PaxStore: A Distributed Key Value Storage System

Zhipeng Tan, Yongxing Dang, Jianliang Sun, Wei Zhou, Dan Feng

PaxStore：A Distributed Key Value Storage System

Zhipeng Tan, Yongxing Dang, Jianliang Sun, Wei Zhou, Dan Feng
Wuhan National Laboratory for Optoelectronics, School of Computer Science, Huazhong University of Science and Technology, Wuhan, China

Abstract—Consistency, availability, scalability, and tolerance to the network partition are four important problems in distributed systems. In this paper, we have designed a consistent, highly available distributed key value storage system that can run on lots of general devices and solve the four problems in distributed systems, we call it as PaxStore. It uses zookeeper to complete leader election. It uses a centralized Paxos-based protocol to guarantee the strong replica consistency. The system node can automatically recover in case of failure. Experiments show that PaxStore can guarantee the strong consistency and only increases 20% overhead compared with local systems. By using log optimization, such as the circular lock-free queue and Paxos protocol optimization techniques, PaxStore has a high performance and recovery speed than the older system which uses a basic Paxos protocol.

1. Introduction

With the rapid development of computer technology and Internet, especially the emerging of Web 2.0 technology, information grows explosively. Therefore, it is difficult to improve the system performance by using the scale-up[2] method (provide larger and more powerful servers). The scale-out[2] method, in the form of clusters of general machines, is a long-term solution to solve the bottlenecks of storage systems. However, the problems in distributed systems are far more complex than problems in a single machine. We have to solve various anomalies, such as node failure, disk failure, network partition, message missing etc.. It is difficult to build a highly available distributed storage system under complex conditions.

In distributed systems, consistency, availability and partition tolerance are three important issues. However, no distributed systems can simultaneously achieve the three goals according to Brew's CAP Theorem[3]. Stonebraker[4] argued that strong consistency and availability may be a better design choice in a single datacenter where network partitions are rare.

Replica consistency is an important issue of distributed systems. For some application scenarios such as bank, military, and scientific experiment, any inconsistency in replicas is intolerable. There are some popular replica consistency protocols such as two phase commit protocol[5], and Paxos protocol[6] etc. Unfortunately, in hostile system environments, two-phase commit may not guarantee the strong consistency among multiple replicas and the high system availability. With three or more replicas, the Paxos family of protocols is considered to be the only

solution to guarantee the strong replica consistency. However it is not widely used in distributed systems due to its complexity and low efficiency.

Besides consistency, system availability is also one of the key principles in designing a distributed system. Many internet enterprises, like Google and eBay, often have to provide reliable service of 24×7 hours for their users. However, the node failure happens frequently in distributed systems when running on general servers. Therefore how to continuously provide service after a node is down is the problem that we should solve.

This paper presents a new distributed key-value storage system, called PaxStore, which can guarantee the strong replica consistency by using a centralized Paxos-based protocol. The protocol can significantly reduce the overhead compared with basic Paxos. In PaxStore, if the leader failed, PaxStore can automatically select a new leader to provide service uninterruptedly as long as the majority of its replicas are alive. Furthermore, the system node can automatically recover in case of node failure. Experiments show PaxStore can guarantee strong consistency among replicas and only increases 20% overhead compared with local systems. Furthermore, PaxStore is five times or more as fast as the older which also uses a basic Paxos protocol on write.

The rest of the paper is organized as follows. Section 2 provides a detailed survey of existing work and the related backgrounds. Section 3 presents the design of PaxStore. Section 4 is the implementation of PaxStore. Section 5 gives an experimental evaluation of PaxStore. Section 6 summarizes our work and draws conclusions.

2. Related Work

Brew's CAP theorem[3] is of great significance in the distributed systems, which shows that it is impossible for any distributed system to simultaneously provide all of the three following guarantees: consistency, availability and partition tolerance. Actually, many distributed storage systems choose two of the above goals based on their own application characteristics.

