Exploring Eventual Consistency Support in ZHT

Shukun Xie, Ran Xin
Illinois Institute of Technology
sxie11@hawk.iit.edu, rxin@hawk.iit.edu

ABSTRACT

ZHT, short for zero-hop distributed hash table, aims to provide high availability, good fault tolerance, high throughput and low latencies, at extreme scales of millions of nodes. To reach these goals we designed and implemented the replication scheme and eventual consistency support. Replication is one of the most commonly used fault tolerance scheme in distributed system, with which single node failure won't affect the availability of the system. Also, we applied eventual consistency as the consistency model. Eventual consistency allows updating replica asynchronously, which can reduce the latency of requests, and provides high reliability, which means every data access can get the latest updated value. Our experiments provided performance evaluation of different consistency strategies.

Keywords

ZHT, Replication-based Fault Tolerance, Eventual Consistency

1. INTRODUCTION

Reliability at massive scale is one of the most import issues in distributed system. Replication is a well-known fault tolerance strategy for distributed storage system. The discussion of replication has been started a long time since early database works [1]. Replication can provide two major advantages, which are: 1) a read request is higher possible to be satisfied than if the data only has a single copy in the network; 2) it offers the opportunity to distribute the workload on frequently accessed data. However, we need to take careful consideration when we work on the consistency issue with replication. Properly choose the time when to resolve inconsistency is very important to the performance. Otherwise it may increase the latency and operation rejection possibility of distributed system, especially when node failure happens. [2]

ZHT [3, 23], zero-hop distributed hash table, aims to provide high availability, good fault tolerance, high throughput and low latencies, at extreme scales of millions of nodes. It applies replication-based fault tolerance and our project works on the consistency issue for ZHT. Different systems tend to apply different consistency strategies due to different tradeoff based on system requirements. For ZHT, eventual consistency is a better solution. It allows certain inconsistency window exists after write happens and resolve it when a lookup to the corresponding data happens. Therefore, it can guarantee the reliability at the situation where primary server is unable to reach without heavy performance loss on latency. This report will deeply introduce our project in following organization: Section 2 will provide background knowledge of the technique we applied and the system we worked on, and also the design motivations. Section 3 will includes our design and implementation in detail. Section 4 will talk about our experiments and results on performance. Section 5 and 6 will introduce some related work on consistency issues and the conclusion.

2. BACKGROUND AND MOTIVATION

2.1 Background

Our project aims to provide eventual consistency support for ZHT. As we discussed in Section 1, the goal of ZHT is a building block for future distributed systems, such as parallel and distributed file systems, distributed job management systems, and parallel programming systems. Since replication is used to provide fault tolerance, consistency between multiple replicas need to be considered. Currently, the replicas in ZHT have distinct orders, which means client requests always go to a single replica (e.g. primary replica when it is applicable). At this situation, consistency is straightforward to be maintained. The primary and secondary replicas are strongly consistent and the other replicas are weakly consistent, which makes ZHT follow weak consistency model. This currently design has its potential problem of data loss, inconsistency and unbalanced workload.

Our work focuses on providing better consistency model support. Basically, we have two options, which are strong consistency and eventual consistency. Strong consistency requires primary actively updates the data on all the replicas before it returns the write request acknowledgement to clients. [2][4] So, strong consistency can provide high data availability because the following data accesses can always achieve the latest updated value. However, this takes expensive cost. And in some systems with strong consistency, like Harp, [5] write operations only commits when the updating is successful among all replicas, which could make operation highly possibly rejected when the machines are not very stable or do not have great recovery mechanism. In addition, only primary can serve lookup requests because replicas cannot return latest updated value when the inconsistency repair procedure from primary to replicas is undergoing.

Unlike strong consistency, eventual consistency [7] [12] only requires primary updating the secondary replica before it return write request acknowledgement to client and rest replicas can be asynchronously updated afterward by each other. This can reduce the latency of write operation. To maintain the reliability, eventual consistency also applies version exchange. Every key-value data copy has a version, and a version comparison procedure happens when a replica received a lookup request. So, part of inconsistency is detected and repaired on lookup requests. Although the version exchange process will slow down the lookup operation responses, overall eventual consistency can lead to fault-tolerant, highly available and low-latency. So, compared with strong consistency, eventual consistency is the best relative solution for ZHT. Many distributed system are using eventual consistency, like CouchDB [9], MongoDB [10], and Cassandra [11].

2.2 Motivation

ZHT applies replication-based fault tolerance and replicas have distinct ordering in terms of which ones are accessed by clients. Therefore, the failure of a single node doesn't affect ZHT as a whole. However, to make sure every data access can get the latest updated value, we still need to work on the consistency issue between primary and replicas of the same key-value pairs when write operation happens on primary.

In addition, due to the distinct ordering of replicas, only primary is serving all the request and clients can interact with replicas only at the time when primary is inaccessible, which means the access pressure on primary is higher than replicas. We want distribute some of the requests on primary to replicas in order to get a better latency performance.

3. PROPOSED SOLUTION

3.1 Overview

Our project aims to design and implement a technique in a real distributed system and evaluate its performance. Majorly we focus on two parts shown as follows.

Replication: Our project sets up a replica list for each server (primaries and replicas). Then every server knows which else holds the same data and it can communicate with these servers when it is necessary.

Eventual Consistency: With eventual consistency, our project allowed replicas to serve lookup requests. Whenever a replica received a lookup request, it will first find the latest version of the requested data and send a version comparison request with primary. If the primary detects a version conflict exists, which shows the inconsistency, it will send the latest data to the replica. In addition, when a primary received a write operation, it will update its secondary replica after it assigned the data the latest updated version and executed this write operation on itself.

3.2 Replica List

Our project sets up a Replica List based on the Neighbor List for each server when it starts. The Replica List records the host names and ports of all the servers stored the same data, including the one this list locates on. Our project treats the Replica List as a preference list, which means when a server wants to update a replica, it will find the first replica in this list indexed after it, and when it finds a server is failed to communicate, it will treat the first server indexed after the failed one in Replica List to be the substitution. The construction of this Replica List follows the following rules:

1. The number of replicas is configured by "NUM_REPLICAS" in "zht.conf". The size of Replica List should be this number plus 1.
2. For each data item, we use the hosts indexed after the host of the Primary server that holds this data item in Neighbor List (We see Neighbor List as circular, which means when the index reaches the end of the list, it starts over from the head) to be the host of Replica servers.
3. The port differential between every two adjacent servers in Replica List is configured by "PORT_DIFFERENTIAL" in "zht.conf". With this port differential and the port of the server itself, we can easily get the port of Primary server or other Replica servers.

The ports of primary servers are configured in file "zht.conf" with "PORT". And the ports of replica servers are configured with command that starts them. Combine with the port differential, servers are able get their position in the Replica List, which is useful for data synchronization and version comparison for eventual consistency.

