe}	single request execution time on server
S_x	variables for sizes
C_x	for all kinds of constant values
$L_{s/b}$	single/batch mode latency
$Th_{s/b}$	single/batch mode throughput in ops/s
$B_{s/b}$	single/batch mode bandwidth in bytes/s

The total transferring time in a single latency is

$$t_{trans}(S_{req} + S_{res}) = \alpha_{tr} * S_{req+res} + 2C_{trans}$$

Accessing server side local hash table is done in $O(1)$, the request execution cost (t_{exe}) on the server side is considered as a linear function of request size (key-value size in bytes). Considering the cost changes negligibly (in μs) with different request size compare to the network communication overhead (in ms), the execution cost can be considered as constant, denoted by C_{exe}. Merging all constant variables to C_s we have the latency model:

$$\begin{aligned} L_s &= t_{trans}(S_{req+res}) + t_{exe}(S_{req}) \\ &= \alpha_{tr} * S_{req+res} + 2C_{trans} + C_{exe} \\ &= \alpha_{tr} * S_{req+res} + C_s \end{aligned} \tag{4.4}$$

We further have single node throughput:

$$Th_s = n_{ops}/L_s = \frac{1}{\alpha_{tr} * S_{req+res} + C_s} \tag{4.5}$$

Similarly we have single node bandwidth:

$$B_s = S_{req+res}/L_s = \frac{(S_{req+res})}{\alpha_{tr} * S_{req+res} + C_s} \tag{4.6}$$

4.2.5.2 Batch request processing model. We formulate the request batching model based on our own system. In ZHT/Q, batching mechanism is on both client and server side. The client proxy maintains a list of batches, each of which is associated to a destination server. A request handler pushes requests to their batches according to the destination. A batch monitor checks a condition value of each batch and decide to send it or not. When a server receives a batch, it sequentially executes the requests and pack the results into a new batch and then send back to the client. Apparently the condition value is the key to the batching system.

Based on this logic, we formulate the request latency. Form formula 4.7c we can see that the batch latency can be described as a linear function of two variables: request number (n) and total data size to transfer (S_{Batch} includes requests and response from a server, formula 4.7a). t_{r_int} is the interval between each request arrival. Batching time $t_{batching}$ is the time from the first request is submitted to the batch is sent. Batch waiting time t_w is the time from the last request is submitted to batch to the batch is actually sent. By running a set of test batch requests with different sizes and measuring the latency L_b and batching time $t_{batching}$, we can decide factor α_{tr}, C_{exe} and C_2 through linear regression.

$$S_{AllBatch} = \sum_{i=1}^{n} S_{(req+res)\_i} \tag{4.7a}$$

$$t_{batching} = (n-1)t_{r\_int} + n * t_{proc} + t_w \tag{4.7b}$$

$$\begin{aligned} L_{batch} &= t_{batching} + t_{trans}(S_{AllBatch}) \\ &\quad + (n-1) * C_{exe} + C_1 \\ &\approx t_{batching} + \alpha_{tr} * S_{AllBatch} \\ &\quad + n * C_{exe} + C_2 \end{aligned} \tag{4.7c}$$

Now we evaluate throughput. Different from single request mode, throughput in batch mode only needs the batch latency. Then we have the batching mode

throughput as follows. Note that the batching interval can be considered as a linear function of batch size n, as the more requests in a batch, the longer the interval is. The average arrival rate (in requests/second) is denoted by β.

$$Th_b = n/L_{batch} \tag{4.8a}$$

$$= \frac{1}{t_{r\_int} + t_{proc} + \alpha_{tr} S_{req+res} + (t_w + C_1)/n} \tag{4.8b}$$

$$t_{r\_int} = 1/\beta \tag{4.8c}$$

It is easy to derive that in the ideal case the maximal throughput of a batching system on client side is actually only related to the request arrival rate β and the network transfer rate α_{tr} (formula 4.9). This holds when the network and servers are not saturated and becomes bottlenecks. This model is validated by our experiments in section 4.3.3.

$$\lim_{n\to\infty} Th_b = \frac{1}{1/\beta + t_{proc} + \alpha_{tr} * S_{req+res} + C} \tag{4.9}$$

4.3 Performance Evaluation

In this section we evaluate the performance of our system with different batching policies and various workloads through micro benchmarks. We run real system micro benchmarks on Amazon EC2 with moderate scales (up to 32 instances) and simulations on large scales (up to 512 nodes) to measure the performance.

4.3.1 Workloads. For better coverage of different application scenarios, we define 3 types of workloads with various requirements. In all three workloads, requests are sent from clients in tight loops. Like Facebook [98] and MICA's [105] workloads, we focus on small requests with fixed key (10 bytes) and value length (20 bytes), 95% `get` and 5% `put`.

Workload with no explicit latency QoS. In this type of workload, application requests are relatively less latency-sensitive and well tolerating a wide range

of response latencies. This covers a category of applications that do not have a explicitly specified response time limit, such as logging and archiving systems. For this scenario, logical batch size (the number of requests in a batch) is the only parameter, meaning that setting a limit for batch size would work for most of cases.

Workload with single QoS latency. In this workload, key-value requests in each experiment are given same QoS latency. Assuming one application only has one QoS setting, this workload represents the scenario in which one application with many clients is served by the data store. In the test data set, the QoS time varies from 1 ms to 1000 ms.

Workload with multiple QoS latency. This workload simulates the scenario that multiple different applications use a single deployment of key-value store as a service. Note that applications in this case have various QoS latency time, ranging from 1ms to 1000 ms. The workload is organized as shown in table 4.3. Requests in workloads of pattern 1, 3 and 4 have more even QoS distribution, while pattern 2 represents a highly skewed workload.

Table 4.3. Workload with multiple QoS

QoS latency time	1ms	10ms	100ms	1000ms
Pattern 1	25%	25%	25%	25%
Pattern 2	4%	32%	32%	32%
Pattern 3	0%	33%	33%	33%
Pattern 4	0%	0%	50%	50%

4.3.2 Experiment setup and metrics. For detailed performance study, we conduct real system micro benchmark on Amazon EC2 with 2 to 32 C3.large instances, half as servers, half as clients. We separate servers from clients to avoid any local communication. For better understanding the performance and scalability on large scale deployments, we construct a PeerSim [20] based simulator. We use the data

captured from real system to calibrate and validate our simulation results (section 4.3.6).

We focus on 3 metrics that accurately reflect batching performance, namely individual request latency, batch latency and node throughput. Request latency presents the duration from a request is submitted to the response is returned by a server. Batch request presents the duration from the first request enter the batch, to the batch response is returned by a server. node throughput presents the number of finished requests in one second by the system. Please note that in individual request mode, low latency directly means high throughput. But in batch mode, it is a different story. Low average request latency imply a smaller batch size, which is not likely to make good use of the system resource. On the contrary, longer average request latency (while still within QoS) usually implies high throughput, due to the system has longer time to accumulate requests.

4.3.3 Applications with no explicit QoS. There are also many applications that have a response time expectation but do not have QoS options in their APIs, we set a maximal time limit 1 second during the experiment, by then a batch will be sent even if the batch size has not reached the threshold. The actual latency distribution is shown in table 4.4. As expected, the throughput (Fig.4.6) increases with the batch size. However it is worth to note that the throughput increasing rate is much slower than that of batch size. This is because the batching cost and the time for waiting requests are accumulated during batching. When the batch size is n, it takes $n * (t_c + t_{cost})$ time to wait and to push all the n requests into the batch, where t_c is the interval between 2 contiguously arrived requests, and t_{cost} is the time cost for processing a request in the batch. This implies a linear batching cost with logical batch size. According to formula 4.7c in section 4.2.5.2, we have predicted this overhead and the potential throughput degradation upon very large batch size.

Under this workload, as expected, the batch latency is significantly longer with larger batch size (Fig.4.5(c) and Fig.4.5(d)).

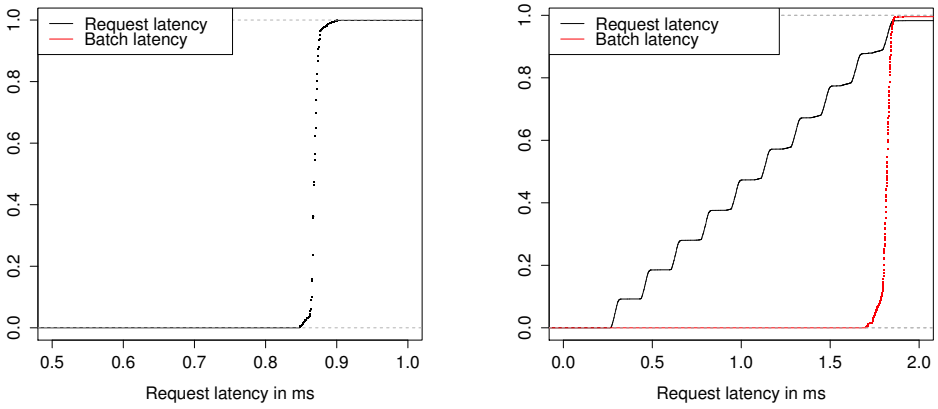
On throughput (Fig.4.6), unsurprisingly bigger batch sizes bring higher single node throughput, but the increment is not proportional to the batch size due to the accumulated batching overhead and increased data transferring cost. On different scales, the single node throughput stays consistent, which indicates excellent scalability.

Table 4.4. Latency in ms: fixed batch size

Latency Percentage	75%	90%	99%	Mean
Batch size = 1	0.871	0.874	0.880	0.869
Batch size = 10	1.484	1.803	1.850	1.087
Batch size = 100	16.18	17.93	35.196	12.252
Batch size = 1000	238.1	246.7	276.8	231.8

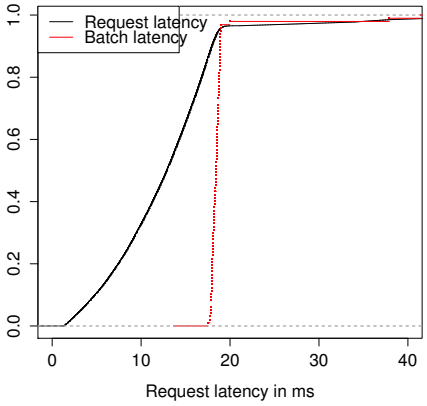
4.3.4 Single application with latency QoS. Typical applications with QoS requirement usually specify only one QoS value. This experiment represents the use case that multiple clients of single application access the data store. We can see that more than 99% requests are satisfied within the QoS time, except for the workload with very long QoS latency, in which 95% requests are satisfied. The throughput increases with the QoS time (Fig.4.8). Due to the longer QoS time, each batch can accumulate more requests before sending, which means larger batch size and throughput. This is also the reason why the throughput shows similar pattern with fixed size batching (Fig.4.6).

The measured batch latency is proportional to the specified QoS latency. Note that higher batch latency (red line) is desired because it can accumulate more requests and yield higher throughput (Fig.4.8). This also implies that if measured request latency is much shorter than QoS, it causes waste to the system efficiency. Thus

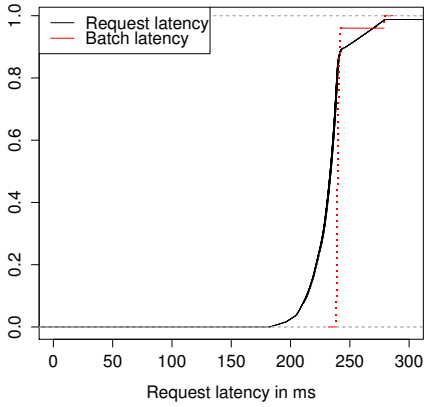


(a) Size = 1

(b) Size = 10



(c) Size = 100



(d) Size = 1000

Figure 4.5. Batching with fixed batch size. Expected longer batch latency can be observed in experiments wirh larger size (Fig.4.5(c) and Fig.4.5(d)). Note that batch latency is proportional with batch size.

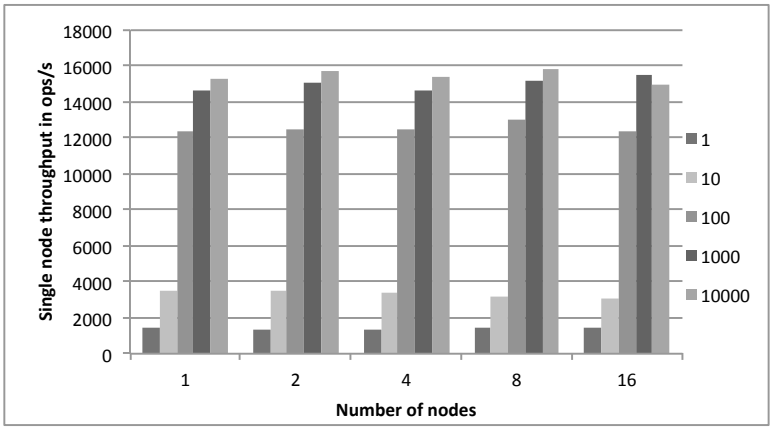
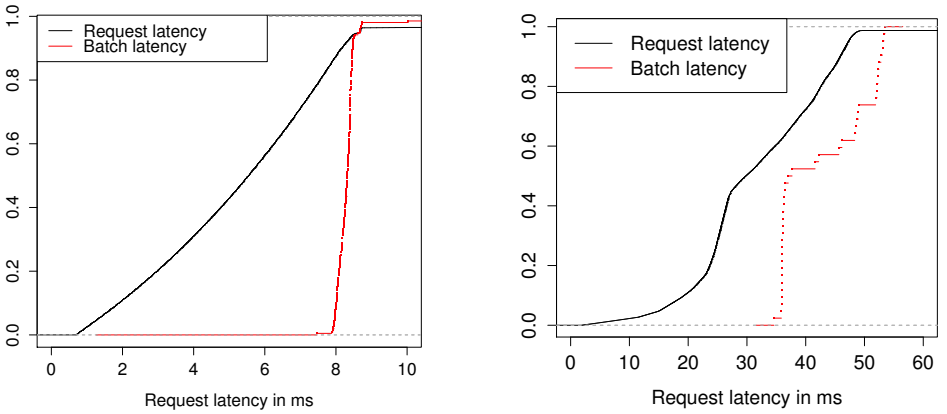


Figure 4.6. Throughput with fixed batch size. Bigger batch sizes bring higher equivalent throughput, but the increment is not linear due to accumulated batching and data transferring cost.

a "lazy" but good enough (just to satisfy QoS) batching strategy is welcomed. On throughput (Fig.4.8), similar to the trends shown in static batching (Fig.4.6), longer QoS brings higher throughput.

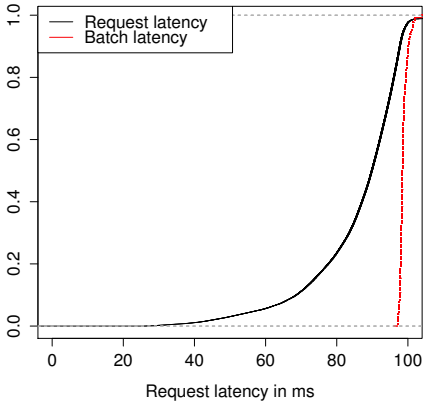
4.3.5 Multiple applications with different latency QoS. The workload that we use to test the system is organized as table 4.3. Requests have different QoS latency requirements, and are submitted in random order. Although QoS latency are mostly satisfied, the workload pattern has huge impact on throughput. In pattern 1, 3 and 4, requests QoS latencies appear with same probability, while pattern 2 presents a highly skewed workload. Similarly as the results shown in fixed batch size experiments (Fig.4.6), longer QoS latency implies larger batch size, thus higher throughput (Fig.4.10).