Many relational databases use the two-phase commit protocol, such as MySQL, which has very good C (strong consistency), but it's A (availability) & P (partition tolerance) are poor. For example, these systems can prevent data from being lost when facing with disk failures. But they may not provide service if a node fails or in the abnormal network conditions.

Dynamo[8], and Cassandra[9] provide high availability and partition tolerance by using eventual consistency. In CAP terminology, they are typical AP systems. Dynamo uses the Quorum mechanism to manage replicas, which is a decentralized system. When facing replicas inconsistency, applications must resolve the conflicts by using data update timestamp.

The Paxos algorithm was proposed by Leslie Lamport in 1990[7], which is a consistency algorithm based on message passing. At first, it didn't attract people's attention because it is difficult to understand. However, in recent years, the widespread use of Paxos algorithm proves its important role in distributed systems. The basic idea of the Paxos algorithm is that, the successful execution of each request

needs the acceptance and execution of the vast majority of nodes in the systems; every Paxos instance has a sequence, which executes from small to large and all nodes have the same instance execution order; if a new node joins systems, it can recover data through catch-up mechanism to achieve the same status as the existing nodes. But the basic Paxos protocol is a decentralized protocol which requires multiple network communications, its efficiency is low. PaxStore uses a centralized Paxos-based protocol with small network overhead.

Zookeeper[10] uses basic Paxos to select the master node which controls data update. If the master goes down, it will select a new master. Zookeeper can guarantee strong consistency. But its design goal is to provide distributed lock service and high availability service for other distributed systems. It is not a dedicated distributed storage system, so its performance is poor. Google's Chubby[11] is also based on Paxos protocol, which is similar to zookeeper.

Megastore[12] is a distributed storage system based on Paxos protocol developed by Google, which relies on Bigtable. It has the advantages of both the scalability of a NoSQL datastore and the convenience of the traditional RDBMS, and provides both strong consistency guarantees and high availability. However, it uses the Paxos protocol without being fully optimized, its write performance is not good.

Rao et al designed a scalable, consistent, and highly available data store by using Paxos protocol, which is called Spinnaker[13]. But it doesn't analyze the situation that two leaders may appear in one system. Its read and write performance are not good.

Based on the above, we designed PaxStore by using zookeeper cluster and high performance Leveldb engine. In addition, we used a number of optimization techniques, such as log optimization, circular lock-free queue etc. It can not only guarantee strong consistency, but also improve the system performance, and keep the system scalability.

3. Design of PaxStore

3.1 Architecture

All data are divided into different ranges based on the key value of every record. The basic components of PaxStore include client, zookeeper cluster and storage server which include leader and follower. The architecture of PaxStore is depicted in Figure 3.1. The replica's number can be configured, here we set it to be 3. Every range has a leader and two followers. Client only sends write requests to the leader which synchronizes data to followers based on our Paxos-based protocol, but both of the leader and the followers can provide read service. In order to simplify the leader election process, we use Zookeeper for auxiliary election. At the same time, Zookeeper can also monitor the system state. PaxStore can elect a new leader automatically and records the times of leader election as epoch. Each write request is assigned a number (sequence) to indicate its execution order. When a new node joins in system, it will run a zookeeper client and connect with zookeeper server, and then upload its metadata such as epoch, IP, and LSN (the largest write request sequence in log), into zookeeper server. At last, PaxStore uses an improved and optimized Leveldb

as local storage engine.

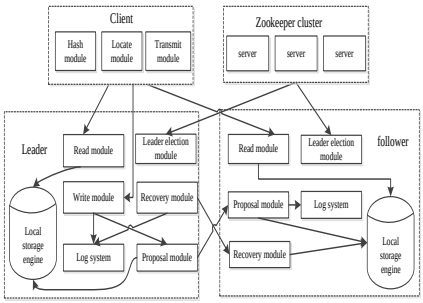


Figure3.1: PaxStore Architecture

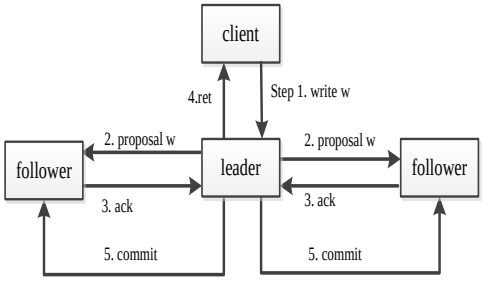


Figure 3.2: Protocol Flow Chart

3.2 Protocol Analysis

The basic process of the distributed replica protocol used by PaxStore is shown in Figure 3.2.