3.3 Lookup

Both of primary and replica servers are able to serve Lookup requests. Since the number and selection rule of replicas for each data item is determined (see Section 3.2), after client found the primary server that holds the requested data through hash function, it can randomly select a primary or replica to serve this request. The major difference between primary and replica on serving Lookup requests is that primary returns local lookup result directly to clients, which is shown in Figure #. Replica need first find the latest version of the requested data it holds and compare it with primary, which is shown in Figure #. We designed two types of version comparison requests, version compare and existence check, corresponding to the situation that whether replica contains the requested data. If version conflict is found on primary, it knows that inconsistency exists between it and the replica which sent this version compare request. Then it will send the latest data to replica or notify the replica that the data has been removed, and the replica will first update itself before it returns the updated data to the client. Otherwise, the primary will simply return an acknowledgement and replica knows it already has the latest updated version of the data, or the data was removed from primary either. Then this replica can return the data it holds to client. The flow chart in Figure 3 and 4 shows how replicas serve lookup request and primary serves version comparison request in detail.

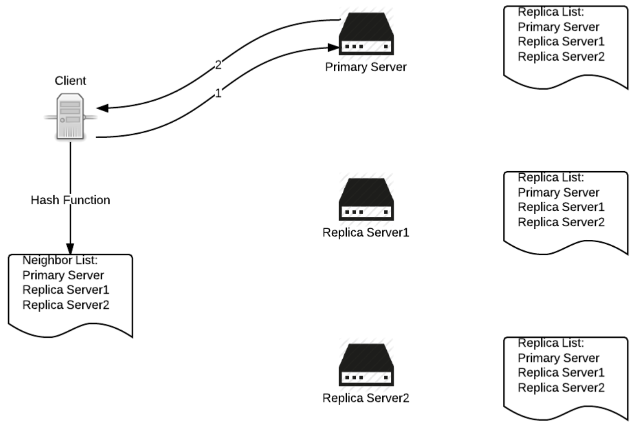


Figure 1 Lookup Request Served by Primary Server: 1) Client sends Lookup request to Primary Server; 2) Primary sends Lookup result to Client

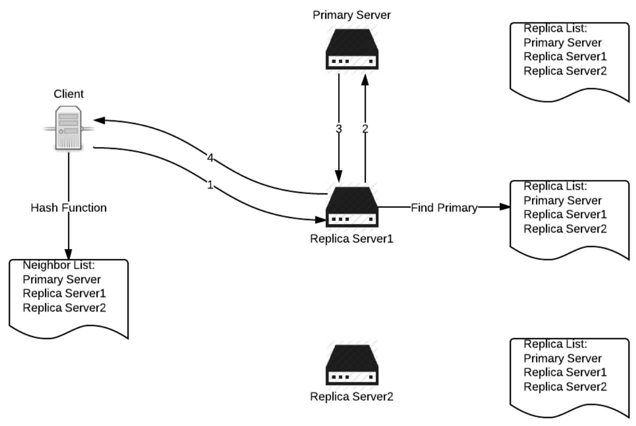


Figure 2 Lookup Request Served by Replica Server: 1) Client sends Lookup request to Replica Server; 2) Replica sends Version Compare request to Primary Server; 3) Primary sends Version Compare result to Replica Server; 4) Replica server sends Lookup result to Client

3.4 Insert/Append/Remove

Unlike Lookup requests, only primary servers can serve these three kinds of write requests. For Insert and Append requests, primary servers will first execute local lookup to find the current latest version of the data item with the same key and update the version of the data sent by clients before it executs a local Insert or Append. In addition, after local Insert or Append is finished, primary servers actively forward this Insert/Append request received from client to the first replica, which is the secondary replica we mentioned above, in its Replica List to repair the inconsistency. It will return the request acknowledgement to client after it receives a successful acknowledgement from the secondary replica. Therefore, we can see client, primary server and the secondary replica are maintained with strong consistency. We do this for failure handling (Section 3.5). For remove requests, the primary also need to update the secondary replica with the same procedure, except it doesn't need execute local lookup first to get the latest version.

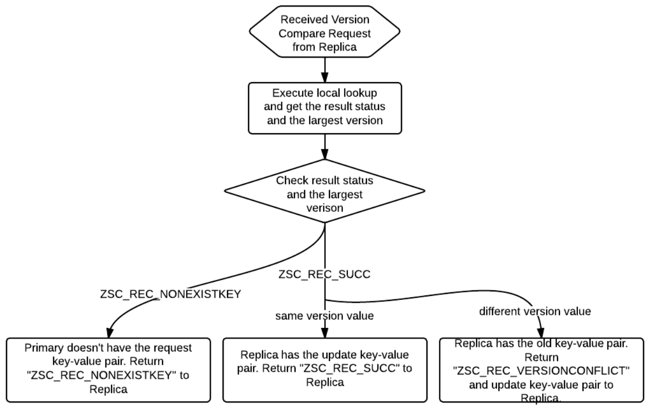
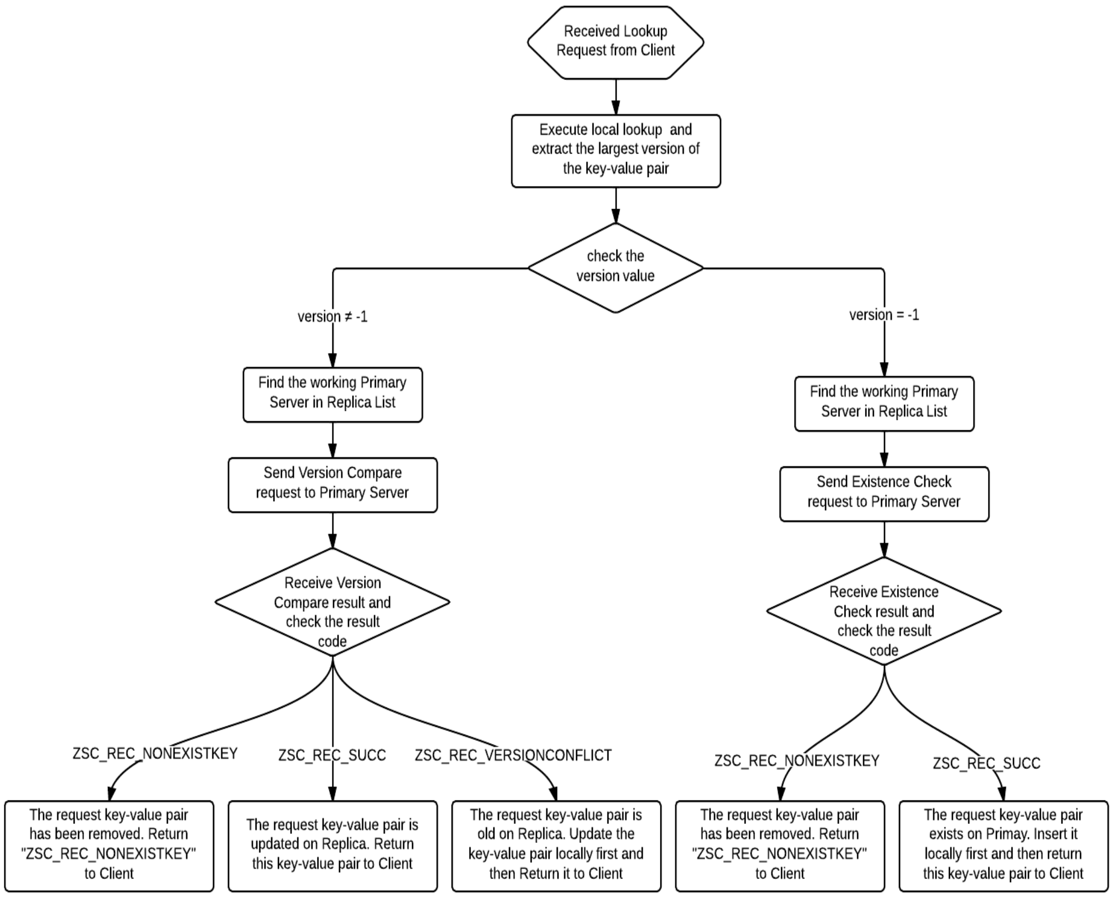