Interestingly we find workload pattern 2 and 3 only have 4% difference, but the throughput of workload pattern 3 is almost 3x higher (Fig.4.10), the measured batch latency (Fig.4.9) also shows almost 10x difference. On the contrary, performance profiles of pattern 1 and 2 are similar, but the workload distributions are totally different (Tab.4.3). The results shows how a small part of requests with low QoS

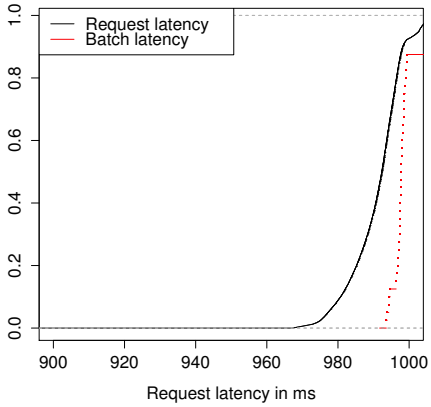


(a) QoS = 10 ms

(b) QoS = 50 ms



(c) QoS = 100 ms



(d) QoS = 1000 ms

Figure 4.7. Workload with single QoS latency represents single-application scenarios. Higher batch latency (red line) is desired because it can accumulate more requests and yield higher throughput (Fig.4.8). Batch latency is proportional to the QoS, and is close to the single request QoS, implying that the system and network are still far from being saturated.

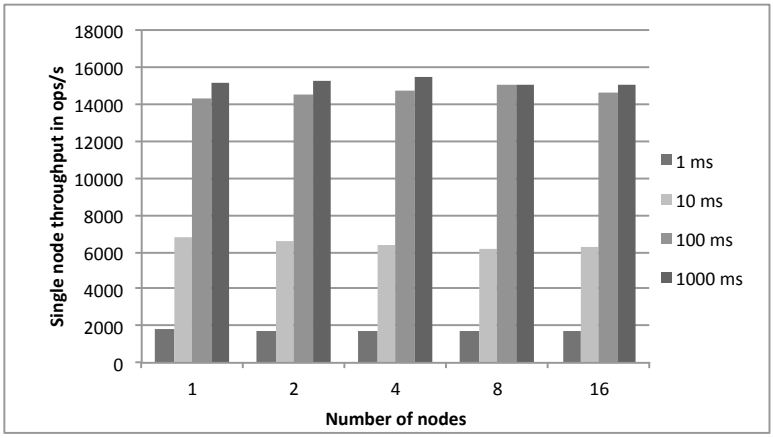


Figure 4.8. Throughput: batching with static QoS latency. Longer QoS latency requests can wait longer in the batch, which allows the system to accumulate more request thus higher throughput.

latency can significantly influence overall performance. It also remind us that EDF batching strategy still has great potential to improve.

4.3.6 Throughput comparison on large scales. In this section we discuss the experiments on large scale deployments with simulation results. We construct the simulator on top of PeerSim [20].

4.3.6.1 Simulation Validation. We firstly validate the simulation with real experimental results from fixed batch-size (Fig.4.11) and EDF dynamic batching (Fig.4.12) on Amazon EC2 cloud. The result only shows less than 5% error between real test and simulation result. This implies that the simulator can precisely predict the batching mechanism and the simulated throughput result on large scale is validated.

4.3.6.2 Simulation Results. We conduct experiments to evaluate the system scalability and total throughput with different batching mechanisms. Up to 512 nodes, both fixed-size static batching (Fig.4.13) and EDF batching (Fig.4.14) with different workloads show nearly constant single node throughput which demonstrate excellent scalability.

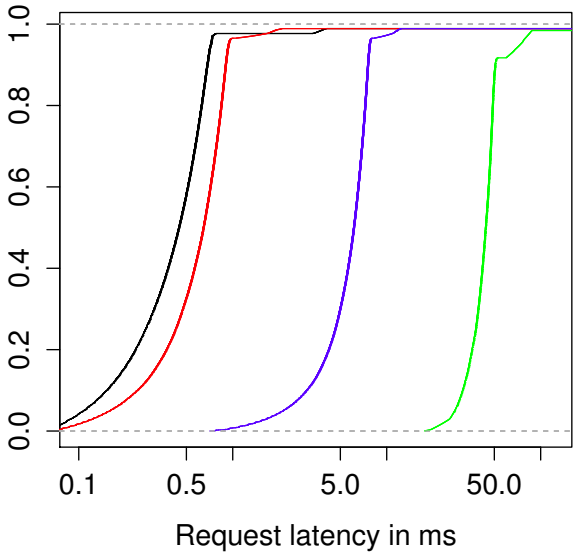


Figure 4.9. Batch latency CDF for different workloads. From left to right, the lines represent pattern 1 (black), pattern 2 (red), pattern 3 (blue) and pattern 4 (green) respectively. The lines for pattern 1 and 2 are pretty close because they both have some low QoS latency (1ms) requests, which significantly increases the batch sending frequency. Corresponding throughput is shown in Fig.4.10.

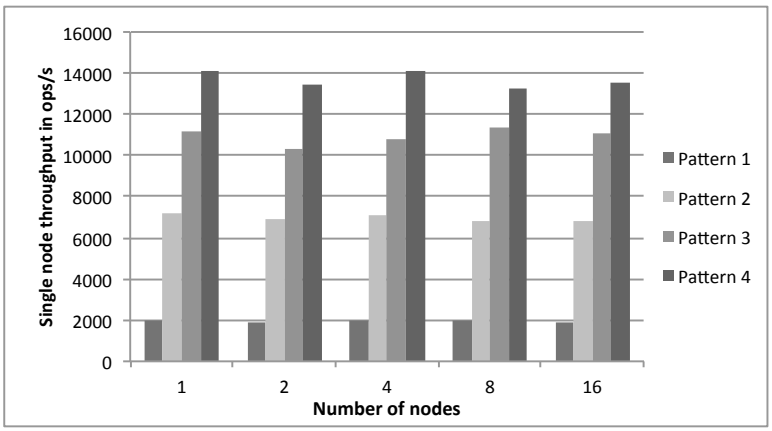


Figure 4.10. Throughput of batching with different workloads. Low QoS latency (1ms) requests in pattern 1 and 2 significantly lower the total throughput, because they force the system to send batches more frequently. Corresponding latency distribution is shown in Fig.4.9.

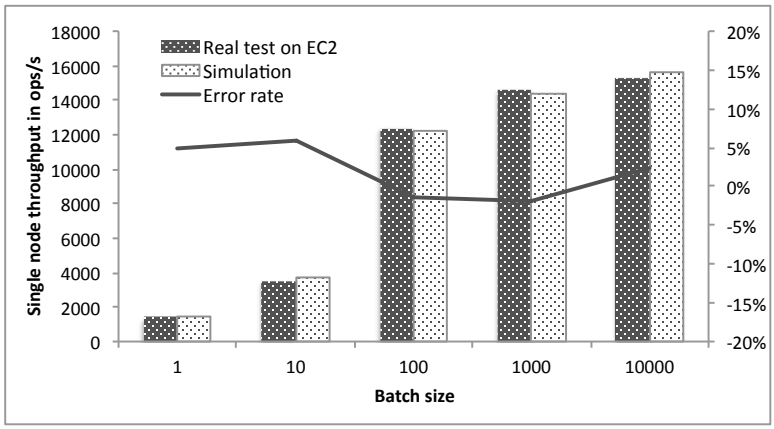


Figure 4.11. Simulation validation: batching with fixed batch size. The minimal difference between simulation result and real tested result from EC2 shows that the simulation can precisely predicts the performance of batching mechanism.

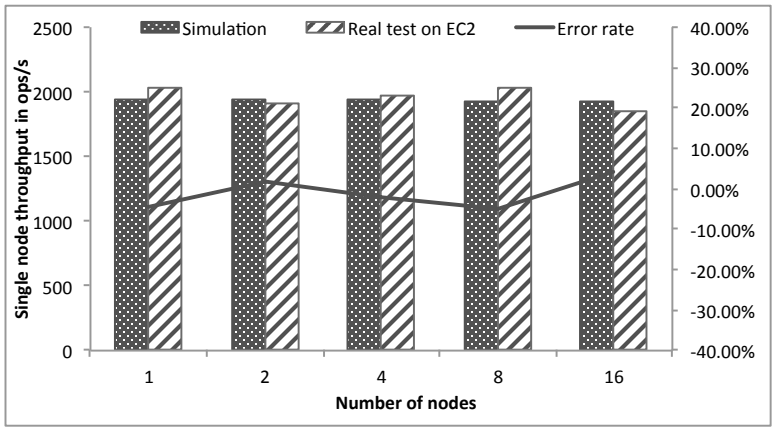


Figure 4.12. Simulation validation: EDF dynamic batching with workload pattern 1. The data captured from EC2 cloud differentiates slightly from simulation result, meaning the system scalability character is well simulated.

4.4 Summary

Every application has its own performance requirement but most of current key-value store systems are designed to serve every application request equally. In this chapter we propose a flexible distributed key-value storage system which can be used by cloud providers and data centers to satisfy various applications' QoS requirement concurrently. It uses dynamic and adaptive request batching mechanisms to achieve both QoS on latency and high aggregated throughput. The experiment results show

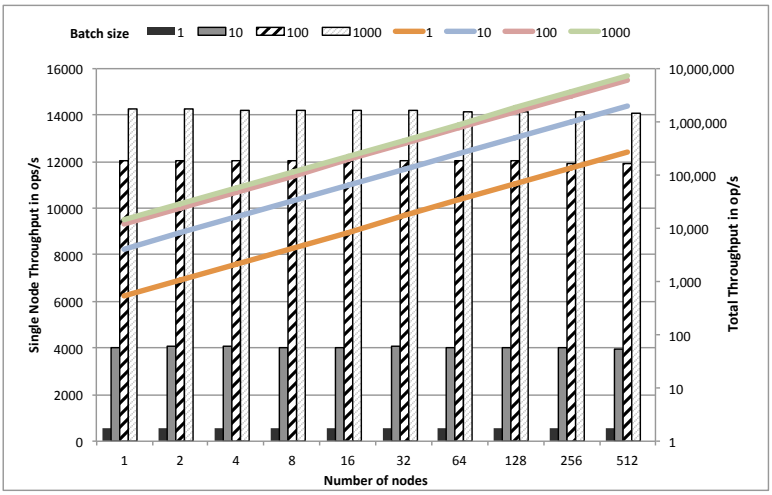


Figure 4.13. Throughput comparison on scales: fixed batch size.

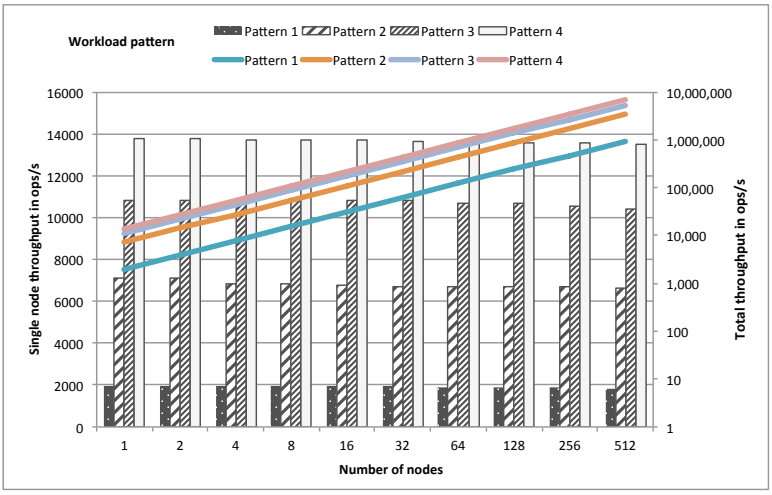


Figure 4.14. Throughput comparison on scales: workload patterns

that our new system delivers up to 28 times higher throughput than the base solution while more than 99% of requests' latency requirements are satisfied. The results also remind us that wide range of latency requirements need to be handled carefully.

Algorithm 4.1 Batch requests handling

```
1: procedure Thread_RequestHandler
2:     repeat
3:         batch ← batch.addToBatch(newReq)
4:         deadline_new ← time_newArrv + QoS_new
5:         num_req_batch ← num_req_batch + 1
6:         size_byte_batch ← size_byte_batch + size_new
7:         batch.mutex_lock()
8:         if deadline_batch ≥ deadline_new then
9:             deadline_batch ← deadline_new
10:        end if
11:        batch.mutex_unlock()
12:    until Terminated
13: end procedure
14:
15: procedure Thread_Monitor
16:    repeat
17:        for all batch in BatchList do
18:            batch.mutex_lock()
19:            if condition_send(policy) = True then
20:                sendBatch()
21:                deadline_batch ← ∞
22:                batch.requests ← φ
23:            end if
24:            batch.mutex_unlock()
25:        end for
26:    until Terminated
27: end procedure
```

Algorithm 4.2 Earliest-Deadline First Batching

1: **procedure** $condition_{send}$

2: **if** $dl_{batch} \leq sys_{delay} + time_{cur} \| num\_req_{batch} \geq max\_req \| size\_byte_{batch} \geq max\_size_{batch}$

3: **return** True

4: **else**

5: $deadline_{batch} \leftarrow \infty$

6: $batch.requests \leftarrow \phi$

7: **return** False

8: **endif**

9: **end procedure**

Algorithm 4.3 Dynamic parameter tuning

```
1: function ParameterTuner(L, size_n, size_b, sys_delay)
2:     if L > ExpectedLatency() then
3:         size_n ← size_n/2
4:         size_b ← size_b/2
5:         sys_delay ← 2 * sys_delay
6:     end if
7: end function
8: function ExpectedLatency(L_cur, L̄)
9:     if isIndividualRequest then
10:        ExpectedLatency ← WeightedAvgLatency()
11:    else
12:        ExpectedLatency ← ExpectedBatchLatency()
13:    end if
14: return ExpectedLatency
15: end function
16: function WeightedAvgLatency(L_cur, L̄)
17:    if L̄ = 0 then
18:        L̄ ← L_cur
19:    else
20:        L̄ ← (L_cur + L̄)/2
21:    end ifreturn L̄
22: end function
```

CHAPTER 5

OTHER NOSQL DATABASES FOR LARGE SCALE APPLICATION SYSTEMS

5.1 Introduction

In the cloud era people are building increasingly bigger distributed application systems for different purposes. In recent days complex distributed applications can be developed and deployed more easily due to the presence of a wide variety of components and frameworks, such as distributed data storage systems, distributed message queues, publish-subscribe systems and so forth. Data storage is one of the most important. This category covers a wide range of systems, such as file systems, SQL databases, NoSQL databases, blob stores, object stores and so forth. They play critical roles in stateful distributed system design and development, as most of such systems need globally accessible storage, such as a database or a file system. Especially NoSQL databases, with their help, the users atop can write their applications or upper layer easily while enjoy the advantages in performance, capacity and scalability.

In thie chapter, we discuss two systems that are designed and implemented with another large category of scalable distributed NoSQL databases, called column-oriented databases (or BigTable-like data stores), such as Cassandra [12] and BigTable [106]. They provide richer features such as complex query and flexible schemes. As a cost, their performance in terms of latency and throughput are generally not as good as key-value store.