（1）Client sends write request (w) to the leader.

（2）After receiving W, leader firstly serializes W, appends W with epoch and sequence, then it writes the serialized W into log synchronously. In parallel with the log force, leader sends the serialized W to all of the followers.

（3）When the followers receive the proposal W message, they write it into log synchronously and send ACK message to leader.

（4）After writing W into the log and receiving more than 1 ACK message from followers, leader writes W into local storage engine, and send RET message to client.

（5）Furthermore, Leader periodically sends commit message to the followers to ask them to apply all pending write requests up to a certain sequence to their local storage engine.

Until now, the leader and followers have the same and the latest value of W.

From the above descriptions, it is obvious that under normal circumstances, the protocol overhead is extremely small, and only a RTT (Round-Trip Time) is needed to commit a write.

The client read protocol is also a Quorum-based protocol. As the follower may have an inconsistent state with leader for only a short time (leader periodically send COMMIT message to follower), we can choose either strong consistent read (read records from leader) or weak consistent read (read records from leader or followers). When choosing strong consistent read, the system needs first read record from leader and then read epoch message from a follower of this leader, if the follower has the same epoch message with leader, it shows that we read data successfully, otherwise the system errors occur.

4. Implementation

4.1 Component of Storage Node

The basic components of node are shown in Figure 4.1. It includes a log system, a storage engine and a zookeeper cluster. The replica consistency among multiple nodes is guaranteed by improved Paxos-base protocol which is described in Section 3.2. We choose Leveldb as our key-value storage engine, and replace its log module with our high available log system. The details of the log system and storage engine will be described in section 4.2.

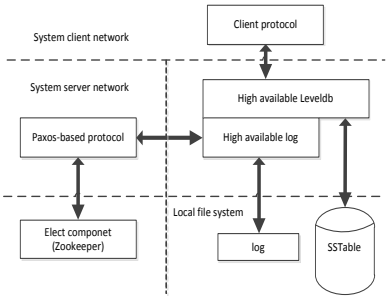


Figure 4.1: Component of Storage Node Figure

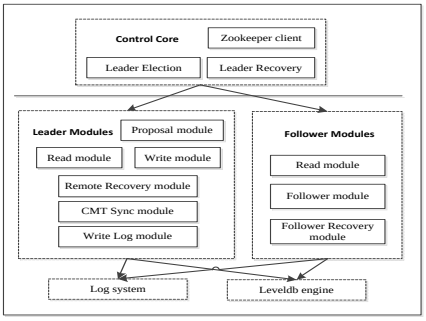


Figure 4.2: The Software Modules of Node

4.1.1 Software Modules of Node

The software modules of node are depicted in Figure 4.2. Each node has five functional modules, that is, control core, leader modules, follower modules, log system and Leveldb storage engine. The control core includes zookeeper client module, Leader Election module and Leader Recovery module. Leader modules include Write module, Read module, Proposal module, Remote Recovery module (help followers to recover data), CMT Sync module and Write Log module. Follower modules include Read module, Follower module (used to response the proposal request and CMT request sent by leader) and Follower Recovery module. If a node is leader, the running modules include control core, leader modules, log system and storage engine. If it is a follower, the running modules include control core, follower modules, log system and storage engine.

If the leader goes down, system will elect a new leader from the remaining alive nodes by their leader election modules. The new elected leader should first stop its old follower modules, and deal with all of the data that have been written into log but have been written into leveldb engine. It will write these data into storage engine and send these data to at least one follower to write into follower's local storage engine. Finally, the new elected leader starts all of the leader modules to become a real leader. Now, system can continue to run normally.