Received Version Compare Request from Replica
Execute local lookup and get the result status and the largest version
Check result status and the largest verison
ZSC_REC_NONEXISTKEY
ZSC_REC_SUCC
same version value
different version value
Primary doesn't have the request key-value pair. Return "ZSC_REC_NONEXISTKEY" to Replica
Replica has the update key-value pair. Return "ZSC_REC_SUCC" to Replica
Replica has the old key-value pair. Return "ZSC_REC_VERSIONCONFLICT" and update key-value pair to Replica

Figure 3 Flow Chart of Primary Serves Version Compare Requests

When a replica receives a write request from a primary or another replica and there are other replicas indexed after it in its Replica List, it needs to update the next replica after it executed local write. To reduce the latency, at first, we create a thread for updating replica task, put this thread into a queue for following execution and immediately return acknowledgement to primary/replica. However, this method restricts the scalability of our system due to the extreme large number of newly created thread when the number of client and servers is increased to a relative large scale. Then we modified our design and put the request need to forward to another replica into a queue and return the Insert/Append/Remove acknowledgement to the sender immediately. In this method we only create one thread at the time this replica receives the first write operation, which keeps running in a loop waiting for some request that is pushed into the queue. This implies we don't require the consistency between replicas is strong. Figure 5 shows the communication between primary and replica for write operations. Figure 6 and 7 shows the flow chart of how primary server Insert/Append requests and how replicas forward Insert/Append request. Due to remove request is simpler than the other two write operations, we don't provide flow chart specifically.

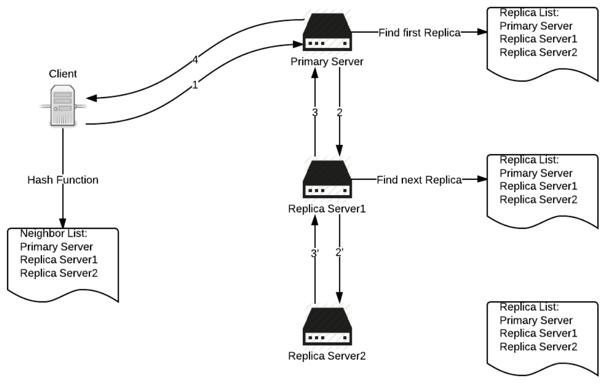
Combine this active inconsistency repair strategy and version comparison mentioned in Section 3.2, our project



Received Lookup Request from Client
Execute local lookup and extract the largest version of the key-value pair
check the version value
version ≠ -1
version = -1
Find the working Primary Server in Replica List
Send Version Compare request to Primary Server
Receive Version Compare result and check the result code
Send Existence Check request to Primary Server
Receive Existence Check result and check the result code
ZSC_REC_NONEXISTKEY
ZSC_REC_SUCC
ZSC_REC_VERSIONCONFLICT
The request key-value pair has been removed. Return "ZSC_REC_NONEXISTKEY" to Client
The request key-value pair is updated on Replica. Return this key-value pair to Client
The request key-value pair is old on Replica. Update the key-value pair locally first and then Return it to Client
The request key-value pair exists on Primay. Insert it locally first and then return this key-value pair to Client

Figure 4 Flow Chart of Replica Server Lookup Request and Sent Version Compare Request.

could guarantee that every access to the data item will get the updated value and the latency of write operation will be reduced.



Client
Hash Function
Neighbor List: Primary Server Replica Server1 Replica Server2
Primary Server
Replica Server1
Replica Server2
Find first Replica
Find next Replica
Replica List: Primary Server Replica Server1 Replica Server2
1
2
3
4
2'
3'

Figure 5 Insert/Append/Remove Requests: 1) Client sends Insert/Append/Remove request to Primary Server; 2) Primary Server synchronizes I/A/R request to first Replica; 3) First Replica sends I/A/R acknowledgement to Primary; 4) Primary Server sends Insert/Append/Remove acknowledgement to Client

3.5 Primary Failure Handling

The major consideration for replication is fault tolerance, so it is very important for us to design eventual consistency support with consideration for primary server failure.

The basic idea is that we create an attribute called "reachable" to each entry in Neighbor list and Replica list. This attribute is used to record the host communication history. Every time the host in the list is failed to communicate, the reachable value is increased by 1 and this value can at most be 2. When the reachable value is set to be 2, it is treated to be unreachable. Otherwise, if this host is successfully be reached before the value of reachable is set to be 2, its reachable value will be reset to be 0.

When a client tries to find a server to serve a request, it will search all the hosts hold the primary and replicas of the requested key-value pair in the Neighbor list, until it find one with reachable value less than 2. The client will treat this hosts as the primary and the replicas start from it as reachable. If this request is Lookup, our project will randomly select one among these reachable hosts to serve this request. If this request is write request, our solution will use the found host to serve this request. Figure 11 shows the flow chart on client for this situation.

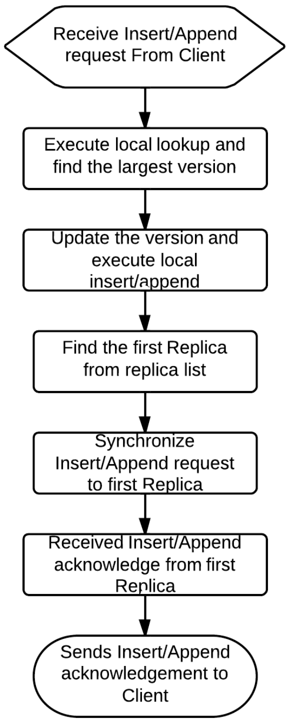

Receive Insert/Append request From Client
Execute local lookup and find the largest version
Update the version and execute local insert/append
Find the first Replica from replica list
Synchronize Insert/Append request to first Replica
Received Insert/Append acknowledge from first Replica
Sends Insert/Append acknowledgement to Client

Figure 6 Flow Chart of Primary to serve Insert/Append Request

When a primary/replica receives a Lookup request sent from client, it needs to check with primary of the version of the requested key-value pair. Similarly with client, this primary/replica needs to check its Replica list first to find the entry with reachable value less than 2 and try that server. If this server is not working, this primary/replica will send version comparison request again with the next entry in the Replica List. Figure 9 and 8 shows an example of the situation when Lookup request falls on a failed primary server and the flow chart on primary/replica including sending version compare request.