5.2 State Management for Scientific Applications on Cloud

The data generated by scientific simulations and experimental facilities is beginning to revolutionize the infrastructure support needed by these applications. The on-demand aspect and flexibility of cloud computing environments makes it an attractive platform for data- intensive scientific applications. However, cloud computing

poses unique challenges for these applications. For example, cloud computing environments are heterogeneous, dynamic and non-persistent which can make reproducibility a challenge. The volume, velocity, variety, veracity and value of data combined with the characteristics of cloud environment make it important to track the state of execution data and application's entire lifetime information to understand and ensure reproducibility. This chapter proposes and implements a state management system (FRIEDA-State) for high-throughput and data-intensive scientific applications running in cloud environments. Our design addresses the challenges of state management in cloud environments and offers various configurations. Our implementation is built on top of FRIEDA (Flexible Robust Intelligent Elastic Data Management), a data management and execution framework for cloud environments. Our experiment results on two cloud test beds (FutureGrid and Amazon) show that the proposed solution has a minimal overhead (1.2ms/operation at a scale of 64 virtual machines) and is suitable for state management in cloud environments.

5.2.1 Introduction. Data analysis is central to next-generation scientific discoveries. Cloud is as an emerging platform and increasingly attractive to scientists due to its flexibility and convenience. But cloud environments are typically transient. Virtual machine instances are terminated after applications complete execution. Users cannot leave data and/or revisit the resource setup to diagnose discrepancies. In the cloud environment, users have the responsibility to capture everything before the virtual machines are shutdown.

Big data scientific applications need to track every step of the scientific process, data access and environment for lineage, reconstruction, validity and reproducibility purposes. It is important to know the environment in which the applications run (e.g., floating point operations could give different results on different machines). Users might also wish to "rerun" some (e.g., only what failed) or all of the tasks.

Provenance tools have tracked workflow and data lineage at various levels (e.g., operating system [107], file systems[108], databases[109], and workflow tools ([95,110–112]. Many monitoring tools ([61,113–120]) have been developed to monitor real-time system changes. These systems provide methods to collect, aggregate, and query monitoring data. However, this data is often insufficient for reproduction since they do not capture human knowledge. Furthermore, state management in cloud environments needs to tackle additional challenges due to its characteristics. First, the transient nature of the environments makes it important to capture metadata and state at various levels. Second, the performance and reliability characteristics of virtual machines is important to consider in the design of the collection system. Finally, different clock drifting rates on physical machines make it hard to have a unified time view for the end-user to rebuild meaningful semantics.

In this chapter, we propose FRIEDA-State, a state management system for cloud environments. We use the term state to represent the metadata from both execution framework and applications. FRIEDA-State addresses the transient nature, performance concerns and clock drifting issue[121] in its design. FRIEDA-State is currently implemented atop of FRIEDA [122], a data management and execution framework for cloud environments, which supports a high-throughput and data-intensive scientific applications,. We present a key-value based collection system to manage state in dynamic transient environments. We design and implement a vector clock[123] based event-ordering mechanism to address the clock drifting issue.

FRIEDA-State collects static and dynamic state data. Static state data is the information that doesnt change when the system is running (e.g., CPU/Memory info, environment variables and software stack information). Dynamic state data, on the other hand, changes during application running, such as the information on details of the input file that is processed, the time taken for a machine to finish execution or

failure of jobs.

Specifically, the contributions of our work are:

- Design and implementation of FRIEDA-State, a state management system for scientific applications running in cloud environments, with lightweight capturing, efficient storage and vector clock-based event ordering
- Evaluation on multiple platforms (FutureGrid and Amazon EC2) at scales of up to 64 VMs; results show good efficiency with minimal overhead (1.2ms/operation at 64-node scales)

5.2.2 Use cases. Scientific applications need to track their scientific process for a number of reasons including a) real-time monitoring b) tracking data lineage c) validation of results d) reconstruction or repeating some or all of the experiments and, e) reproducibility of research results. Users might want to track their configuration and environment settings and repeat some or all of the experiment or validate a certain result (e.g., floating point). The state information might also be used for post-execution analysis. For example, the users might like to query job statistics and understand why some jobs took longer than others. Users might want to rerun the same experiment and/or run the same experiment with slightly different parameters.

5.2.3 Challenges. Next, we discuss the challenges on state collection, storage, and event synchronization. State collection and storage: State information is generated on each of VMs and multiple VMs are part of an application execution. High capture latency may degrade the application performance. Thus, scalable collection of data is important in the design of FRIEDA-State. Information aggregation and appropriate storage mechanisms are also important and different solutions might have different trade-offs. Centralized storage system (e.g., databases) could result in concurrent read/write bottlenecks and be the source of single-point failures. Distributed solutions

often suffer from high operation latency and often require extra dedicated hardware. Cloud environments are dynamic, virtual machines may not run on the same physical machines. This implies that the physical time clocks may not run at exactly the same speed because of the slight difference between crystal oscillators on different machines thus result in drifting [121]. With different drifting rates, at the end of a long run, the base-time gap between each virtual machine could be big. The drifting issue is serious in large-scale distributed systems and is even more serious due to transient nature of clouds. Synchronized bootstrap time clocks may not be guaranteed in distributed cloud environments. It is important that the events/states captured on the machines is unified for the end-user to build meaningful semantics.

5.2.4 FRIEDA framework. Our state management system is built on top of FRIEDA[122]. FRIEDA is a Flexible Robust Intelligent Elastic Data Management framework. FRIEDA manages the lifecycle of data that includes storage planning and provisioning, data placement and application execution of scientific applications in cloud environments. Similar scientific application execution frameworks include SciMATE and Smart [35, 124–128].

FRIEDA enables users to plug-in flexible data management strategies for different application patterns by separating data control from execution. FRIEDA supports a Master-Worker execution model. There are three major components in FRIEDA architecture, namely controller, master and workers. The controller takes charge of environment setup and configurations for data management and application execution. The master is responsible for managing application execution and data distribution. The workers accept data and computation jobs from the master and execute them locally. After all workers finish their jobs, the framework will collect output data from all nodes. State management system collects information from the resource provisioning and execution phases.

5.2.5 System Design and Implementation. Figure 5.1 shows the system architecture of the state management system (FRIEDA-State). FRIEDA-State has a collection component in each of the main FRIEDA actors the controller, master and worker. FRIEDA-State works in two phases: capturing and storage. It allows multiple storage solutions to be plugged into the framework to meet different usage needs. The runtime state capture component collects two types of state: static and dynamic. The static states are collected mainly from a configuration YAML file, which is used in FRIEDA to configure the virtual machines (e.g. it defines the roles and the setups of the master and workers). The YAML file is populated in the state management system from the controller node once the experiment starts. The remaining static information (e.g. system information) is collected from the worker virtual machines directly. Dynamic states are captured from the FRIEDA framework through built-in functions. Once captured, states are encapsulated in key-value pairs and pushed to one of three storage solutions that is selected by the user. FRIEDA-State currently supports raw files, Cassandra or DynamoDB (on Amazon Web Services).

5.2.5.1 State Description. Each state in our system has the following fields. The state name is used as the key and the rest are used as values.

State name. This is used to represent the type of event.

State information. This field captures the state content.

Role. This field captures the source of the event or the role of the host (i.e., master or worker),

Hostname/IP. This captures the identity of the host where the state was collected.

Logical timestamp. We set a field for logical time for ordering the events captured from distributed nodes. The logical timestamps is used as a part of vector

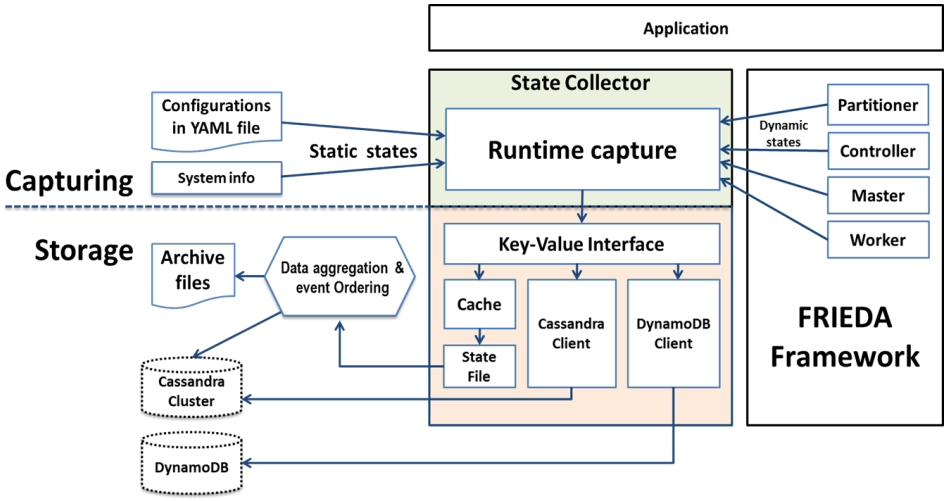


Figure 5.1. FRIEDA-State system architecture. Capturing is the first layer. State collector captures static states from two source, configuration YAML file and system information. Dynamic states are captured from FRIEDA-State functions, which are called in FRIEDA framework. Captured states are encapsulated into the form of key-value pairs and pushed to one of three storage solutions as selected by the user.

clock.

Local timestamp. The local timestamp is also captured that indicates the time when the event is captured on a local host and be used to order events within a virtual machine.

5.2.5.2 Static state capture. Static state represents the data that will not change during application execution. In FRIEDA, most of the static states are covered in a configuration YAML file. The YAML file includes platform name, image ID, instance type, authentication information, application details. The YAML file allows users to setup environment, software installation and the application running details. The YAML file is loaded into memory and stored as structured data items and then dumped into a data store or state file as a record. Other static states, such as hardware info (CPUINFO/MEMINFO), software stack information and so on are captured when the virtual machines are launched.

5.2.5.3 Dynamic state capture. Dynamic state information represents the data that changes during the run-time of applications. For example, the identification of input files processed by a worker and the duration of the execution are captured by FRIEDA-State. We capture two types of information a) communication events and b) application execution details.

All communication events, such as connection made (and to which machine), data received (and from where), etc., are captured. These events not only describe the communication itself, but also carry vector clock information for later event reordering (section III.F). Application execution and data flow details include the commands executed on each worker, I/O operations and application execution time etc. This can be used to track the application execution and to analyze run-time problems.

5.2.5.4 State Storage. The essence of state management is to capture data in distributed environment and store for future queries. For designing such a scalable system with low latency, the major concern is storage architecture. The state operation latency must be very low to prevent degradation of the application performance. Scalability is also important since the storage system could be a bottleneck when serving many clients for writing and/or query. FRIEDA-State currently supports three storage options: files, Cassandra data store and DynamoDB (on Amazon). This allows users and applications to select the right storage while accounting for the tradeoffs for their needs.

Files. This mode uses files for capturing state. Captured states are first written to files, which are later aggregated from all machines at the end of execution. Files as a storage mechanism provides some advantages over key-value stores and databases. First, simple memory-file operation is significantly faster than single node key-value stores, due to the fact that memory-file operation executes sequential writes while key-value stores execute writes randomly (hash table or B-Tree). Second, file-

based mechanisms do not require any additional services and hence does not add any overheads on the nodes. Third, merging files are simple and fast since the files are already naturally ordered due to sequential writes on each node. Therefore sorting the state files has a linear time complexity. Files are not good to query on, but they are easy to manipulate and archive.

We capture states from all FRIEDA components on VMs and store them in an in-memory cache before being flushed to disks. The cache size is customizable to address the tradeoffs between robustness and performance. If application fails, the states can still be found on those VMs since they are flushed to disk. For even better durability, it can write to a distributed file system or a block store that can survive beyond the life of the virtual machine, based on the configuration. Finally all states files are copied to a target machine for merging. If application fails, states are still saved within the FRIEDA-State framework. But if VMs or FRIEDA fails during execution, users might lose unsaved states.

Cassandra. We include Cassandra as one of the storage solutions[12], as it provides rich features for managing semi-structured data. It is easy to plug-in other NoSQL databases in FRIEDA-State. Users control the number of Cassandra instances according to their performance and capacity needs. Cassandra could share the virtual machines with the application or run on a separate cluster. Key-value stores are known to work well when deployed on dedicated machines. Practically, users can use a private cluster to host the Cassandra cluster. They can also setup a dedicated virtual cluster on cloud to serve the requests. In this case, users will need to periodically move state data to a more permanent storage.

DynamoDB. The third storage solution is based on DynamoDB, a NoSQL database available on Amazon Web Services. With this type of cloud databases services, users dont have to deploy a software stack to run and configure those data

stores, but have to pay extra money for the service.

5.2.5.5 Storage architectures. FRIEDA-State supports multiple storage architectures, namely centralized, distributed and local storage. This sub-section will describe each of these architectures in detail.

Centralized Storage Solution. In our current implementation, centralized solution is based on a single node key-value store or database. When all nodes in the system generate states, collecting and storing can result in a storage bottleneck. Depending on the application type, the rate of state generation can be different. If the data generated is minimal and at a low rate, centralized storage solution will work perfectly in practice. An important advantage of centralized solution is that all events can be naturally ordered as they arrive at the storage server and assigned a timestamp based on servers local time. The solution naturally provides persistence of the state data beyond the lifetime of the virtual machines.

Distributed storage solution. Similar to centralized solution, FRIEDA-State support distributed storage solution through NoSQL databases (e.g., Cassandra). High write concurrency is a big challenge for all types of storage systems. Distributed storage solutions, such as distributed databases, key-value stores can serve large amounts of write requests and spread them to many nodes to achieve scalable performance and load balancing. In this type of solution, a group of dedicated data storage servers will be started prior to application execution. States generated on any node in the system will be written remotely to the data store. The latency of this operation depends on the data store solution and could be up to a few milliseconds. To deploy data stores on all the nodes that will generate states will not help much on performance, because running the data stores consumes extra CPU and memory resources, and messages still need to be sent between all VMs over the network.

Local storage solution. We implemented this solution based on traditional files. Local storage eliminates network latency (the major part of operation latency).. Considering the data collected is likely to be queried after an application run, offline storage solutions are reasonable. Writing to local disk/memory is extremely fast compared to remote access and no extra resources are consumed by data store programs on VMs. The hard part of using local storage is aggregating data from many VMs and to merge into a form that offers a single query interface. File-based solution is not perfect though. If users want to query the states during the system running, extracting the desired records is complicated and slow.

5.2.5.6 Event reordering. Distributed event ordering is an important topic in large-scale distributed system design. The goal is to keep a global logical order of events based on timestamps. In FRIEDA-State, we use modified vector clock[123] to maintain time order. FRIEDA-State uses 1-to-n communication pattern since communication only happens between master and workers. We use the master node clock as major clock and all workers logically synchronize to it using vector clock. By comparing attached master clock value in communication messages, we can tell which event happened earlier. If the master clock reads are same, then these events occurred in the same machine. It is trivial to order events within one machine by sorting the local timestamps. When using file-based storage solutions, events are sequentially written to files and thus are naturally ordered. Each state record has two fields for vector clock: one for local clock, another for master clock. Events on master node have same values for both clocks.

In the beginning, all logical clocks are set to be 0. Once a new state is captured and stored, the corresponding clock value is increased. The local clocks increase naturally along with the events happening, and the master clock can only be updated when a message is received that contains a new master clock value. Workers states

collection can be divided into independent event groups by master clock value. In each group, events are naturally ordered and do not interleave with those in other groups. The possible causal relation between different groups, if exist, is determined by master clock. Thus, the problem of reordering and merging different events is reduced to sorting the event groups. Sorting groups is simple and has the same time complexity as the merging phase in merge-sort, namely O(n), where n is number of groups. For example, in 5.2, sorting by master clock M value, all the events are divided into five groups. Each group presents an atomic sequence in a machine. Since the inner events of a group are naturally ordered, the reordering is efficient.