4.1.2 Leader Logic

The basic implementation framework of leader, which handles the client requests

by differentiating read and write.

（1）Leader execution logic

The design of read logic is simple. Read Worker thread manages the establishment and disconnection of read connection from the client. PaxStore can directly read the required data from local Leveldb engine. But in order to improve the read performance, we design a thread pool to use multi-core platform.

Write logic is the core part of the leader. The writing process is described in the following. First, Write Worker thread receives write request from client, then, it adds the request into Value Queue and sends a notify message to Proposal thread. Second, Proposal thread reads request from Value Queue, serializes it (i.e., adds epoch and sequence message) and then sends it to Proposal Round-robin Queue. The Proposal Queue is a circular lock-free queue which can reduce the synchronization overhead among threads. Third, PaxStore sends proposal message to follower, in parallel Write Log thread reads proposal message from Proposal Queue and then writes it into local log system. Once receiving at least half of the ACK message from followers (in our system, it needs to receive an ACK message), the system can write this request into local Leveldb engine and return Ret message to client. Periodically, Leader will also send CMT message to followers.

（2）Leader Election

The design principle of Leader election algorithm is to use a simple way to ensure that only one Leader can run normally at any time. The system cannot lose the committed write requests in leader election. If there is a majority of nodes alive, there must be the node containing all of the committed write requests. We only need to elect the node that has the largest LSN if it has the largest Epoch as leader.

The implementation of leader election needs the help of Zookeeper cluster. Every node will create an ephemeral file on the zookeeper server to save its metadata such as LSN, Epoch, and IP, when it joins system. If a node disconnects with zookeeper because of node failure, network partition or other reasons, its corresponding ephemeral file will disappear automatically. Once more than half of the nodes join system, they will compare their Epoch message and LSN message to elect a Leader. Leader will create an ephemeral Leader file on the zookeeper cluster to save its metadata. If Leader disconnect with zookeeper, this ephemeral Leader file will disappear automatically and system will elect a new leader.

In distributed systems, the case that there are two leaders may occur inevitably, as depicted in Figure 4.3, due to network reasons, A loses connect with zookeeper server, then system will do leader election again. B and C disconnect with A and C is elected as new Leader. But A may continue to run, so system has two leaders A and C at this time. PaxStore can ensure that only C can run normally. As no follower connects with A, even if it receives write requests, it can’t execute these write requests successfully because it can’t receive ACK. System will force to stop A until the client and zookeeper server find that A is in the isolate state. This can deal with the situation of the two leaders.

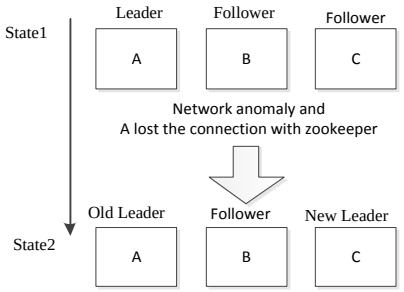


Figure 4.3: Two Leaders appear

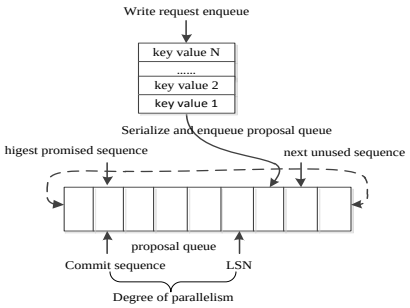


Figure 4.4: The Proposed Round-robin Queue

（3）Leader Design Optimization

Parallel processing optimization: firstly, leader executes the proposal sending and log writing in parallel, and then PaxStore executes multiple write requests in parallel. PaxStore can handle multiple proposal messages simultaneously. As shown in Figure 4.4, the commit sequence represents the largest committed request sequence, the highest promised sequence represents the largest request sequence that has receive ACK message, the LSN represents the largest request sequence that has been written into log system, the next unused sequence represents the smallest sequence number that has not been used. The requests between commit sequence and highest promised sequence are not written into Leveldb storage engine; the requests between highest promised sequence and next unused sequence are not proposed. The next unused sequence minus commit sequence is the current degree of parallelism. In order to control the system delays, we set an appropriate degree of proposal parallelism. To avoid proposal lost, as well as out-of-order problems, PaxStore uses TCP protocol and sets the TCP's sending buffer and receiving buffer to an appropriate value.