When a primary/replica receives a write request from client, it knows it need to execute full primary function, which has to update the secondary replica. The secondary replica is the one with reachable value less than 2 and indexed after this primary/replica in Replica List. Figure 10 shows the communication between primary and replicas when a write operation is served with primary failure. In Figure 12, we only showed the flow chart of queue processing, which is the major part different from how Section 3.4 forwarding requests.

We validated the failure handle solution on 8 physical nodes. After two primary servers have been down manually, the whole system could still serve requests well. We will validate the failure handle mechanism scale up to more nodes in the future.

3.6 Laziness and Unreliable Eventual Consistency

Laziness eventual consistency is an extreme version of our eventual consistency strategy. It only relies on version comparison to maintain consistency. It will provide low latency for write operation, because no updating replicas happen when serving such request. But, the latency of Lookup requests will be very high, due to the version comparison and following data transmission. Also, all the inconsistency repairs happen when a Lookup request is waiting for response, the overall throughput may **also** be low. In addition, it is not reliable. If no lookup ever requests some key-value pair and the primary holds this data is power off, this data will be lost. Laziness eventual is part of our initial designs. We will use it in the performance evaluation to show the performance improvement when we combine two mechanisms, version comparison and active inconsistency repair in our solution.

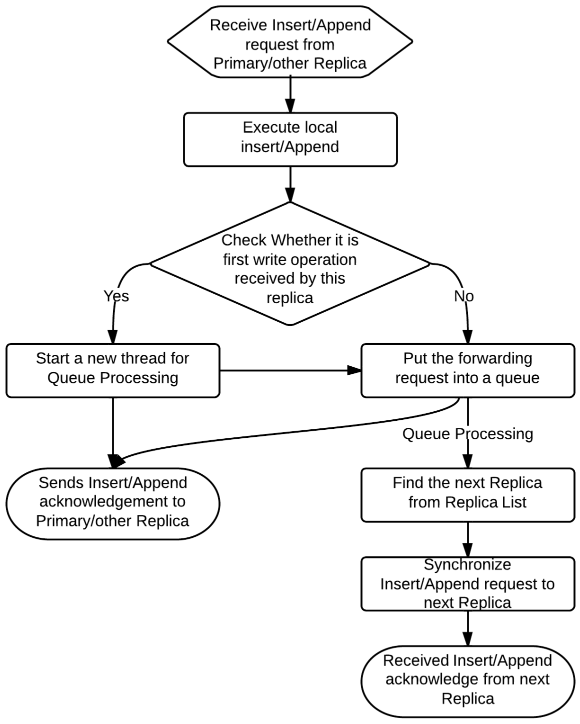

Receive Insert/Append request from Primary/other Replica
Execute local insert/Append
Check Whether it is first write operation received by this replica
Yes
No
Start a new thread for Queue Processing
Put the forwarding request into a queue
Queue Processing
Sends Insert/Append acknowledgement to Primary/other Replica
Find the next Replica from Replica List
Synchronize Insert/Append request to next Replica
Received Insert/Append acknowledge from next Replica

Figure 7 Flow Chart of Replica serving Insert/Append request from primary/other replica