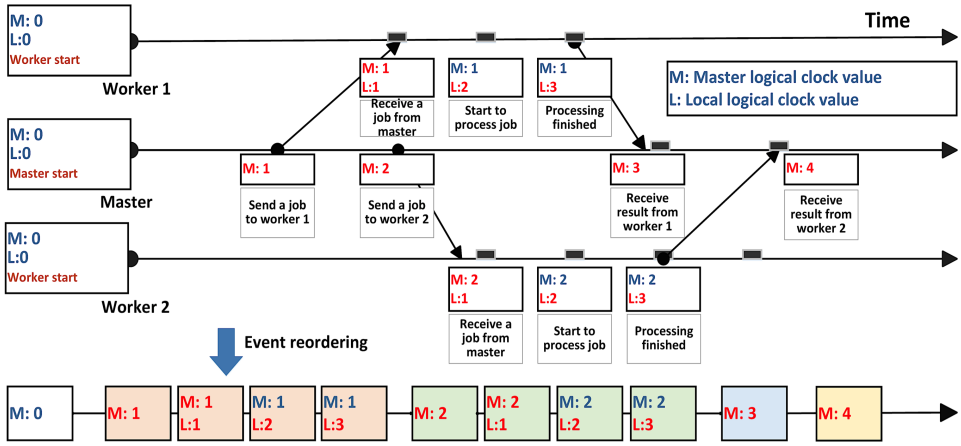


Figure 5.2. Event reordering example. Three machines have their local clock and maintain a vector clock. Each event will increase the local clock value; each received message will update others clock value in their local vector. The masters clock is used to maintain the time order when reordering the events. Sorting by master clock M value, all the events are divided into 5 groups. Just sort the groups will order all the event.

5.2.6 Evaluation. In this section, we describe the performance of the state management system, with different storage solutions.

5.2.6.1 Testbeds.

- FutureGrid Sierra, a research purpose public cloud, experiments used up to 16 virtual machines.

- Amazon EC2 cloud, up to 128 c1.medium virtual machines, each has 2 virtual CPU, 5 elastic compute units and 1.7GB memory.
- DataSys, an 8-core x64 server at IIT, dual Intel Xeon quad-core w/ HT processors, 48 GB RAM. This machine is used for experiment to study the merging overhead.

5.2.6.2 Scientific Workloads. We use two applications that are representatives of scientific workloads using cloud environments: Image Comparison and Event Processing. Image Comparison compares an image with other images in the set to find similarities. These applications are representative of typical data processing scientific workflows.

5.2.6.3 Experiment Setup. We use the same workload for three storage solutions, respectively based on files, Cassandra and DynamoDB. For synthetic benchmarks, on each state client, we send 10K requests in a tight loop to simulate an extremely operation-intensive scenario. Each request consists of 20 bytes key and 80 bytes value. Both key and value are randomly generated.

File-based solution. For file-based solution, the key-value pairs are saved to a local file on each client. Next, all these files are copied to a shared NFS directory, located on a dedicated VM where the files are merged. This is a simple solution for demonstrating state aggregation. Apparently its vulnerability to single point failures and the bottleneck can be addressed by well-known techniques such as mirroring or parallel file systems. We measure the time of writing to files, moving files to NFS server and merging events. We amortize the cost of file moving to state storage to get the average equivalent latency per state.

Cassandra-based solution. We use 1 to 8 Cassandra servers on dedicated VMs, and send requests from 1 to 128 state clients on VMs.

DynamoDB-based solution. DynamoDB is a service provided by Amazon. The data servers dont need to be deployed on the VMs. The VMs only need to communicate to the remote Amazon data stores and this has a minimal performance impact on local VMs. We provision the maximum available throughput for DynamoDB, which is 10K ops/sec. Up to 128 VMs on Amazon EC2 are used as state clients to send requests.

5.2.6.4 Metrics.

The metrics measured and reported are average latency and throughput.

Average Latency. We consider the average latency as per request to write a state to data stores, measured in microseconds. Note that the latency includes the round trip communication and storage access time. Measuring latency for Cassandra and DynamoDB is straightforward, but file-based solution needs more care. We use the formula below to calculate the average equivalent latency t_{ave} for file-based solution, where t_w is the average file write latency, T_{moving} and $T_{merging}$ are the total time spent on moving and merging respectively, n is the total number of operations:
$t_{ave} = t_w + (T_{moving} + T_{merging})/n$

Throughput. The number of operations the system can handle over some period of time, measured in Operations per seconds (Ops per second).

5.2.6.5 Synthetic benchmark.

Capture overhead. We conduct micro benchmarks on scales of up to 128 VMs. Note that the latency of file-based state management includes amortized cost for file moving, reordering and merging. Cassandra data stores crashed frequently and cannot serve requests at a scale of 8 servers with 128 clients. Similarly, DynamoDBs maximal throughput is reached at this scale and started to give errors, thus we only show results at a scale of 64 clients. Figure 5.3 shows that file-based state solution

has significant advantage over other two capture methods. When clients number increases, moving files to a single server causes contention. But this cost gets amortized across all requests. A single node Cassandra is saturated with 8 clients. On larger scales, multiple servers show some benefits but it is still limited, compared to the file based approach. DynamoDB shows very stable performance when facing different client request pressure before it is saturated, but its at least three times slower than Cassandra at most scales.

Similar to the latencies, file-based solution delivers significantly higher throughput than other two. At 64 nodes, file-based solution achieves 52K ops/s, which is five times faster than a dedicated eight nodes Cassandra cluster and 18 times faster than DynamoDB based solution.

Events reordering and file merging overhead. On an 8-core Xeon server, we generate up to 512 state files. Each state file contains 10000 state records. Using a simple single thread merging program, 4 files cost 16ms, and 512 files cost 8209 ms. This can be further improved with more sophisticated merging algorithms in the future.

File-based state management. We measured the time for capturing state and writing to file, moving and merging files respectively. We set an in-memory cache to boost the disk write performance. As shown in Figure 5.6 a full-size cache setting brings around 10% performance increase.

Since the state capturing on a local machine doesnt involve any contention, the latency is actually constant, around 500us. Simultaneously moving a large number of files can cause contention, either on network or disks. The time spent on moving files keeps increasing even when the time is amortized. Better methods to aggregate data will be needed when running at larger scales. Merging overhead increases as well, but

still negligible compare to other two overheads.

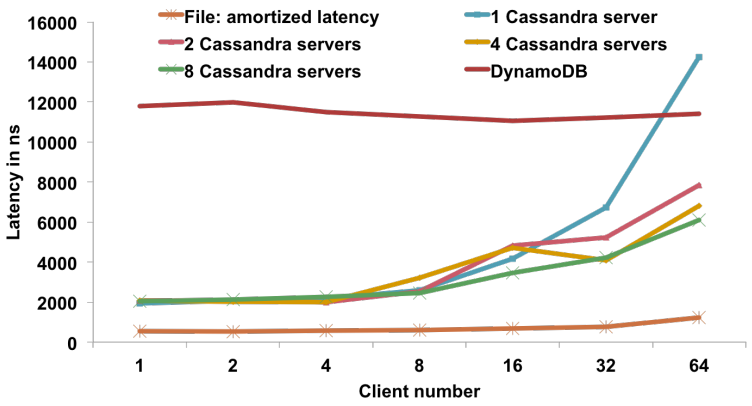


Figure 5.3. File-based solution has lower average latency. Cassandra performance decreases with the scale. DynomoDB latency doesnt change much with scale, but failed 128 clients test.

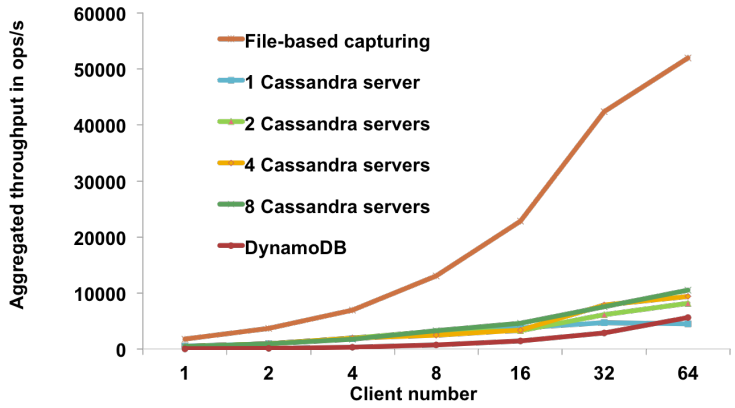


Figure 5.4. System throughput comparison. File-based solution obtains the maximum aggregated performance increases with scale.

5.2.6.6 Scientific applications. With integrated state management system in FRIEDA, we run two scientific applications (Image Comparison and Event Processing) to evaluate the overall performance impact of state collecting on real applications. Both applications are evaluated on FutureGrid [129] system.

Although in synthetic benchmarks we observed huge difference of performance among different storage solutions, in application tests, we see no significant difference

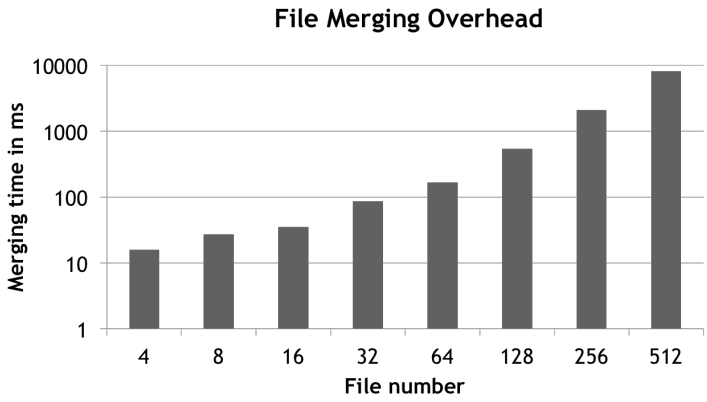


Figure 5.5. File merging time becomes longer with number of files.

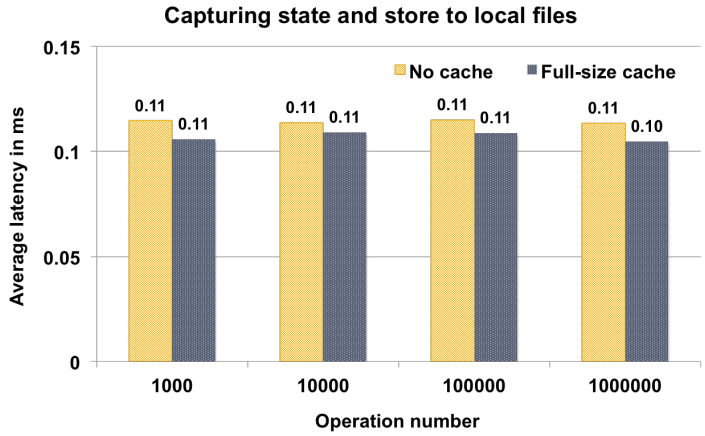


Figure 5.6. File write operation scales well. Full-size cache brings around 10% performance gains

(state-introduced overhead is less than 5%). This is mainly because the micro benchmark tests execute operations in a tight loop while real applications have sparser and random patterns, so the total time spent on state management is very low compared to the application running time.

5.2.7 Summary. Scientific applications are increasingly using cloud environments and need a way to track the applications entire lifetime information both for monitoring and ensuring reproducibility. We propose and implement a state management system (FRIEDA-State) for a broad type of scientific applications running in cloud

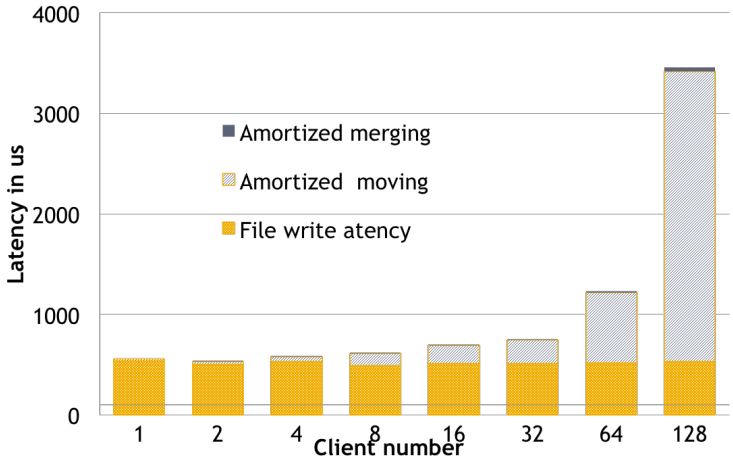


Figure 5.7. File-based storage write latency is constant while merging time is increasing slightly. The amortized moving time increases exponentially.

environments. FRIEDA-State has an innovative design that allows various storage mechanisms to be plugged-in while providing different trade-offs in durability, performance and usability. In this section, we discussed our implementations based on files, Cassandra and DynamoDB respectively and evaluated them on two cloud platforms. The evaluation showed that FRIEDAState has very low overhead even when running at a scale of 64 virtual machines. File-based storage solution offers significantly better performance than key-value stores (e.g. Cassandra) on moderate scales. Furthermore, in some conditions, file-based storage is better than cloud databases services (e.g. DynamoDB) as well, in terms of latency and aggregated throughput. The major part of overhead of file-based storage solution is file moving, when using a centralized data server. Further scalability can be achieved with better merging algorithms for file-based systems or deploying larger number of NoSQL data nodes. We expect that as we increase scale into 100s and 1000s of VMs, that the centralized data server will become a bottleneck, and distributed key-value stores would begin to offer better performance.

5.3 Scalable Cloud Data Infrastructure for Sensor Networks

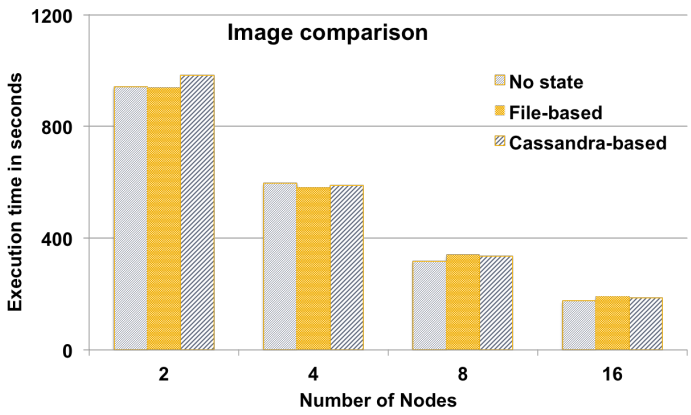


Figure 5.8. Application: image comparison

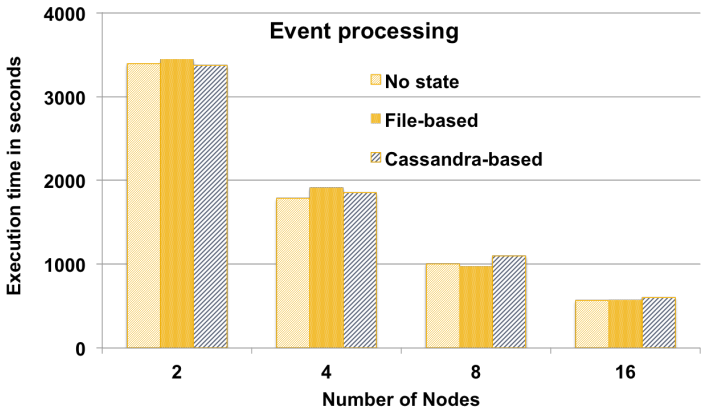


Figure 5.9. Application: event processing

As small, specialized sensor devices become more ubiquitous, reliable, and cheap, increasingly more domain sciences are creating "instruments at large" - dynamic, often self-organizing, groups of sensors whose outputs are capable of being aggregated and correlated to support experiments organized around specific questions. This calls for an infrastructure able to collect, store, query, and process data set from sensor networks. The design and development of such infrastructure faces several challenges. The challenges reflect the need to interact with and administer the sensors remotely. The sensors may be deployed in inaccessible places and have only intermittent network connectivity due to power conservation and other factors.