4.1.3 Follower Logic Design

The basic implementation framework in the follower is depicted in Figure 4.5. The basic implementation framework of follower is similar to Leader, but follower works relatively simpler than Leader. The design of read logic of follower is the same as leader. Follower does not have to deal with the client writes directly. It receives the proposal message sent by Leader, and then detects whether the sequence of proposal message is continuous or not; if it is, it receives this proposal and puts this proposal message into Fproposal Queue, follower writes this proposal into local log system and sends ACK to Leader. Because the communication between Leader and Follower uses TCP protocol, it ensures that the sequence of proposal message sent by Leader is continuous, if the proposal message sequence received by follower isn't continuous, Paxos-based protocol will not work normally; then follower will exit from system.

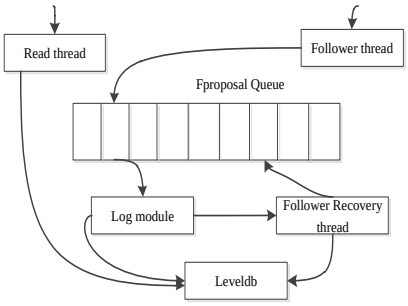


Figure 4.5: Follower Execution Logic

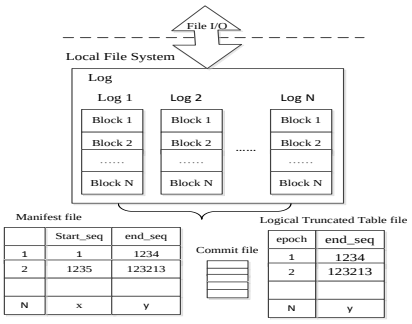


Figure 4.6: Log System Figure

If a new follower joins in system, it starts the follower recovery thread to finish recovery, which includes local recovery and remote recovery. The follower recovery mechanism will be depicted in section 4.4.

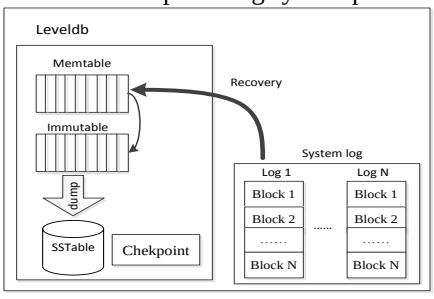
4.2 Implementation of Log and Storage Engine

The Log System is an important component of PaxStore. It stores both the data and metadata required by the normally running of PaxStore. Furthermore, log can also ensure that system can automatically complete the recovery.

The log structure is shown in Figure 4.6. The Log System is designed based on local file system. The threshold of each log file size can be configured. When reading data, we use block as a unit and the block size can be configured. The manifest file records the metadata of each log file and helps us to locate log file when reading data.

Logical Truncated Table file records the largest corresponding commit sequence of each Epoch, which can help determine which record can be read, and which record needs to be discarded when in recovery.

The above files constitute the basic log system. In order to meet the requirements of the strong system consistency, every write operation is synchronous, so the disk overhead is relatively large. In order to improve system performance, we use the overwrite method to optimize the log system, that is, we pre-allocate a fixed size of log file and clear all of the data content of the file, and then write all of the records into the file by using fdatasync() function instead of fsync. The fdatasync function has a much high performance than fsync because it needn't to update metadata of file. This method can improve log system performance.