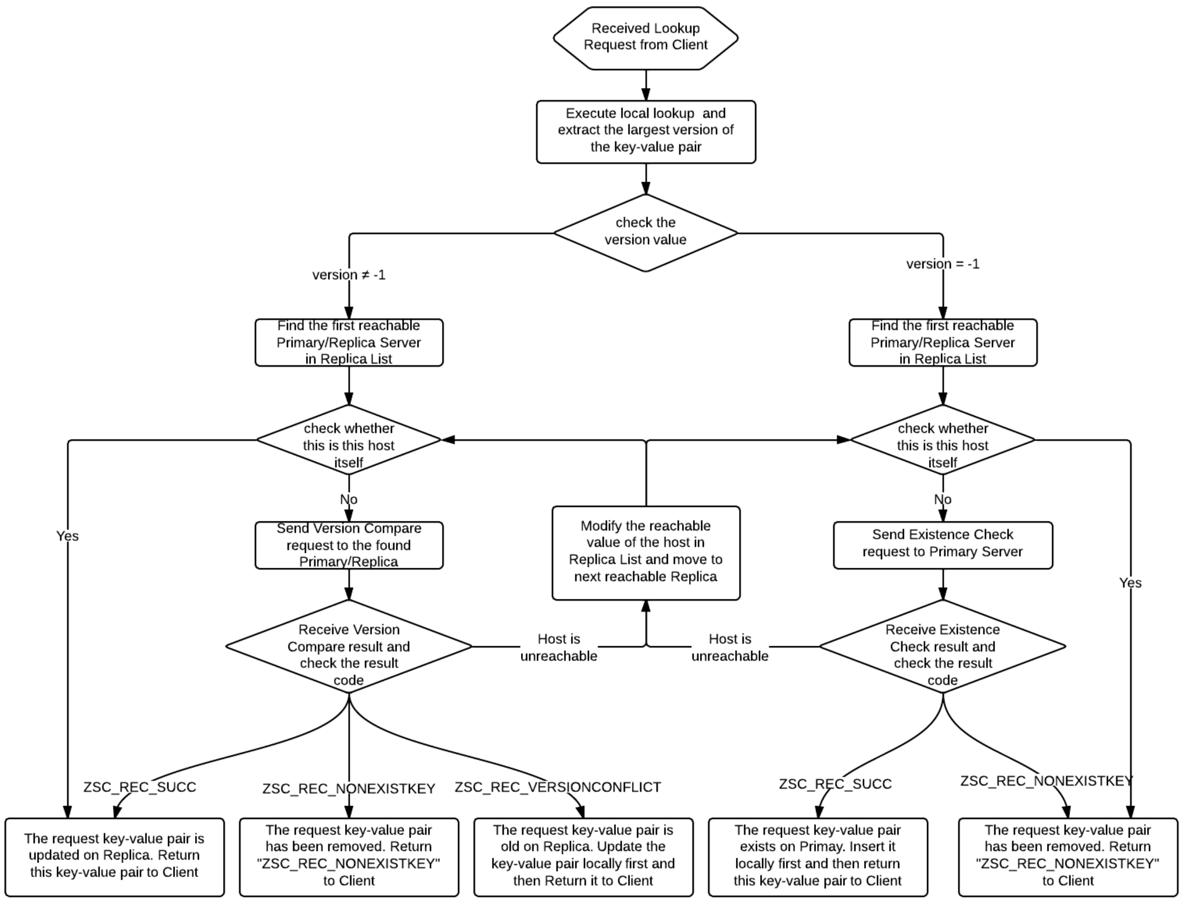

Received Lookup Request from Client
Execute local lookup and extract the largest version of the key-value pair
check the version value
version ≠ -1
version = -1
Find the first reachable Primary/Replica Server in Replica List
Find the first reachable Primary/Replica Server in Replica List
check whether this is this host itself
check whether this is this host itself
Yes
No
No
Yes
Send Version Compare request to the found Primary/Replica
Modify the reachable value of the host in Replica List and move to next reachable Replica
Send Existence Check request to Primary Server
Receive Version Compare result and check the result code
Host is unreachable
Host is unreachable
Receive Existence Check result and check the result code
ZSC_REC_SUCC
ZSC_REC_NONEXISTKEY
ZSC_REC_VERSIONCONFLICT
ZSC_REC_SUCC
ZSC_REC_NONEXISTKEY
The request key-value pair is updated on Replica. Return this key-value pair to Client
The request key-value pair has been removed. Return "ZSC_REC_NONEXISTKEY" to Client
The request key-value pair is old on Replica. Update the key-value pair locally first and then Return it to Client
The request key-value pair exists on Primay. Insert it locally first and then return this key-value pair to Client
The request key-value pair has been removed. Return "ZSC_REC_NONEXISTKEY" to Client

Figure 8 Flow Chart on Primary/Replica serves lookup request with Primary Failure

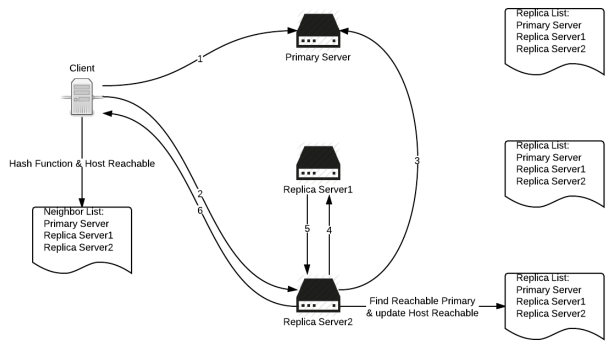

Client
Primary Server
Replica Server1
Replica Server2
Hash Function & Host Reachable
Neighbor List: Primary Server Replica Server1 Replica Server2
Replica List: Primary Server Replica Server1 Replica Server2
Replica List: Primary Server Replica Server1 Replica Server2
Replica List: Primary Server Replica Server1 Replica Server2
Find Reachable Primary & update Host Reachable
1
2
3
4
5
6

Figure 9 Lookup To Primary with Primary Failure: 1) Client sends Lookup request to Primary Server; 2) Client sends Lookup request to random Replica Server (Replica 2); 3) Replica 2 sends Version Compare request to Primary Server; 4) Replica 2 sends Version Compare request to Replica Server 1; 5) Replica 1 sends Version Compare result to Replica Server 2; 6) Replica 2 sends Lookup result to Client.

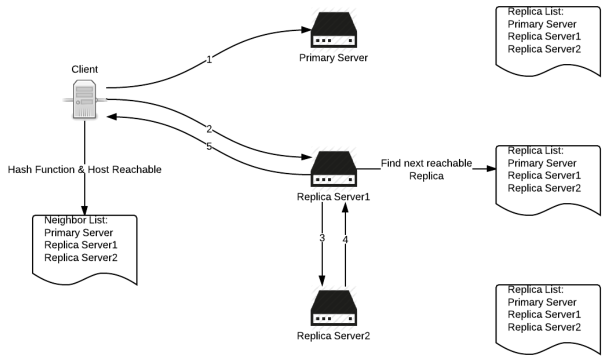

Client
Primary Server
Replica Server1
Replica Server2
Hash Function & Host Reachable
Neighbor List: Primary Server Replica Server1 Replica Server2
Replica List: Primary Server Replica Server1 Replica Server2
Replica List: Primary Server Replica Server1 Replica Server2
Replica List: Primary Server Replica Server1 Replica Server2
Find next reachable Replica
1
2
3
4
5

Figure 10 Insert/Append/Remove with Primary Failure: 1) Client sends I/A/R request to Primary Server; 2) Client sends I/A/R request to next reachable Replica (Replica 1); 3) Replica 1 synchronizes I/A/R request to next reachable Replica (Replica 2); 4) Replica 2 sends I/A/R acknowledgement to Replica 1; 5) Replica 1 sends I/A/R acknowledgement to Client

Unreliable eventual consistency is a variant on our eventual consistency. The data copy on replicas would be updated both on read and write operations. However, the primary would return back the result to client after the completion of update on itself, while an update process would be issued between primary server and secondary replica. Only the data on primary server could get the latest data in this case. If the primary is down when the secondary replica is updating, the data may be lost. This unreliable eventual consistency provides a low latency on write with unreliable issues.

3.7 IMPLEMENTATIONS

We implemented our eventual consistency mechanism based on the original ZHT source code in C++. As the original ZHT code, Google Protocol Buffer [20] is used to transfer the messages over networks. We used Git [21] for source version control. Our project has been open sourced and it could be found on github [22].

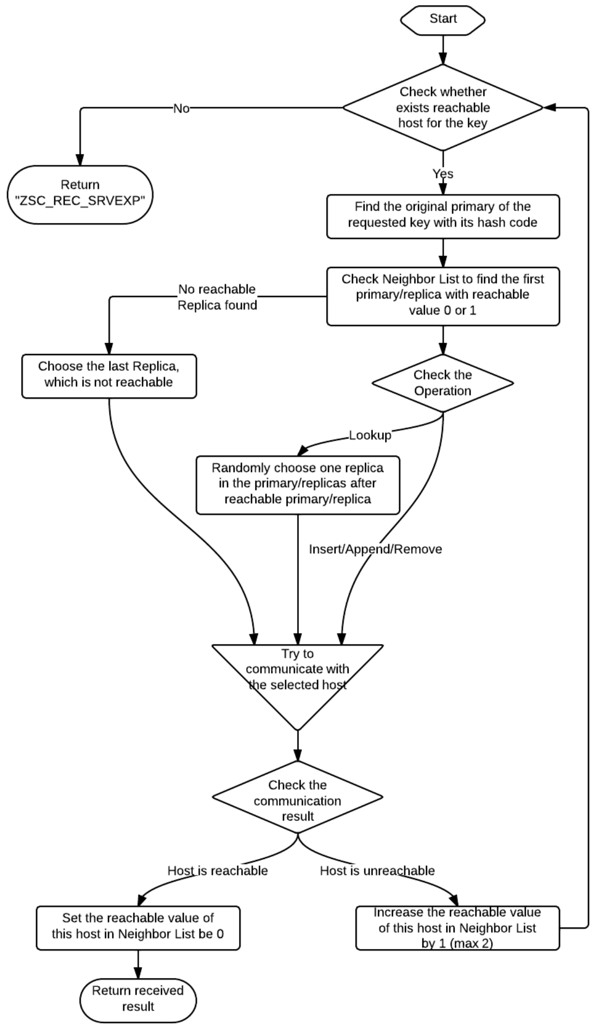

Start
Check whether exists reachable host for the key
No
Return "ZSC_REC_SRVEXP"
Yes
Find the original primary of the requested key with its hash code
Check Neighbor List to find the first primary/replica with reachable value 0 or 1
No reachable Replica found
Choose the last Replica, which is not reachable
Check the Operation
Lookup
Randomly choose one replica in the primary/replicas after reachable primary/replica
Insert/Append/Remove
Try to communicate with the selected host
Check the communication result
Host is reachable
Host is unreachable
Set the reachable value of this host in Neighbor List be 0
Increase the reachable value of this host in Neighbor List by 1 (max 2)
Return received result