This requires communication protocols that can withstand unreliable networks as well as an administrative interface to sensor controller. Further, the system has to be scalable, i.e., capable of ultimately dealing with potentially large numbers of data producing sensors. It also needs to be able to organize many different data types efficiently. And finally, it also needs to scale in the number of queries and processing requests. In this work we present a set of protocols and a cloud-based data streaming infrastructure called WaggleDB that address those challenges. The system efficiently aggregates and stores data from sensor networks and enables the users to query those data sets. It address the challenges above with a scalable multi-tier architecture, which is designed in such way that each tier can be scaled by adding more independent resources provisioned on-demand in the cloud.

5.3.1 Introduction. The last several years have seen a raise in the use of sensors, actuators and their networks for sensing, monitoring and interacting with the environment. There is a proliferation of small, cheap and robust sensors for measuring various physical, chemical and biological characteristics of the environment that open up novel and reliable methods for monitoring qualities ranging from the geophysical variables, soil conditions, air and water quality monitoring to growth and decay of vegetation. Structured deployments, such as the global network of flux towers, are being augmented by innovative use of personal mobile devices [130–143], use of data from social networks, and even citizen science. In other words, rather than construct a single instrument comprised of millions of sensors, a "virtual instrument" might comprise dynamic, potentially ad hoc groups of sensors capable of operating independently but also capable of being combined to answer targeted questions. Projects organized around this approach represent important areas ranging from ocean sciences, ecology, urban construction and research, to hydrology. This calls for an infrastructure able to collect, store, query, and process data set from sensor networks. The design and development of such infrastructure faces several challenges. The first group of chal-

lenges reflects the need to interact with and administer the sensors remotely. The sensors may be deployed in inaccessible places and have only intermittent network connectivity due to power conservation and other factors. This requires communication protocols that can withstand unreliable networks as well as an administrative interface to sensor controller. Further, the system has to be scalable, i.e., capable of ultimately dealing with potentially large numbers of data producing sensors. It also needs to be able to organize many different data types efficiently. And finally, it also needs to scale in the number of queries and processing requests. In this section we present a set of protocols and a cloud-based data store called WaggleDB that address those challenges. The system efficiently aggregates and stores data from sensor networks and enables the users to query those data sets. It address the challenges above with a scalable multi-tier architecture, which is designed in such way that each tier can be scaled by adding more independent resources provisioned on-demand in the cloud.

5.3.2 Design and implementation.

5.3.2.1 Design considerations.

Write scalability and availability. The system needs to support many concurrent writes from a large sensor network, which continuously captures data and sends it to the cloud storage. The system should be always available for writing. For achieving these goals, we propose to use a multi-layer architecture. A high performance load balancer is used as the first layer to accept and forward all write requests from sensor controller nodes evenly to a distributed message queue, which works as a write buffer and handles requests asynchronously. A possible alternative to the load balancer a dynamic scheduler such as MATRIX [92]. A separate distributed Data Agent service keeps pulling messages from the queue, preprocess it and then write to the data store.

The capability to present various data types. There can be many different kinds of sensors in a sensor network, each of them can read different number and types of values. There is no fixed scheme for data formats from the different types of sensors. Therefore we need a flexible data schema so to enable a unified API to collect and store the data, and to organize data in a scalable way for further use query and analytics. To address this issue, we design a flexible message data structure that easily fits into a large category of scalable distributed databases, called column-oriented databases (or BigTable-like data stores). This design enables us to elevate the rich features, performance advantage and scalability from column-oriented databases, as well as to define a unified data access API.

5.3.2.2 Architecture. We design a loosely coupled multi-layer architecture to boost the scalability while maintaining good performance. As shown in fig 5.10, the system is composed of a sensor controller node and a data server that is both written to by the sensors and read from by the clients. On the server side, there are 5 layers of components, namely load balancer, message queue, Data Agent, database, and query execution engine. Each layer can be deployed on a dedicated or shared virtual cluster. If any layer becomes bottleneck, it can be scaled easily by simply adding more resource.

Sensor Controller Node. The sensor controller node accepts data captured from sensors and wraps into a basic message. The client can also send multiple messages in batch through a single transfer as needed. When the clients need to send a big file such as an image by a full-spectrum camera, it will first send the blob through our API to the cloud blob store(such as Amazon S3) or our own file-based blob storage system, and then send a reference message to the database. This reference message is organized in the same way as the basic message, which contains all the metadata, but in the data field, it holds a URL link or a pointer to that file.

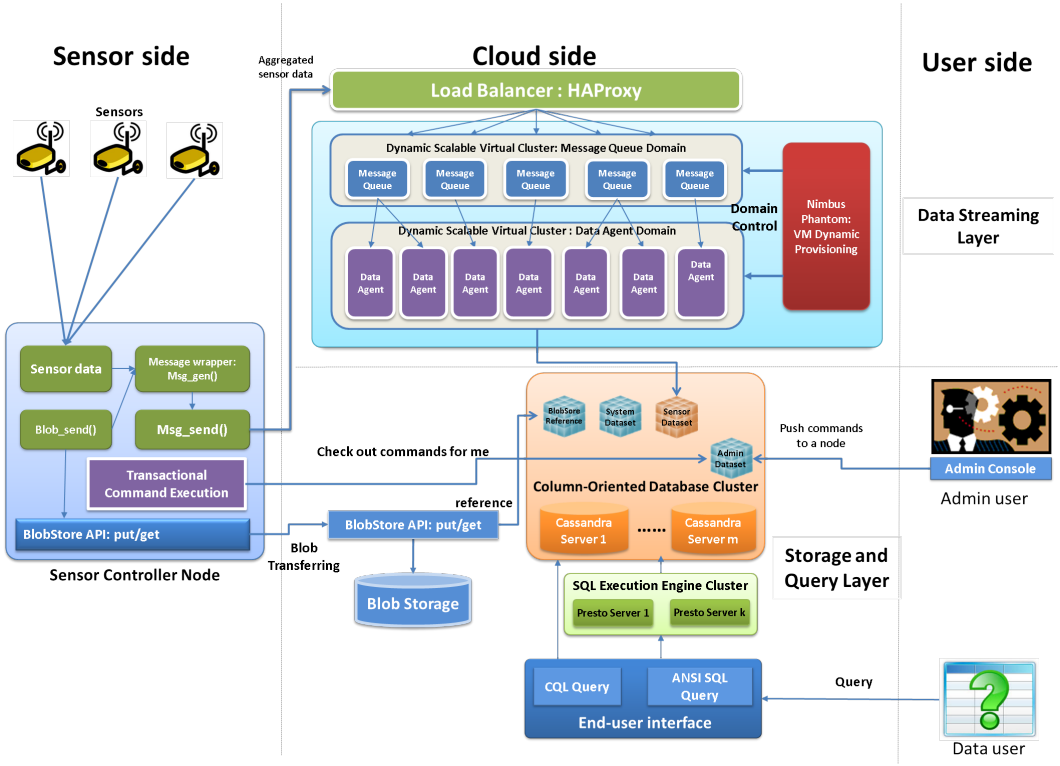


Figure 5.10. System architecture. Sensor controller nodes send messages and blobs to the cloud storage through APIs. Load balancer forwards client requests to data streaming layer. Nimbus Phantom controls dynamic scaling of queue servers and data agent servers.

All these communication are enhanced by our transactional transferring mechanism.

Designing dynamic scalable services. On clouds, most resources are virtually unlimited [77]. All component layers in our system are organized as scalable services, most of which can scale in a dynamic and automatic manner. To allow them to do so, a couple of requirements have to be satisfied [144]. First, the parallel instances of a service can run on multiple machines independently. This ensures a service can be scaled out. Second, a new instance of the service should be able to discover and join a running service. This ensures live scaling of a service, which means the service can scale any time on demand without halting.

Load balancer service. A load balancer is used on top of the whole architecture. The load balancer does not only balance the workload, but also offers a single access point to the clients so as to hide the potential change (such as scaling or failure) in the message queue layer. Load balancer is setup on a big virtual machine.

Message queue service. The message queue service is asynchronously replicated across multiple VMs, which ideally are located close to each other. We choose Availability and Partition tolerance from the CAP theorem [145] and assume that the connection between the VMs is reliable, which is reasonable within a single cloud provider. In this way, any single failed queue won't cause any data loss. A new message queue server can join a cluster easily. We used a simple script to setup and start new message queue service, join the cluster and update the load balancer config file, and then run a reload on load balancer server to finish the system scaling.

Data Agent service. The only job that a Data Agent does is pull messages from message queues, preprocess them, and then push to the storage service. There is no communication and dependency between any two Data Agents. For adding new Data Agents, users only need to tell the new agent where to pull messages and where

to push to, which can easily been set as start parameters. Thus both requirements are satisfied.

Storage service. To meet the various needs of sensor network applications, we design a hybrid storage service that combines a column-oriented database[106] and a blob store. Most of sensor readings are small, and can be put into the database while big files are send to the blob store. For each blob in the blob store, there is a tuple in the database, through which the blob is presented as a regular sensor reading. Thus users can access the data via a unified API.

Design discussion. In This work we use dedicated message queue service (RabbitMQ) and data storage (Cassandra) in order to provide best response time(or latency). One possible alternative solution is to use cloud services such as Amazon SQS and DynamoDB respectively. Uses of cloud service can simplify the system implementation and deployment. However this convenience is at no cost. As we observed before, the response time of both SQS and DynamoDB are multiple times slower than most of user deployed software services. Economic cost is also a big concern.

5.3.2.3 Transactional command execution. In sensor networks, administrators often need to carry out diagnosis and system maintenance by running a series of commands remotely. Conventional remote login such as SSH or Telnet won't work as desired because of the unreliable communication channels. The command execution subsystem must be able to recover from most of the interruptions and communication failures. For solving this problem, we designed and implemented a transactional protocol and stateful data middleware to track the command execution sessions and to persist the results. When a user needs to execute a series of command, s/he firstly sends an execution request to the middleware, which assigns an incremental session ID to the request and then forwards the request to a dedicated database table on cloud.

The commands will not be executed immediately. Instead, controllers check out the available commands from database periodically and then execute them sequentially. We use pull method on controllers instead of opposite, because the cloud side doesn't know if and when a usable network connection is available, so it must be the controller that starts a communication and get the commands. If a controller finds more than one sessions in the database, it will firstly execute the session that with a smallest ID. As the commands are executed on a controller, the results are given a sequence number and push to the database. The user can query the database any time to see if there is any result available. Since both commands and results are persisted in the database, the data lose caused by connection failure is minimized.

5.3.2.4 Implementation. We have implemented the sensor controller client, user query client, administration client, data agent servers and all the adaptors between components in Python [18]. We choose RabbitMQ as message queue server, and Cassandra [12] as the column-oriented database for storage backend. For dynamic scaling on various tiers of the system, we adopted Nimbus Phantom [144], an automatic cloud resource manager and monitoring platform, which enable us manage each tier independently. This work has been integrated into its ongoing parent project, Waggle, at Argonne National Laboratory. The whole system is currently running with real sensors and collecting environmental data.

5.3.3 Experimental Results. As we writing this section, the Waggle sensor network is still in development and doesn't have many sensor controllers. So we used synthetic benchmarks to evaluate our system's performance and functionality on a public science cloud, FutureGrid. This method actually simulated the worst case of the real-world scenario: all sensor controllers happen to send data at the same time, while the normal case is that they send data in a random manner. We claimed that WaggleDB can handle highly concurrent write requests and provide high scalability.

To demonstrate this we evaluated the system with up to 8 servers and 128 clients, in terms of latency, bandwidth, client/server-side scalability and dynamic scaling. We used Tsung benchmarking tool to generate 128 clients on up to 16 virtual machines, and ran up to 8 servers on other virtual machines. We sent 10,000 write requests in a tight loop from each client to the load balancer, which then forwarded requests to queue servers concurrently. Each request was a fixed-length string message.

5.3.3.1 Concurrency and client-side scalability. To determine the capability of handling concurrency and client-side scalability of WaggleDB system, we measured average latency, aggregated data bandwidth and throughput with 1 server. The request latency consists of the time spent on data transferring and request processing. To understand the latency composition better, we measured the latencies with different message sizes, from 10 bytes to 10k bytes. Figure 5.11 shows that while the client scale increased by 32 times, the average latency only increased by 2.2 times. This implied great potential of client-side scalability. Note that the experiments were conducted with client-side tight loops that were only bounded by CPU performance and network bandwidth. Since the real-world clients in sensor networks generally only contact servers occasionally, it's safe to claim that the single-server system can handle many more than 32 real clients. It's worth noting that the latency differences between 10 to 10k bytes message sizes were very small (less than 20%). Within measured message size range, latency was not sensitive to message size. This implied that request processing (open/close socket, acknowledgement) takes more time than data transferring within 10k bytes range.

5.3.3.2 Concurrency and server-side scalability. To determine the server-side scalability and measure the overall performance, we fixed the message size to 1000 bytes and performed similar experiment with up to 128 clients and 8 servers. As more clients joined, the latencies of all server scales increased as shown in figure 5.12. The

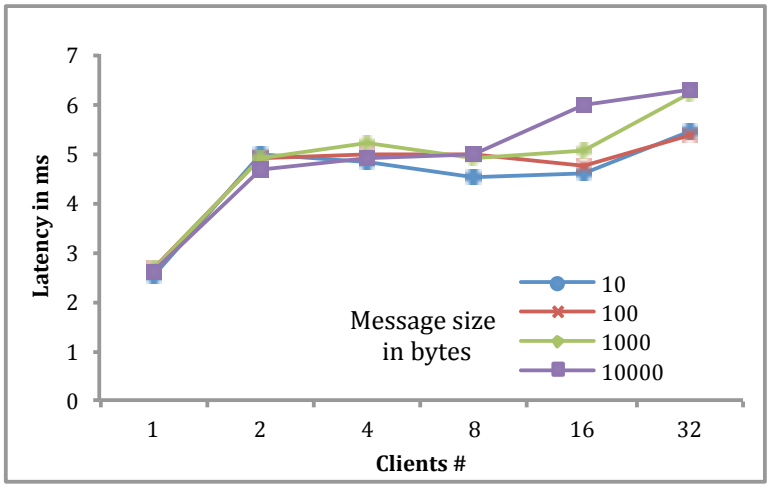


Figure 5.11. Average latency only slowly increased with client number.

more the servers were used, the less the latency increased. On the single-node system, latency started to increase rapidly on 32-client scale and above. This suggested that single server got saturated when serving more than 32 clients.