4.7: Paxtore Storage Engine

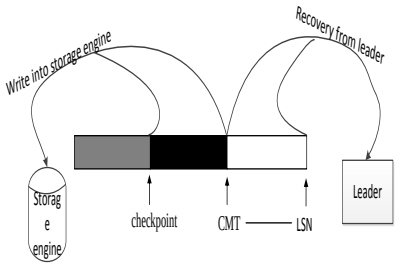


Figure 4.8: Log Layout

Leveldb log module is used to do local recovery for itself. PaxStore has its local log system, and Leveldb can get all of its needed data from PaxStore log system. So we modify Leveldb and remove its log module. As shown in Figure 4.7, Leveldb can get all of the data from PaxStore log system when in local recovery.

4.3 Recovery

4.3.1 Follower Recovery

Follower recovery is different from ordinary database recovery; it contains local recovery and remote recovery.

As shown in Figure 4.8, the records before checkpoint have been written to storage engine, so we need to recover these records. The records between checkpoint and CMT have been committed, so we can read them from local log system directly. The records between CMT and LSN are not yet confirmed, we need to do remote recovery, and they may have been committed and may be stale. In order to ensure complete recovery, follower should send remote recovery request to leader, receive recovery data and write these data into Leveldb (storage engine).

4.3.2 Leader Recovery

When the leader goes down, system will elect a new leader; then the new leader should do leader recovery work. New leader should re-propose the requests between CMT and LSN because these data may return to client already or haven't been committed. After at least one follower and new leader both write these data into storage engine, system can run normally.

5. Performance Evaluation

5. 1 Write Latency

Write delay is an important parameter of evaluating our system and protocol. We optimize our log system, that is, we pre-allocate a fixed size of log file in order to use the overwrite method rather than append write method to write records. We inject 10,000 records with the same size into system. The size of write requests ranges from 512 Bytes to 8192 Bytes every time and all of the write log operations are synchronous. We compare the write latency of PaxStore between overwrite and append write method. As shown in Figure 5.1, the write latency increases with the increase of write requests size. In addition, the performance of overwrite method is much higher than append write method. This is because when it uses overwrite method, the log data block has been previously allocated, every write operation doesn't require the high overhead of disk seek operation, and every synchronous log write operation doesn't need to write metadata of log file by using fdatasync. The results show that our optimization of log can significantly improve PaxStore performance.

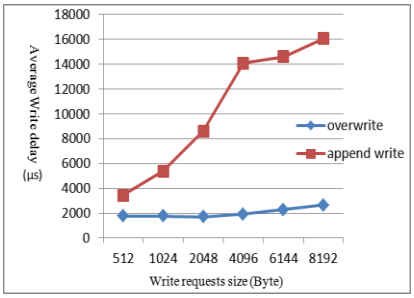


Figure 5.1: Write Latency

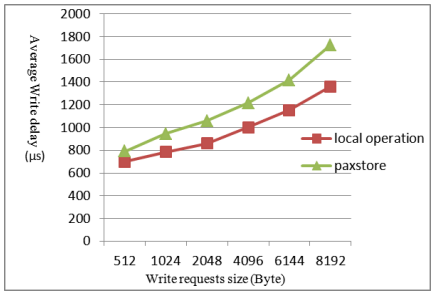


Figure 5.2: Protocol Overhead

5. 2 Protocol Overhead

As shown in Figure 5.2, we firstly set the replica's number is1, that is, leader doesn't send any data to other nodes to measure the latency of local operations. The log uses asynchronous write mode. Then we set replica's number is 3 to measure the write latency of PaxStore, and the log uses the same write mode too. The figure shows that the overhead of our Paxos-based protocol is small which increases by about 20% overhead over the local operation. There are many reasons, for example, the execution of every write request only needs one RTT; write disk operation and network communication work in parallel when dealing with a write request; we use a circular lock-free queue which can reduce overhead caused by locking.