Figure 11 Flow Chart of clients for sending a request with Primary Failure

4. PERFORMANCE EVALUATION

In this section, we describe the performance of eventual consistency model, latencies and throughput. We will introduce the testbeds and workloads first. Secondly, we present a comprehensive performance evaluation.

4.1 Testbeds, Metrics, and Workloads

The experiment environment mostly refers to the ZHT paper [1]. Our experiments are executed on kodiak cluster, a 1028-node cluster from the Parallel Reconfigurable Observational Environment (PROBE) at Los Alamos National Laboratory [13]. Each node in this cluster is setup with a two 64-bit AMD Opteron processors and 8GB RAM. We start one client and three servers (one primary and two replicas) on each node. Every client creates a long list of key-value pairs and sequentially sends all the key-value pairs for insert, then lookup, then append and then remove. Each client sends 10k requests for each of these four operations to the servers. The clients and servers would be launched by parallel ssh, which can reduce the latency to start the benchmark. The system has been tested scale up to 256 physical nodes with one client, one primary and two replicas on each. We compared our eventual consistency model with the original ZHT system (without replicas), strong consistency model, laziness consistency model and unreliable eventual consistency model.

The metrics we evaluate is as follows,

Latency: The time taken for a request to be submitted from a client and a response to be received by the client, measured in milliseconds.

Throughput: The number of requests a system can handle over a certain time, measured in ktasks per second.

Scale: We will run our experiments up to 256 nodes.

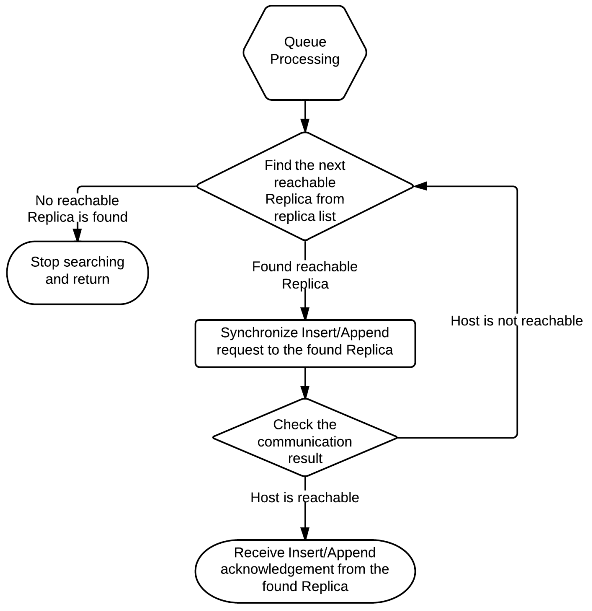

Queue Processing
Find the next reachable Replica from replica list
No reachable Replica is found
Stop searching and return
Found reachable Replica
Synchronize Insert/Append request to the found Replica
Host is not reachable
Check the communication result
Host is reachable
Receive Insert/Append acknowledgement from the found Replica

Figure 12 Flow Chart of Queue Processing with Primary Failure

4.2 Latencies

The latency has been used here to measure how long it needs to be taken for a task, averagely.

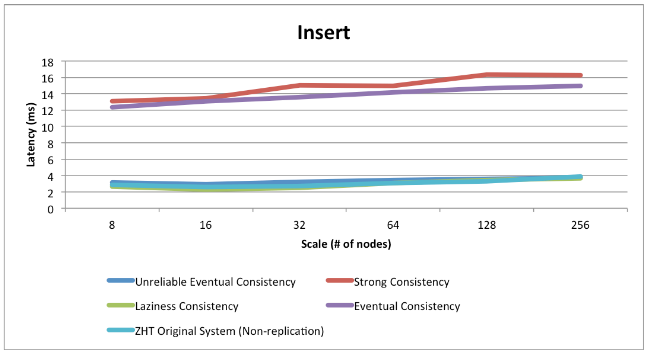


Figure 13 Latency Performance for Insert Operation

Figure 13 illustrates the latency of insert operations with different number of nodes. We can conclude that The latencies of strong consistency and eventual consistency are slower than that of ZHT original system, laziness consistency and unreliable eventual consistency on insert operations up to 344%. The reason is that strong consistency and eventual consistency are required to wait for insert on primary and replica; however, the client would receive the result after the insert on primary only with ZHT original system, laziness consistency and unreliable eventual consistency. Strong consistency has the slowest latency due to that the client can receive the result after insertion on both replicas; however; the client can receive the result after insertion just on the first replica with eventual consistency.

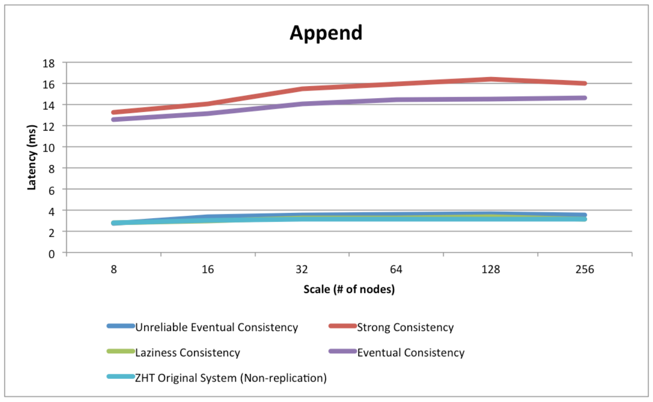


Figure 14 Latency Performance for Append Operation

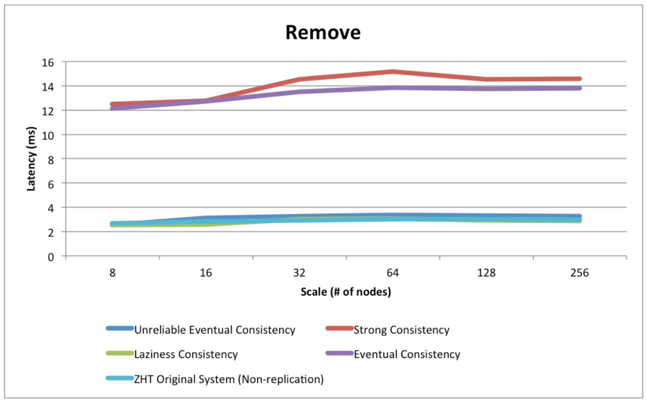


Figure 15 Latency Performance for Remove Operation

Figure 14 and Figure 15 illustrate the latency of append and remove operations with different number of nodes. We can conclude that ZHT original system, laziness consistency model and unreliable eventual consistency have lower latency than that of strong consistency and eventual consistency on both append and remove operations.