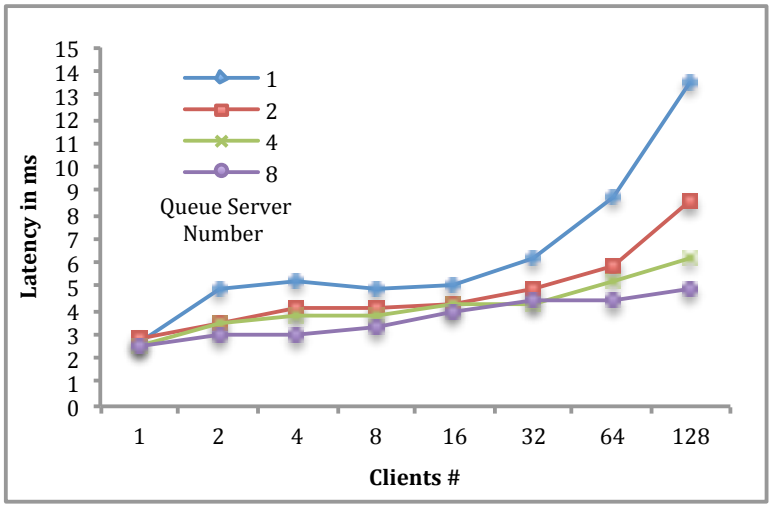


Figure 5.12. Latency comparison. The more the servers were used, the less the latency increased.

5.3.3.3 Dynamic scaling. To verify the functionality and performance impact of dynamic scaling, we measure the latency while scaling queue server and Data Agent server layers on the fly. We firstly tested queue server layer. We fixed the number of clients to 128, started the experiment with single-queue server, ran 30 seconds, doubled the queue servers, and repeated. Eventually we scaled queue server layer to 8 nodes. In figure 5.13, the column chart shows the average latency decreased significantly. Take single-server system as a baseline, scaling to 2, 4 and 8 server brought 36%, 55% and 64% performance gain respectively. The curve chart shows the real-time latency in logarithmic scale. The three high peaks were caused by the load balancer configuration reloading (new list of servers).When adding more than 4 queue servers, the latency decreased slower, because it already close to the ideal value and system was nearly idle.

5.3.4 Summary. In this work we present a set of protocols and a cloud-based data streaming infrastructure called WaggleDB that address those challenges. The system efficiently aggregates and stores data from sensor networks and enables the users to query those data sets. It address the challenges for accommodating sensor network

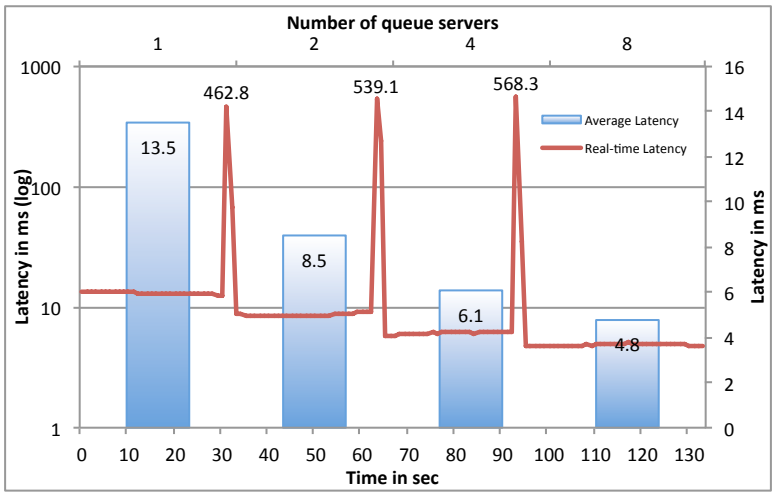


Figure 5.13. Real-time and average latency on dynamic scaling queue servers.

data streams in cloud with a scalable multi-tier architecture, which is designed in such way that each tier can be scaled by adding more independent resources provisioned on-demand in the cloud. The featured high availability and scalability, flexible data scheme and transactional command execution make it a good candidate for sensor network data infrastructure in cloud era.

CHAPTER 6

RELATED WORK

6.1 NoSQL Storage Systmes

In this section we introduce some existing works on NoSQL storage systems related to this dissertation, including distributed hash tables, key-value stores and column-oriented data stores, because they play critical roles in building scalable distributed systems. Numerous distributed NoSQL data stores have been proposed and implemented over the years. Some widely cited and discussed projects and solutions (including Chord [146], CAN [147], Pastry [148], Kademlia [149], Tapestry [150], RIAK [151] and Cassandra [12]) adopt logarithmic routing algorithms, resulted in increased latency with systems scale. Some other works such as Dynamo [152] and Memcached [13] , use constant routing algorithms to achieve a nearly constant latencies like ZHT. Amazon's Dynamo is a key-value storage system that Dynamo hosts some of Amazon's core services. It inspired many similar projects and started a NoSQL trend in both academia and industries. The major focus of Dynamo is to provide an "always-on" experience to it's upper level applications. Dynamo claim to be a zero-hop DHT, where each server has enough routing information locally to route requests to the appropriate target server directly. Memcached is a simple but efficient in-memory a key-value store. It was designed as a cache to speed up distributed data access such as web page caches. Due to it's specific purpose, Memcached doesn't support dynamic membership, replication and persistence. The length of the keys and values are strictly limited to 250 and 1M bytes respectively. Cassandra is firstly inspired by Amazon's Dynamo, strives to be an "always writable" system. In later implementations, it's more considered as a column-family store, like Google's BigTable [106], although it still support key-value interfaces. Cassandra is very popular in industries, but it gets little use in high performance computing areas many

supercomputers don't have good support on Java stack. Another drawback of Cassandra is its logarithmic routing strategy, which makes the performance scalability a big issue.

Some recent key-value store projects generally focus on providing new features or exploring new approaches to boost performance. HyperDex [153] is a distributed key-value store that provides a search primitive that enables queries on secondary attributes. Some adopt new storage backend technology to boost the performance, such as SkimpyStash [154]. SkimpyStash uses a hash table directory in RAM to index key-value pairs stored in a log-structured manner on flash. Due to the relatively simple yet complete basic functionality of key-value stores, there are also many research projects use them as demonstrative prototypes to verify their designs. For example Pileus [100] is a replicated key-value store that allows applications to declare their consistency and latency priorities via consistency-based service level agreements (SLAs). MICA [105] is another scalable key value store that can handle millions of operations per second using single general multi purpose core system. MICA achieved this by encompassing all aspects of request handling by enabling parallel access to data, network request handling, and data structure design. SPANStore [155] presents a key value store that exports a unified view of storage services in geographically distributed data centers. SPANStore combines three main principles, span multiple cloud providers to minimize cost, estimating application workload at the right granularity and finally minimizing use of compute resources. SPANStore in some scenarios was able to lower cost by 10X. Masstree [156] presents anther key value store designed for SMP machines. Masstree functions by keeping all data in memory in a form of concatenated B+ trees. Lookups use optimistic concurrency control, a read-copy-update like technique but no writing on shared data. With these techniques Masstree is able to execute more than six million simple queries per second. For enabling NoSQL databases to handle highly concurrent distributed transactions, Claret [157]

is recently proposed. It utilizes abstract data type (ADT) semantics in databases to provide a safe and scalable programming model for NoSQL databases. LOCS [158] is system equipped with customized SSD design, which exposes its internal flash channels to applications, to work with LSM-tree based KV store, specifically LevelDB in LOCS. Main motivation of LOCS was to overcome inefficiencies to naively combining LSM-tree based KV stores with SSD. They were able to show 4X increase in storage throughput after applying the proposed optimization techniques. Small Index Large Table (SILT) [159] presents a memory efficient high performance key value store system based on flash storage that can scale to serve billions of key value items on a single node. SILT focuses on using algorithmic and systemic techniques to balance the use of memory, storage and computation.

6.2 Boosting performance of distributed storage systems

To achieve high throughput, low latency and better scalability in distributed storage systems on clouds, numerous works have been done in many aspects. Some focus on optimizing network communication. Kielmann's work [160] adopts dynamic load balanced multicast to offer more efficient communication for data-intensive application. Sata developed a model-based algorithm [161] for optimizing I/O intensive applications in clouds through VM layer coordination. Wolf's work [162] attempts to optimize massively parallel I/O and data locality. Some are trying to exploit new hardware, such as NV-RAM. Panda's team, in another hand, proposed new storage primitive [163] for emerging storage hardware. For large-scale storage-class memory systems, Jung [164] attempts to utilize Resistive Random Access Memory (RRAM), another promising memory technology to offer higher bandwidth and lower latency. There are also works done on parallel SQL databases, such as ParaLite [165], which supports collective queries to parallelize User-Defined Executables (EDU).

6.3 Request batching and QoS in key-value stores

For performance improvement, request batching are already used in some production systems, such as Facebook Memcached [13] and Amazon DynamoDB [14]. In these systems, users explicitly wrap requests into batches. Memcached allows users to call a `multiget` API to submit a batch of `get` requests. Note that the batch contains requests that will go to multiple servers. Then the server that initially received the multiget request will have to communicate to many other servers, which may increase the actual latency. Adding more servers will not help this case because the busy server is CPU bounded. This problem is now known as Multiget Hole [166]. In ZHT/Q, batching works in the background and users do not know any details. Multiget Hole problem is avoided by making the requests in a batches have a same destination server. DynamoDB has a similar mechanism for request batching. Another issue with these solutions is that an user must have all the requests ready by hand and then pack them in batches. This requires users to use a very different set of APIs, thus some times change the applications' logic.

Request batching is also used to reduce power consumption. In [167] Cheng used a request batcher on server side to buffer requests so to keep the CPU in idle mode for longer time to save energy. In [168] Wang proposed a batching technique with DVFS for virtual machines to save power. But neither focuses on performance perspective and multiple QoS requirements. Similar ideas are also used in collective I/O [169–171] for storage performance optimization.

There are couple of key-value store projects support QoS or SLA (service level agreement). Pileus [100] is a key-value store that allows applications to declare their consistency and latency requirements. The performance difference is implemented via choosing different consistency level and replication options. Zoolander [172] is a key value store that supports latency SLAs. Similar with Pileus, Zoolander makes use of systems data and workload conditions along with different replication options

to deliver different performance level.

6.4 Provenance

Traditional data provenance represents the change history of data objects. Previous works on data provenance [173] have addressed different aspects, from operating system [107] to file systems [108] to from databases [109]. In our previous work [9], we have shown that distributed key-value stores can boost performance. Karma [174] provenance framework gives a set of tools for collecting provenance from workflow and process. Milieu [175] focuses on provenance collection for scientific experiments in HPC systems.

6.5 System Monitoring

Monitoring gives users a perspective that combines resource utilization, cost efficiency and performance. Previous work has focused on runtime model and attempt to reach the balance between runtime overhead and monitoring capability [113]. Earlier works include Ganglia [114], a distributed monitoring system for clusters and grid systems. FRIEDA-State is event-driven i.e., it does not proactively go to fetch information, and hence is more efficient.

6.6 Unsynchronized Time Clocks and Event Ordering

In large scale distributed systems, unsynchronized clocks and drifting issue are inevitable. Based on different time baselines, its hard to build meaningful semantics from distributed events or logs without synchronization or logical clock mechanisms. Synchronization to a standard time source (atomic clock or GPS clock) is simpler. Typical cases are Precision Time Protocol [176] and NTP [177]. In recent projects, Google Spanner [99] adopts similar way to offer a synchronized clock to global scale databases and offers 5ms accuracy in global scales. Many works have been done for distributed event ordering. Beside Lamports timestamp [178], Vector Clock [123] is

another popular approach in todays systems.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

Today NoSQL databases are increasingly used as building blocks of large scale applications on cloud. But in high performance computing areas, they just start to catch attention. This dissertation firstly addresses several challenges on storage systems at extreme scales for supercomputers and clouds by designing, implementing and applying a zero-hop distributed NoSQL storage system (ZHT). ZHT has been tuned for HPC systems first, then more optimizations are applied to make it a good candidate building block for cloud based applications. This work brings the convergence of NoSQL databases on clouds and supercomputers, it also explores many opportunities in large system design and implementation. At the time of writing this dissertation, ZHT has been adopted in some real systems for supercomputers and clouds.

Then we build a new NoSQL storage system based on ZHT and optimize it to satisfy QoS on latency while achieving high throughput. It supports different QoS latency on a single deployment for multiple concurrent applications, which is a real challenge faced by may big companies and institutions that run NoSQL storage on clouds.

Beside ZHT, we have also explored other NoSQL families, especially column-oriented databases, which provide richer features such as complex query and flexible schemes. Having collaborated with Argonne National Laboratory and Lawrence Berkeley National Laboratory, we have designed and implemented two cloud based systems for their scientific applications based on column-oriented databases.

Based on these experiences we believe that NoSQL storage could transform the architecture of future storage systems in both HPC and clouds, and open the

door to a broader class of applications that would have not normally been tractable. Furthermore, the concepts, data-structures, algorithms, and implementations that underpin these ideas at the largest scales, can be applied to emerging paradigms, such as Cloud Computing, Many-Task Computing, and High-Performance Computing.

7.2 Future Work

Based on ZHT, we have many ideas for future work. There are also many possible use cases where ZHT could make a significant contribution in performance or scalability. Among them, we plan to extend our current work into the following projects.

Key-value store based graph processing system. The emerging applications in big data and social networks issue rapidly increasing demands on graph processing. Graph query operations that involve a large number of vertices and edges can be tremendously slow on traditional databases. We are working on designing and implementing of a new Bulk Synchronous Parallel (BSP) model based graph processing system based on ZHT. The new system is named Graph/Z [82], which can be considered as another Pregel-liked graph processing system, but it inherits some important features from ZHT, a distributed key-value store, which differentiate Graph/Z from other systems. ZHT is a zero-hop distributed key-value store featured with high scalability, persistency and fault tolerance. By leveraging ZHT's persistency, Graph/Z can run with a much larger working dataset.

Integrate ZHT with Swift parallel scripting language. Swift is a system for the rapid and reliable specification, execution, and management of large-scale science and engineering workflows. It supports applications that execute many tasks coupled by disk-resident datasets - as is common, for example, when analyzing large quantities of data or performing parameter studies or ensemble simulations. We re

closely working with Argonne National Lab on MTC (Many Task Computing) Swift so as to boost its performance and scalability through ZHT.

BIBLIOGRAPHY

[1] K. Batcher, "Supercomputer io quote." `http://en.wikipedia.org/wiki/Ken_Batcher`. Accessed: 2014-12-30.

[2] N. Liu, C. Carothers, J. Cope, P. Carns, and R. Ross, "Model and simulation of exascale communication networks," *Journal of Simulation*, vol. 6, pp. 227–236, Nov. 2012.

[3] N. Liu and C. D. Carothers, "Modeling Billion-Node Torus Networks Using Massively Parallel Discrete-Event Simulation," in *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, PADS '11, (Washington, DC, USA), pp. 1–8, IEEE Computer Society, 2011.

[4] N. Liu, A. Haider, X.-H. Sun, and D. Jin, "Fattreesim: Modeling large-scale fat-tree networks for hpc systems and data centers using parallel and discrete event simulation," in *Proceedings of the 3rd ACM SIGSIM/PADS Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '15, (New York, NY, USA), ACM, 2015.

[5] J. Cope, N. Liu, S. Lang, P. Carns, C. D. Carothers, and R. Ross, "CODES: Enabling co-design of multilayer exascale storage architectures," in *Proceedings of the Workshop on Emerging Supercomputing Technologies (WEST)*, (USA), June 2011.

[6] R. H. Schmuck, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST'02, (Monterey, CA), pp. 16–16, 2002.