5. 3 Comparison with Zookeeper

As shown in Figure 5.3, we compare the write performance of PaxStore with the older system which also uses a kind of Paxos-based protocol. Both of their logs use a synchronous write mode. When the size of write request is more than 2000 Bytes, PaxStore is five times or more as fast as the older. This is because the older is not a specialized storage system. Furthermore, we use a variety of methods to optimize PaxStore, such as overwrite log system, round-robin queue, disk and network works in parallel etc. The results show that PaxStore has a very high write performance.

Figure 5.3: Compare PaxStore with the older

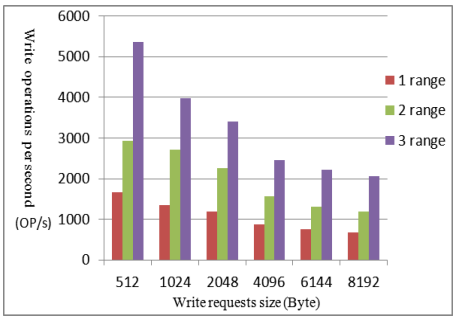


Figure 5.4: System Scalability

5. 4 System Scalability

System can divide all of the data into some ranges based on the key, and each write request can only be written into one range. Every range has its own leader and followers. As shown in Figure 5.4, we test the system performance based on different data range number. In order to achieve optimal performance, every node runs only one PaxStore instance. Obviously, the system performance has a linear growth with the increase of data range number regardless of how much the size of write requests is. The results show that PaxStore has a linear scalability.

5.5 System Recovery

For distributed systems built on the commodity machine, node failure is frequent. In PaxStore, we set replica's number as 3, if one follower goes down, system can run normally, but if two nodes failure, system will stop service. System will elect a new leader from the remaining two nodes when the leader goes down. This process is very fast. The system can complete the leader election using less than 3s latency, which doesn't have a huge impact on the normal running of system. This is because once the leader goes down, zookeeper cluster will immediately perceive this situation and notify the other nodes, and system can elect a new leader by comparing the metadata of existing followers. Because zookeeper needs time to clean up obsolete information and receive new information, it may has 3s delay.

It is important to measure the recovery speed of our system when new follower joins in system and recovers to the current state of the system. As shown in Figure 5.5, we first write 10,000 records into system, and then new follower joins in system and recovers these 10,000 records. The size of write request is range from 512 to 8192 Bytes. The experiments show that the recovery of 10000*512 Bytes size records only needs 8s and 10000*8192 only needs 15s.

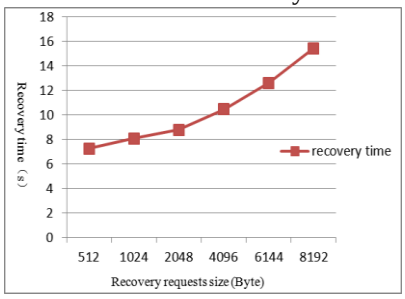


Figure 5.5: Follower Recovery Time

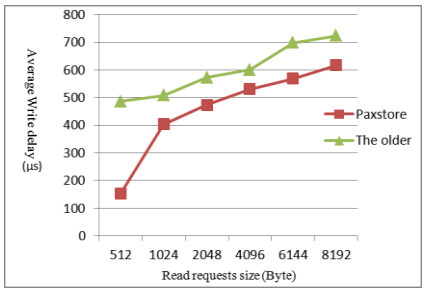


Figure 5.6: Read Latency

5.6 Read Latency

Because the read operation is not related to the complicated protocol, we only need to read data from local Leveldb engine, so the system read performance is basically the same as Leveldb. As shown in Figure 5.6, we firstly write 500,000 records into PaxStore, and then read the data based on random key. The records size ranges from 512 to 8192 Bytes. Results show that the read delay is only 150μs when records size

is 512 Bytes, and the read delay increases as the records size increases. Besides, we test the read performance of the older. It is obvious that the read latency of PaxStore is much smaller than the older regardless of how much the size of request is.