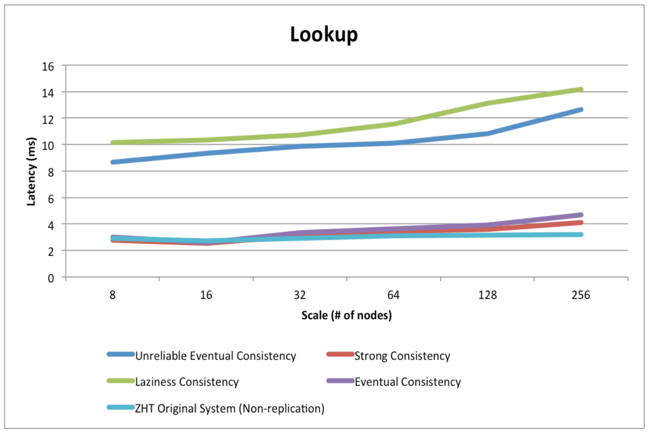


Figure 16 Latency Performance for Lookup request

Figure 16 illustrates the latency of lookup operations with various numbers of nodes. We can conclude that ZHT original system, strong consistency and eventual consistency model have a lower latency than that of unreliable eventual consistency and laziness consistency up to 342%. The reason is as follows. Since the lookup operation is served on primary only with ZHT original system and strong consistency, no more network communication would be issued after the local lookup operation on primary. However, since the replica could also be used to serve lookup operations in all the eventual consistency models, the key-value pair on replicas has to be compared with that on primary server, which results in more network communications. The more network communications could result in higher latency in all the three eventual consistency models. In the three eventual consistency models, the latency of eventual consistency is slower than that of laziness consistency and unreliable eventual consistency. The reason is that the strong consistency has been maintain between primary and first replica in eventual consistency, which results in that the number of version conflict would be smaller than that in laziness consistency and unreliable consistency. Even the key-value still need to be compared with primary server once the first replica received the lookup request, the smaller number of version conflicts happen, the fewer data would be sent back to the first replica, which results in lower latency.

The latency of lookup operations would not change a lot with the increasing of the number of nodes on ZHT original system, strong consistency and eventual consistency model. . It shows that the laziness consistency and unreliable eventual consistency model does not scale well due to extra data would be transferred when version conflict happens.

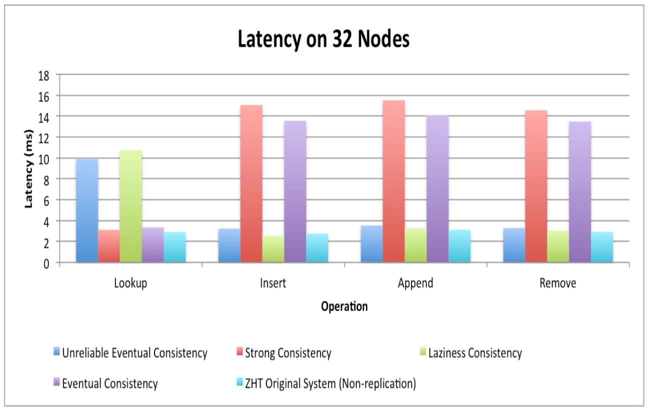


Figure 17 Latency of 4 operations on 32 nodes

Figure 17 illustrates the latency of insert, append, remove and lookup operations on 32 nodes. We can conclude that eventual consistency has a lower latency than laziness consistency and unreliable eventual consistency on lookup operation. The latency of eventual consistency is lower than strong consistency on insert, append and remove operations and still maintain the reliability.

4.3 Throughput

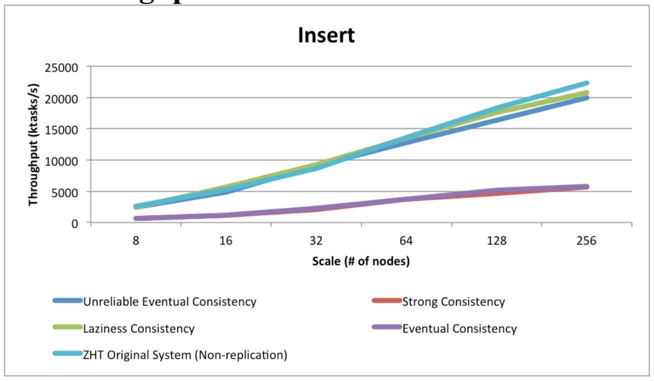


Figure 18 Throughput Performance of Insert Operation

Figure 18 illustrates the throughput of insert operations with various numbers of nodes. The throughput can increase near-linearly with scale. The ZHT original system, laziness consistency and unreliable eventual consistency have a higher throughput than that of strong consistency and eventual consistency on insert operation up to 293%. The reason is that client could receive the result after update on primary only with ZHT original system, laziness consistency and unreliable eventual consistency.

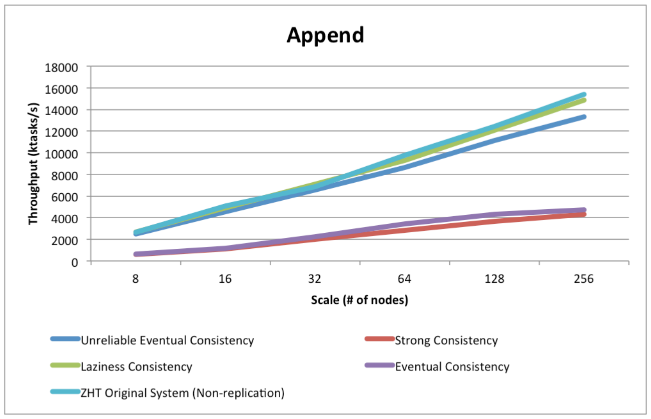


Figure 19 Throughput Performance of Append Operation

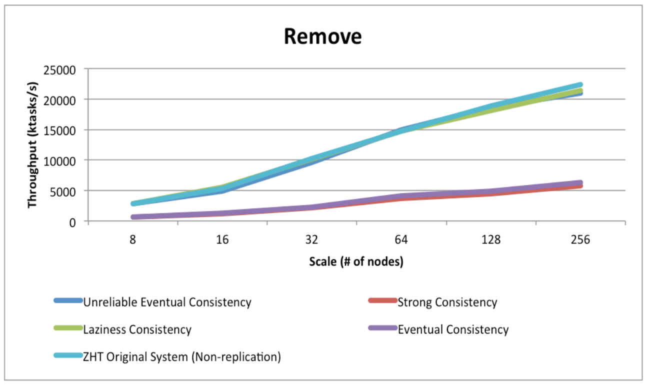


Figure 20 Throughput Performance of Remove Operation

Figure 19 and Figure 20 illustrate the throughput of append and remove operations with different number of nodes. We can conclude that strong consistency and eventual consistency have a lower throughput than that of ZHT original system, laziness consistency and eventual consistency on append and remove operations. The reason is the same as that in insert operation.