[7] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for linux clusters," in *In Proceedings of Annual Linux Showcase and Conference*, (Atlanta, Georgia), pp. 317–327, 2000.

[8] P. Schwan, "Lustre: Building a file system for 1000-node clusters," Proc. of the 2003 Linux Symposium, (Berkeley, CA, USA), pp. 174–186, 2003.

[9] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," IPDPS '13.

[10] T. Li, R. Verma, X. Duan, H. Jin, and I. Raicu, "Exploring distributed hash tables in highend computing," *SIGMETRICS Performance Evaluation Review*, 2011.

[11] K. Grolinger, W. A. Higashino, A. Tiwari, and M. A. M. Capretz, "data management in cloud environments: NoSQL and NewSQL data stores," *Journal of Cloud Computing, Advances, Systems and Applications, Springer*, vol. 2, p. 22, 2013.

[12] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, Apr. 2010.

[13] R. Nishtala, H. Fugal, S. Grimm, and M. Kwiatkowski, "Scaling memcache at facebook," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI, (Lombard, IL), pp. 385–398, 2013.

[14] "DynamoDB." `http://aws.amazon.com/dynamodb/`. Accessed: 2014-12-15.

[15] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu, "Fusionfs: Towards supporting data-intensive scientific applications on extreme-scale high-performance computing systems," in *Big Data, 2014 IEEE International Conference on*.

[16] D. Zhao, K. Burlingame, C. Debains, P. Alvarez-Tabio, and I. Raicu, "Towards high-performance and cost-effective distributed storage systems with information dispersal algorithms," IEEE Cluster'13, 2013.

[17] T. Li, I. Raicu, and L. Ramakrishnan, "Scalable state management for scientific applications in the cloud," BigData Congress '14.

[18] T. Li, K. Keahey, R. Sankaran, P. Beckman, and I. Raicu, "A cloud-based interactive data infrastructure for sensor networks," IEEE/ACM Supercomputing/SC'14.

[19] I. Sadooghi, S. Palur, A. Anthony, I. Kapur, K. Belagodu, P. Purandare, K. Ramamurty, K. Wang, and I. Raicu, "Achieving efficient distributed scheduling with message queues in the cloud for many-task computing and high-performance computing," CCGrid'14, 2014.

[20] K. Wang, A. Kulkarni, M. Lang, D. Arnold, and I. Raicu, "Using simulation to explore distributed key-value stores for extreme-scale system services," SC '13, 2013.

[21] K. Wang, X. Zhou, H. Chen, M. Lang, and I. Raicu, "Next generation job management systems for extreme-scale ensemble computing," HPDC '14, 2014.

[22] D. Zhao, C. Shou, T. Malik, and I. Raicu, "Distributed data provenance for large-scale data-intensive computing," CLUSTER, 2013.

[23] C. Shou, D. Zhao, T. Malik, and I. Raicu, "Towards a provenance-aware a distributed file system," TaPP13, 2013.

[24] K. Wang, A. Rajendran, K. Brandstatter, Z. Zhang, and I. Raicu, "Paving the road to exascale with many-task computing," SC'12 Poster, 2012.

[25] I. Raicu, I. T. Foster, and P. Beckman, "Making a case for distributed file systems at exascale," LSAP '11, 2011.

[26] D. Zhao, D. Zhang, K. Wang, and I. Raicu, "Exploring reliability of exascale systems through simulations," HPC '13, pp. 1:1–1:9, 2013.

[27] K. Wang, N. Liu, I. Sadooghi, X. Yang, X. Zhou, T. Li, M. Lang, X.-H. Sun, and I. Raicu, "Overcoming Hadoop scaling limitations through distributed task execution," IEEE Cluster'15, 2015.

[28] K. Wang, K. Brandstatter, and I. Raicu, "Simmatrix: Simulator for manytask computing execution fabric at exascales," HPC, 2013.

[29] D. Patel, F. Khasib, I. Sadooghi, and I. Raicu, "Towards In-Order and Exactly-Once Delivery using Hierarchical Distributed Message Queues," SCRAMBL'14, 2014.

[30] D. Zhao, K. Qiao, and I. Raicu, "Hycache+: Towards scalable high-performance caching middleware for parallel file systems," 2014.

[31] D. Zhao, J. Yin, K. Qiao, and I. Raicu, "Virtual chunks: On supporting random accesses to scientific data in compressible storage systems," IEEE BigData'14, 2014.

[32] A. Rajendran and I. Raicu, "Matrix: Many-task computing execution fabric for extreme scales," 2013.

[33] Y. Wang, Y. Su, and G. Agrawal, "Supporting a Light-Weight Data Management Layer Over HDF5," in *CCGrid'13*, IEEE.

[34] L. Yu, Z. Zhou, S. Wallace, M. E. Papka, and Z. Lan, "Quantitative modeling of power performance tradeoffs on extreme scale systems," *Journal of Parallel and Distributed Computing*, vol. 84, pp. 1–14, 2015.

[35] Y. Wang, G. Agrawal, G. Ozer, and K. Huang, "Removing sequential bottlenecks in analysis of next-generation sequencing data," in *IPDPSW, 2014 IEEE International*, IEEE, 2014.

[36] S. Zhang, *Quantitative Risk Assessment under Multi-Context Environments*. PhD thesis, Kansas State University, 2014.

[37] S. Zhang, X. Zhang, and X. Ou, "After we knew it: empirical study and modeling of cost-effectiveness of exploiting prevalent known vulnerabilities across iaas cloud," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pp. 317–328, ACM, 2014.

[38] S. Zhang, X. Ou, and J. Homer, "Effective network vulnerability assessment through model abstraction," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 17–34, Springer, 2011.

[39] S. Zhang, X. Ou, A. Singhal, and J. Homer, "An empirical study of a vulnerability metric aggregation method," in *The 2011 International Conference on Security and Management (SAM'11), special track on Mission Assurance and Critical Infrastructure Protection (STMACIP'11)*, 2011.

[40] J. Homer, S. Zhang, X. Ou, D. Schmidt, Y. Du, S. R. Rajagopalan, and A. Singhal, "Aggregating vulnerability metrics in enterprise networks using attack graphs," *Journal of Computer Security*, vol. 21, no. 4, pp. 561–597, 2013.

[41] R. Zhuang, S. Zhang, S. A. DeLoach, X. Ou, and A. Singhal, "Simulation-based approaches to studying effectiveness of moving-target network defense," in *National Symposium on Moving Target Research*, 2012.

[42] R. Zhuang, S. Zhang, A. Bardas, S. A. DeLoach, X. Ou, and A. Singhal, "Investigating the application of moving target defenses to network security," in *Resilient Control Systems (ISRCS), 2013 6th International Symposium on*, pp. 162–169, IEEE, 2013.

[43] S. Zhang, D. Caragea, and X. Ou, "An empirical study on using the national vulnerability database to predict software vulnerabilities," in *Database and Expert Systems Applications*, pp. 217–231, Springer, 2011.

[44] H. Huang, S. Zhang, X. Ou, A. Prakash, and K. Sakallah, "Distilling critical attack graph surface iteratively through minimum-cost sat solving," in *Proceedings of the 27th Annual Computer Security Applications Conference*, pp. 31–40, ACM, 2011.

[45] S. A. DeLoach, X. Ou, R. Zhuang, and S. Zhang, "Model-driven, moving-target defense for enterprise network security," in *Models@ run. time*, pp. 137–161, Springer International Publishing, 2014.

[46] S. Zhang, "Deep-diving into an easily-overlooked threat: Inter-vm attacks," *Whitepaper, provided by Kansas State University, TechRepublic/US2012, available online: http://www. techrepublic. com/resourcelibrary/whitepapers/deep-diving-into-an-easilyoverlooked-threat-inter-vm-attacks*, 2013.

[47] X. ZHANG, Q. LI, F. SHI, and C. SHEN, "Micro injection molding simulation based on sph method," *CIESC Journal*, 2012.

[48] F. Shi, X. Zhang, Q. Li, and C. Shen, "Notice of retraction particle tracking in micro-injection molding simulated by mis," in *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, IEEE, 2010.

[49] F. Shi, X. Zhang, Q. Li, and C. Shen, "Mould wall friction effects on micro injection moulding based on simulation of mis," *IOP Conference Series: Materials Science and Engineering*, 2010.

[50] S. Fan, Z. Xiang, L. Qian, and S. Changyu, "Numerical simulation of micro injection moulding based on mesh free method," *Sciencepaper Online*, 2010.

[51] X. Li, Z. Lv, W. Wang, C. Wu, and J. Hu, "Virtual reality gis and cloud service based traffic analysis platform," in *The 23rd International Conference on Geoinformatics (Geoinformatics2015)*, IEEE, 2015.

[52] X. Li, Z. Lv, J. Hu, B. Zhang, L. Shi, and S. Feng, "Xearth: A 3d gis platform for managing massive city information," in *Computational Intelligence and Virtual Environments for Measurement Systems and Applications (CIVEMSA), 2015 IEEE International Conference on*, pp. 1–6, IEEE, 2015.

[53] Z.-H. Lv, R.-N. Ma, J.-B. Fang, Y. Han, and G. Chen, "Index structure for multi-scale representation of multi-dimensional spatial data in webgis [j]," *Application Research of Computers*, vol. 9, p. 053, 2010.

[54] W. Wang, Z. Lv, X. Li, W. Xu, B. Zhang, and X. Zhang, "Virtual reality based gis analysis platform," in *22nd International Conference on Neural Information Processing (ICONIP2015)*, Springer, 2015.

[55] X. Li, Z. Lv, J. Hu, L. Yin, B. Zhang, and S. Feng, "Virtual reality gis based traffic analysis and visualization system," *Advances in Engineering Software*, 2015.

[56] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," MSST'12, 2012.

[57] N. Liu, J. Fu, C. Carothers, O. Sahni, K. Jansen, and M. Shephard, "Massively parallel I/O for partitioned solver systems," *Parallel Processing Letters*, vol. 20, no. 4, pp. 377–395, 2010.

[58] T. Li, A. P. D. Tejada, K. Brandstatter, Z. Zhang, and I. Raicu, "ZHT: a zero-hop DHT for high-end computing environment," GCASR, 2012.

[59] T. Li, X. Zhou, K. Brandstatter, and I. Raicu, "Distributed key-value store on HPC and cloud systems," GCASR' 13, 2013.

[60] L. Yu and Z. Lan, "A Scalable, Non-Parametric Method for Detecting Performance Anomaly in Large Scale Computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, pp. 868–877, 2015.

[61] L. Yu and Z. Lan, "A scalable, non-parametric anomaly detection framework for hadoop," in *ACM Cloud and Autonomic Computing Conference (CAC)*, p. 22, 2013.

[62] X. Yang, Z. Zhou, S. Wallace, Z. Lan, W. Tang, S. Coghlan, and M. E. Papka, "Integrating dynamic pricing of electricity into energy aware scheduling for hpc systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), pp. 60:1–60:11, ACM, 2013.

[63] X. Yang, Z. Zhou, W. Tang, X. Zheng, J. Wang, and Z. Lan, "Balancing job performance with system performance via locality-aware scheduling on torus-connected systems," in *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pp. 140–148, Sept 2014.

[64] K. Grolinger, W. Higashino, A. Tiwari, and M. Capretz, "Data management in cloud environments: NoSQL and NewSQL data stores," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 2, no. 1, 2013.

[65] "epoll - linux man page." `http://linux.die.net/man/4/epoll`. Accessed: 2015-1-30.

[66] X. Yang, Z. Zhou, W. Tang, X. Zheng, J. Wang, and Z. Lan, "Balancing job performance with system performance via locality-aware scheduling on torus-connected systems," in *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, (Madrid, Spain), pp. 140–148, IEEE, 2014.

[67] Z. Zhou, X. Yang, Z. Lan, P. Rich, W. Tang, V. Morozov, and N. Desai, "Improving batch scheduling on Blue Gene/Q by relaxing 5d torus network allocation constraints," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, IPDPS'15, (Hyderabad, India), 2015.

[68] K. Brandstatter, T. Li, X. Zhou, and I. Raicu, "NoVoHT: a lightweight dynamic persistent NoSQL key/value store," GCASR' 13, 2013.

[69] "Google protocol buffers." `http://code.google.com/apis/protocolbuffers`. Accessed: 2014-09-30.

[70] "Amazon ec2 cloud." `http://aws.amazon.com/ec2/`. Accessed: 2014-11-30.

[71] "Parallel reconfigurable observational environment." `http://www.nmc-probe.org/`. Accessed: 2014-11-30.

[72] "Ibm BlueGene supercomputers." `http://en.wikipedia.org/wiki/Blue_Gene`. Accessed: 2012-1-30.

[73] "Argonne leadership computing facility." `https://www.alcf.anl.gov/`. Accessed: 2014-11-30.

[74] "Kyotocabinet." `http://fallabs.com/kyotocabinet/`. Accessed: 2012-09-30.

[75] N. Liu and C. Carothers, "Modeling billion-node torus networks using massively parallel discrete-event simulation," in *Proceedings of Workshop on Principles of Advanced and Distributed Simulation* , (Washington, DC, USA), pp. 1–8, 2011.

[76] N. Liu, A. Haider, X. Sun, and D. Jin, "Fattreesim: Modeling a large-scale fat-tree network for hpc systems and data centers using parallel and discrete event simulation," in *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation (PADS), 2011 ACM SIGSIM*, (London, UK), pp. 199–210, ACM, IEEE, June 2015.

[77] I. Sadooghi, J. Hernandez Martin, T. Li, K. Brandstatter, Y. Zhao, K. Maheshwari, T. Pais Pitta de Lacerda Ruivo, S. Timm, G. Garzoglio, and I. Raicu, "Understanding the performance and potential of cloud computing for scientific applications," 2015.

[78] I. Sadooghi, D. Zhao, T. Li, and I. Raicu, "Understanding the cost of cloud computing and storage," GCASR' 12.

[79] K. Feng, T. Che, T. Li, and I. Raicu, "Oht: Hierarchical distributed hash tables," *IIT Technical Report*, 2013.

[80] K. Wang, A. Kulkarni, M. Lang, D. Arnold, and I. Raicu, "Exploring the design tradeoffs for extreme-scale high-performance computing system software," in *IEEE Transaction on Parallel and Distributed Systems (TPDS)*, 2015.

[81] T. Li, K. Keahey, K. Wang, D. Zhao, and I. Raicu, "A dynamically scalable cloud data infrastructure for sensor networks," ACM ScienceCloud 15.

[82] T. Li, C. Ma, J. Li, X. Zhou, K. Wang, D. Zhao, I. Sadooghi, and I. Raicu, "GRAPH/Z: A key-value store based scalable graph processing system," Cluster'15.

[83] D. Zhao, N. Liu, D. Kimpe, R. Ross, X.-H. Sun, and I. Raicu, "Towards exploring data-intensive scientific applications at extreme scales through systems and simulations," TPDS, pp. 1–1, 2015.

[84] I. Sadooghi, D. Zhao, T. Li, and I. Raicu, "Understanding the cost of cloud computing and storage," GCASR, 2012.

[85] C. S. Zhao, Z. Zhang, I. Sadooghi, X. Zhou, T. Li, and I. Raicu, "FusionFS: a distributed file system for large scale data-intensive computing," GCASR, 2013.

[86] D. Zhao, C. Shou, Z. Zhang, I. Sadooghi, X. Zhou, T. Li, and I. Raicu, "FusionFS: a distributed file system for large scale data-intensive computing," GCASR' 13.