5.7 Summary and Result

From the above testing, it is clear that PaxStore has a high performance. The log optimization technology improves the system performance significantly. Our protocol overhead is small which increases 20% overhead over local operation. The write performance is five times over Zookeeper. PaxStore also has a quick recovery speed.

6. Conclusion and Future Work

This paper designs and implements a consistency, high availability, distributed key value storage system, called PaxStore. In the PaxStore, we optimize its log system, circular lock-free queue and Paxos protocol. PaxStore has a high performance and lower protocol overhead. The results show that Paxos-based protocol is a good tool to implement this kind of system[15-16]. By using high available service module, including Chubby and Zookeeper, to do leader election, it can not only improve system performance and avoid a single point of failure, but also simplify the design of PaxStore. The practical experience of PaxStore has constructive value for other high-availability storage system designs.

In future work, we will use write batching method[17] to improve disk utilization and chained push method[18-19] to reduce the network overhead of leader.

Acknowledgments

This work is supported by 973 project 2011CB302301, the National Basic Research 973 Program of China under Grant by National University's Special Research Fee (C2009m052, 2011QN031, 2012QN099), Changjiang innovative group of Education of China No. IRT0725, is supported by Electronic Development Found of Information Industry Ministry.

REFERENCES

[1] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens. Paxos replicated state machines as the basis of a high-performance data store. In: NSDI'11. Proceedings of the 8th USENIX conference on Networked systems design and implementation. Berkeley: USENIX Association, 2011. 11~11

[2] Maged Michael, José E. Moreira, Doron Shiloach. Scale-up x Scale-out: A Case Study using Nutch/Lucene. Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. Page(s): 1-8. March 2007.

[3] E. A. Brewer. Towards Robust Distributed Systems. In PODC, pages 7–7, 2000.

[4] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In SIGMOD, pages 1–8,1984.

[5] Yoav Raz (1995): "The Dynamic Two Phase Commitment (D2PC) protocol ",Database Theory — ICDT '95, Lecture Notes in Computer Science, Volume 893/1995, pp. 162-176, Springer, ISBN 978-3-540-58907-5.
[6] L. Lamport. Paxos Made Simple. ACM SIGACT News, 32(4):18–25, December 2001.
[7] http://research.microsoft.com/users/lamport/pubs/pubs.html#lamport-Paxos
[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In SOSP, pages 205–220, 2007.
[9] Avinash Lakshman, Prashant Malik, Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review archive Volume 44 Issue 2, April 2010 Pages 35-40.
[10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-Free Coordination for Internet-scale Systems. In USENIX, 2010.
[11] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live: An Engineering Perspective. In PODC, pages 398–407, 2007.
[12] J. Baker et al. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In Conf. on Innovative Data Systems Research, 2011.
[13] Jun Rao, Eugene J. Shekita, Sandeep Tata. Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore. VLDB, 2011.
[14] Leveldb: A fast and lightweight key/value database library by Google. http://code.google.com/p/leveldb/.
[15] Atul Adya, William J. Bolosky, Gerald Cermak, etal. Farsite: federated, available, and reliable storage for an incompletely trusted environment. In: OSDI'02. Proceedings of the 5th symposium on Operating systems design and implementation. New York: ACM, 2002.1~14
[16] C. Coulon, E. Pacitti, and P. Valduriez. Consistency management for partial replication in a high Performance database cluster. In: ICPADS 2005. Proceedings of 11th International Conference on Parallel and Distributed Systems. USA:IEEE, 2005. 809~815
[17] N. Santos and A. Schiper, Tuning Paxos for high-throughput with batching and pipeliing. In: ICDCN'12. Proceedings of the 13th international Conference on Distributed Computing and Networking. Berlin: Springer, 2012:153~167
[18] P. Marandi, M. Primi, N.Schiper, etal. Ring Paxos: A high-throughput atomic broadcast protocol. Dependable Systems and Networks, 2010,7129:153~167
[19] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In: OSDI'04. Proceedings of the 6th conference on Symposium on Opearting Systems Design And Implementation. San Francisco, CA, USA: USENIX Association, 2004. 7~8