[87] W.Vogels, "Filesystem in userspace." `http://fuse.sourceforge.net/`. Accessed: 2011-09-17.

[88] A. J. McAuley, "Reliable broadband communication using a burst erasure correcting code," in *Proceedings of the 1990 ACM SIGCOMM International Conference on on Data Communication*, SIGCOMM '90, (Philadelphia, Pennsylvania, USA), pp. 297–306, 1990.

[89] A. G. Dimakis, V. Prabhakaran, and K. Ramchandran, "Decentralized erasure codes for distributed networked storage," *IEEE/ACM Trans. Netw.*, vol. 14, pp. 2809–2816, June 2006.

[90] M. Li, J. Shu, and W. Zheng, "GRID Codes: Strip-based erasure codes with high fault tolerance for storage systems," *ACM Transactions on Storage*, vol. 4, pp. 15:1–15:22, Feb. 2009.

[91] K. Wang, X. Zhou, K. Qiao, M. Lang, B. McClelland, and I. Raicu, "Towards scalable distributed workload manager with monitoring-based weakly consistent resource stealing," ACM HPDC '15, 2015.

[92] K. Wang, X. Zhou, T. Li, M. Lang, and I. Raicu, "Optimizing load balancing and data-locality with data-aware scheduling," IEEE BigData'14.

[93] Z. Zhou, X. Yang, D. Zhao, P. Rich, W. Tang, J. Wang, and Z. Lan, "I/o-aware batch scheduling for petascale computing systems," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pp. 254–263, Sept 2015.

[94] Z. Zhou, X. Yang, Z. Lan, P. Rich, W. Tang, V. Morozov, and N. Desai, "Improving batch scheduling on blue gene/q by relaxing 5d torus network allocation constraints," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pp. 439–448, May 2015.

[95] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: A fast and light-weight task execution framework," SC '07, (Reno, Nevada), pp. 1–12, 2007.

[96] M. A. Jette, A. B. Yoo, and M. Grondona, "Slurm: Simple linux utility for resource management," in *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP)*, (Seattle, WA, USA), pp. 44–60, 2003.

[97] I. Sadooghi, K. Wang, S. Srivastava, D. Patel, D. Zhao, T. Li, and I. Raicu, "Fabriq: Leveraging distributed hash tables towards distributed publish-subscribe message queues," IEEE/ACM International Symposium on Big Data Computing (BDC), 2015.

[98] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," SIGMETRICS '12, 2012.

[99] J. C. Corbett, J. Dean, and M. Epstein, "Spanner: Google's globally-distributed database," OSDI, 2012.

[100] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," SOSP '13.

[101] P. Xiong, H. Hacigumus, and J. F. Naughton, "A software-defined networking based approach for performance management of analytical queries on distributed data stores," SIGMOD '14, 2014.

[102] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Real Time Systems, The International Journal of Time-Critical Computing Systems*, 1990.

[103] M. Berg, F. V. D. D. Schouten, and J. Jansen, "Optimal batch provisioning to customers subject to a delay-limit," *Manage. Sci.*, vol. 44, pp. 684–697, May 1998.

[104] T. Cucinotta, "Access control for adaptive reservations on multi-user systems," in *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, 2008.

[105] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: a holistic approach to fast in-memory key-value storage," NSDI'14.

[106] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, pp. 4:1–4:26, June 2008.

[107] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer, "Provenance for the cloud," in *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST'10, 2010.

[108] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," USENIX Annual Technical Conference, 2006.

[109] P. Buneman and W.-C. Tan, "Provenance in databases," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2007.

[110] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, reliable, loosely coupled parallel computation," IEEE Workshop on Scientific Workflows'07, 2007.

[111] Y. Zhao, I. Raicu, S. Lu, and X. Fei, "Opportunities and challenges in running scientific workflows on the cloud," IEEE CyberC, 2011.

[112] I. Raicu, I. Foster, Y. Zhao, A. Szalay, P. Little, C. M. Moretti, A. Chaudhary, and D. Thain, *Towards Data Intensive Many-Task Computing*. Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management, IGI Global Publishers, 2009.

[113] P. T. Barbon, M. Pistore, and M. Trainotti, "run-time monitoring of instances and classes of web service compositions, in Web Services," ICWS, 2006.

[114] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, July 2004.

[115] J. Shao, H. Wei, Q. Wang, and H. Mei, "A runtime model based monitoring approach for cloud," CLOUD, 2010.

[116] L. Yu, D. Li, S. Mittal, and J. Vetter, "Quantitatively modeling application resilience with the data vulnerability factor," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 695–706, 2014.

[117] E. Berrocal, L. Yu, S. Wallace, M. Papka, and Z. Lan, "Exploring void search for fault detection on extreme scale systems," in *International Conference on Cluster Computing (CLUSTER)*, pp. 1–9, 2014.

[118] Z. Zheng, L. Yu, Z. Lan, and T. Jones, "3 Dimensionall Root Cause Diagnosis via Co-analysis," in *International Conference on Autonomic Computing*, pp. 181–190, 2009.

[119] L. Yu, Z. Zheng, Z. Lan, T. Jones, J. M. Brandt, and A. C. Gentile, "Practical online failure prediction for Blue Gene/P: Period-based vs event-driven," in *International Conference on Dependable Systems and Networks Workshops*, pp. 259–264, 2011.

[120] Z. Zheng, L. Yu, and Z. Lan, "Reliability-Aware Speedup Models for Parallel Applications with Coordinated Checkpointing/Restart," *IEEE Trans. Comput.*, vol. 64, pp. 1402–1415, 2015.

[121] R. Ostrovsky and B. Patt-Shamir, "Optimal and efficient clock synchronization under drifting clocks," PODC, 1999.

[122] D. Ghoshal and L. Ramakrishnan, "Frieda: Flexible robust intelligent elastic data management in cloud environments," SCC, 2012.

[123] F. Mattern, "Virtual time and global states of distributed systems," Workshop on Parallel and Distributed Algorithms, 1988.

[124] Y. Wang, W. Jiang, and G. Agrawal, "Scimate: A novel mapreduce-like framework for multiple scientific data formats," CCGRID '12, 2012.

[125] Y. Wang, G. Agrawal, T. Bicer, and W. Jiang, "Smart: A mapreduce-like framework for in-situ scientific analytics," tech. rep., OSU-CISRC-4/15-TR05, Ohio State University, 2015.

[126] Y. Wang, Y. Su, and G. Agrawal, "A novel approach for approximate aggregations over arrays," in *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, p. 4, ACM, 2015.

[127] Y. Su, Y. Wang, G. Agrawal, and R. Kettimuthu, "SDQuery DSI: Integrating Data Management Support with a Wide Area Data Transfer Protocol," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 47, ACM, 2013.

[128] G. Zhu, Y. Wang, and G. Agrawal, "SciCSM: Novel Contrast Set Mining over Scientific Datasets Using Bitmap Indices," in *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, p. 38, ACM, 2015.

[129] "Futuregrid." `https://portal.futuregrid.org/`. Accessed: 2014-11-30.

[130] A. D'Alessandro and G. D'Anna, *Suitability of low-cost three-axis MEMS accelerometers in strong-motion seismology: tests on the LIS331DLH (iPhone) accelerometer*. Bulletin of the Seismological Society of America, 2013.

[131] Z. Lu, M. S. Lal Khan, and S. Ur Réhman, "Hand and foot gesture interaction for handheld devices," in *Proceedings of the 21st ACM international conference on Multimedia*, pp. 621–624, ACM, 2013.

[132] A. Tek, B. Laurent, M. Piuzzi, Z. Lu, M. Chavent, M. Baaden, O. Delalande, C. Martin, L. Piccinali, B. Katz, *et al.*, "Advances in human-protein interaction-interactive and immersive molecular simulations," *Biochemistry, Genetics and Molecular Biology"Protein-Protein Interactions-Computational and Experimental Tools'*, pp. 27–65, 2012.

[133] Z. Lu and S. Ur Réhman, "Touch-less interaction smartphone on go!," in *SIGGRAPH Asia 2013 Posters*, p. 28, ACM, 2013.

[134] Z. Lu, S. U. Réhman, and G. Chen, "Webvrgis: Webgis based interactive online 3d virtual community," in *Virtual Reality and Visualization (ICVRV), 2013 International Conference on*, pp. 94–99, IEEE, 2013.

[135] Z. Lv, A. Halawani, M. S. Lal Khan, S. U. Réhman, and H. Li, "Finger in air: touch-less interaction on smartphone," in *Proceedings of the 12th International Conference on Mobile and Ubiquitous Multimedia*, p. 16, ACM, 2013.

[136] Z. Lv, G. Chen, C. Zhong, Y. Han, and Y. Y. Qi, "A framework for multi-dimensional webgis based interactive online virtual community," *Advanced Science Letters*, vol. 7, no. 1, pp. 215–219, 2012.

[137] Z. Lv, S. Feng, M. S. L. Khan, S. Ur Réhman, and H. Li, "Foot motion sensing: augmented game interface based on foot interaction for smartphone," in *CHI'14 Extended Abstracts on Human Factors in Computing Systems*, pp. 293–296, ACM, 2014.

[138] M. S. L. Khan, S. U. Réhman, L. Zhihan, and H. Li, "Head orientation modeling: Geometric head pose estimation using monocular camera," in *The 1st IEEE/IIAE International Conference on Intelligent Systems and Image Processing 2013 (ICISIP2013)*, 2013.

[139] Z. Lv, L. Feng, H. Li, and S. Feng, "Hand-free motion interaction on google glass," in *SIGGRAPH Asia 2014 Mobile Graphics and Interactive Applications*, ACM, 2014.

[140] Z. Lv, L. Feng, S. Feng, and H. Li, "Extending touch-less interaction on vision based wearable device," in *IEEE Virtual Reality Conference 2015, 23 - 27 March, Arles, France*, IEEE, 2015.

[141] Z. Lv and H. Li, "Imagining in-air interaction for hemiplegia sufferer," in *International Conference on Virtual Rehabilitation (ICVR2015)*, IEEE, 2015.

[142] Z. Lv, C. Esteve, J. Chirivella, and P. Gagliardo, "Clinical feedback and technology selection of game based dysphonic rehabilitation tool," in *9th International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth2015)*, IEEE, 2015.

[143] Z. Lv, A. Halawani, S. Feng, and H. Li, "Touch-less interactive augmented reality game on vision based wearable device," *Personal and Ubiquitous Computing*, 2015.

[144] K. Keahey, P. Armstrong, J. Bresnahan, D. LaBissoniere, and P. Riteau, "Infrastructure outsourcing in multi-cloud environment," FederatedClouds '12, pp. 33–38, ACM, 2012.

[145] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, pp. 51–59, June 2002.

[146] R. M. Stoica, D. Karger, M. F. Kaashoek, and H. Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. ACM SIGCOMM, 2001.

[147] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *Proceedings of the 2001 ACM SIGCOMM International Conference on on Data Communication*, SIGCOMM, (New York, NY, USA), pp. 161–172, 2001.

[148] Rowstron and P. Druschel, *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*. Middleware, 2001.

[149] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, (London, UK, UK), pp. 53–65, 2001.

[150] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communication*, vol. 22, no. 1, pp. 41–53, 2004.

[151] "Riak." `http://docs.basho.com/riak/latest/`. Accessed: 2014-1-30.

[152] D. H. DeCandia, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "dynamo: Amazons highly available key-value store," in *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP, (New York, NY, USA), pp. 205–220, 2007.

[153] R. Escriva, B. Wong, and E. G. Sirer, "Hyperdex: A distributed, searchable key-value store," in *Proceedings of the 2012 ACM SIGCOMM International Conference on on Data Communication*, SIGCOMM '12, (Helsinki, Finland), pp. 25–36, 2012.

[154] B. Debnath, S. Sengupta, and J. Li, "Skimpystash: Ram space skimpy key-value store on flash-based storage," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, (Athens, Greece), pp. 25–36, 2011.

[155] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services," SOSP '13.

[156] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," EuroSys '12.

[157] B. Holt, I. Zhang, D. Ports, M. Oskin, and L. Ceze, "Claret: Using data types for highly concurrent distributed transactions," in *Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, (Bordeaux, France), pp. 1–4, 2015.

[158] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of lsm-tree based key-value store on open-channel ssd," EuroSys '14.

[159] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," SOSP '11, 2011.

[160] T. Chiba, M. den Burger, T. Kielmann, and S. Matsuoka, "Dynamic load-balanced multicast for data-intensive applications on clouds," CCGrid 2010, pp. 5–14, May 2010.

[161] K. Sato, H. Sato, and S. Matsuoka, "A model-based algorithm for optimizing i/o intensive applications in clouds using vm-based migration," CCGRID '09, pp. 466–471, May 2009.

[162] W. Frings, F. Wolf, and V. Petkov, "Scalable massively parallel i/o to task-local files," in *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pp. 1–11, Nov 2009.

[163] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. Panda, "Beyond block i/o: Rethinking traditional storage primitives," HPCA '11, pp. 301–311, Feb 2011.

[164] M. Jung, J. Shalf, and M. Kandemir, "Design of a large-scale storage-class rram system," ICS '13, (New York, NY, USA), pp. 103–114, ACM, 2013.

[165] T. Chen and K. Taura, "Paralite: Supporting collective queries in database system to parallelize user-defined executable," CCGRID '12, (Washington, DC, USA), pp. 474–481, IEEE Computer Society, 2012.

[166] High Scalability Blog, "Facebook's Memcached multiget hole: More machines != more capacity." `http://highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more-machines-more-capacit.html`. Accessed: 2014-12-15.

[167] D. Cheng, Y. Guo, and X. Zhou, "Self-tuning batching with dvfs for improving performance and energy efficiency in servers," MASCOTS '13, (Washington, DC, USA), pp. 40–49, IEEE Computer Society, 2013.

[168] Y. Wang and X. Wang, "Virtual batching: Request batching for server energy conservation in virtualized data centers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, pp. 1695–1705, Aug. 2013.

[169] Y. Lu, Y. Chen, Y. Zhuang, J. Liu, and R. Thakur, "Collective input/output under memory constraints," 2014.

[170] J. Liu, Y. Chen, and S. Byna, "Collective computing for scientific big data analysis," in *The Eighth International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, 2015.

[171] J. Liu, Y. Zhuang, and Y. Chen, "Hierarchical collective i/o scheduling for high-performance computing," *Big Data Research*, vol. 2, no. 3, pp. 117 – 126, 2015. Big Data, Analytics, and High-Performance Computing.

[172] C. Stewart, A. Chakrabarti, and R. Griffith, "Zoolander: Efficiently meeting very strict, low-latency slos," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, (San Jose, CA), pp. 265–277, USENIX, 2013.

[173] Y. L. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-science," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2005.

[174] Y. L. Simmhan, B. Plale, D. Gannon, and S. Marru, "Performance evaluation of the karma provenance framework for scientific workflows," IPAW, 2006.

[175] Y.-W. Cheah, R. Canon, B. . R. Plale, and L., "Milieu: Lightweight and congurable big data provenance for science, big data (BigData congress)," BigData Congress, 2013.

[176] "Precision time protocol." `http://www.ieee1588.com/`. Accessed: 2014-05-30.

[177] D. L., *Computer Network Time Synchronization: The Network Time Protocol*. Taylor & Francis, 2010.

[178] L. Lamport, "Time, clocks, and the ordering of events in a distributed system.," *Commun*, vol. 21, no. 7, pp. 558–565, 1978.