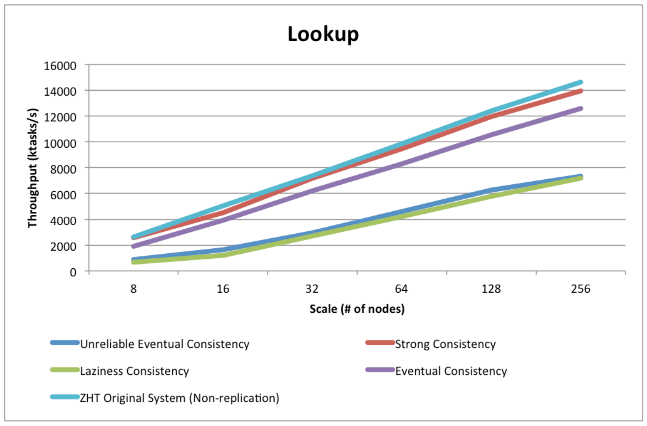


Figure 21 Throughput Performance of Lookup Operation

Figure 21 illustrates the throughput of lookup operations with different number of nodes. the throughput of ZHT original system, strong consistency and eventual consistency model is higher than that of laziness consistency and unreliable eventual consistency. Since the replica could not serve lookup request on ZHT original system, strong consistency and eventual consistency, the number of network communications would be lower than that of laziness consistency and unreliable eventual consistency. And since the primary and the first replica have been updated in insert, append and remove requests with eventual consistency, the number of version conflict in eventual consistency would be lower than that of laziness consistency and unreliable eventual consistency. That is the reason why the throughput of eventual consistency is higher than that of laziness consistency and unreliable eventual consistency in lookup operation.

So, our eventual consistency can provide a lower latency, higher throughput than strong consistency on write operations and provide a lower latency, higher throughput than laziness consistency and unreliable eventual consistency in lookup operation, while we can maintain the consistency between primary and replicas to enhance the fault tolerance on ZHT.

5. RELATED WORK

A lot of distributed system used replication for fault tolerance and the consistency strategy they used is various.

Harp[5] is one of the first distributed system applied primary copy scheme. For consistency, it implemented strong consistency. Each requests will be replied to clients only after primary servers received replies from other backup servers (i.e. replicas). For failures, Harp designed a view change algorithm for failover.

Similar as ZHT, Dynamo [6] is a key-value storage system to provide high availability that are severing some Amazon's core services. Dynamo also applies eventual consistency to allow replica updating asynchronously with data visioning, which is very similar as our project. The different part is Dynamo applies consistent hashing [14]. With consistent hashing, both of nodes and data items has its position on the ring (the output range of hashing). When a data item from a request find its position, it will walk in the ring in clockwise and the first node with larger index than the data item will server this request. This method provides advantage on enabling departure or arrival of a node, because each node is only responsible for the data between it and its predecessor and therefore departure or arrival of a node only affect its immediate neighbors.

6. CONCLUSION

From this project, we had a chance to work a project with relative difficulty and heavy workload, from which we learnt a lot about distributed system.

How to implement a distributed system: ZHT is a real key-value pair distributed storage system, which is under development. We need to first understand the workflow and code of ZHT and then work with it to do our own implementation. This procedure helped us to deeply learning and thinking about the implementation of ZHT, like how the system is layered, how to make communication decision, how to avoid server crash, etc. Also, through the experiments following implementation, we learned how to design experiments and what kind of metrics is useful to evaluate a system.

Consistency models: when we implementing eventual consistency, we don't simply get idea from the mentor and just code. We learn about and compare the differences between consistency models and how they are used in others' research or projects from various materials. To help ourselves learn better, we decoupling the design of eventual consistency and execute experiments on different edition of eventual consistency, like laziness eventual consistency which only used version compare, and unreliable eventual consistency, which ask primary updating secondary replica without guarantee the strong consistency between primary and secondary replica. And we also implemented strong consistency, another common used consistency model, for comparison experiments. All these implementation and comparison helped us to learn about how the designs of consistency models affect latency, throughput and reliability.

System scale: Another important stuff we learnt from this project is how to solve the scalability problem. At first, the scale of experiment can only reach 8 node. Be modifying our design to reduce the thread created for task processing, finally we increased the test scale from initially 8 nodes to 256 nodes.

As a summary, we implemented the replication scheme, eventual consistency support and executed experiments on up to 256 nodes, which has already satisfied our proposal and reached our goal. However, we do have space to make more achievements, like a completed delivery of failure handling, which we considered as functionality to our project in late semester.

For future work, we have some thoughts on following two aspects,

Maintain certain level of Replication: In our project, we consider the situation of single node failure, which should be able handled by our solution. However, this actually reduces the replication level, which may cause data loss if more node failure happens. So, if we will keep working on this project, we will works on how to enable a server to initialize new replica to maintain the original level of replication for each data item during the system is running.

Server recovery: Most clusters have the server recovery scheme, like Harp mentioned above. When a failed server is recovered, it can play its old role before shutdown and recover the lost data, or it can be seen as new arrival server. For the former situation, some distributed system maintains a log file stored in disk to record the activity of a server, which can be used to retrieve the data. For the latter situation, this may be counted as node join and departure problem and need relevant solution.

Replicas server both read and write operations: We can improve our design to make replicas also can serve the read and write requests. In this case, Paxos [16] [17] need to be implemented to resolve the leader selection problem. Chubby [15] and Zookeeper [18] could be used for the implementation.

Support Frequently nodes join and departure: When the nodes would join and departure frequently, it would be hard to maintain the membership. We can provide a solution to resolve it as the design in Scatter [19].

7. REFERENCES

[1]. Lindsay, B.G., et. al., "Notes on Distributed Databases", Research Report RJ2571(33471), IBM Research, July 1979

[2]. Gray, J., Helland, P., O'Neil, P., and Shasha, D. 1996. The dangers of replication and a solution. In Proceedings of the 1996 ACM SIGMOD international Conference on Management of Data (Montreal, Quebec, Canada, June 04 - 06, 1996). J. Widom, Ed. SIGMOD '96. ACM Press, New Y ork, NY , 173-182.

[3]. T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, I. Raicu, "ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table", IEEE IPDPS 2013.

[4]. "De-mystifying "Eventual Consistency" in Distributed Systems", Oracle, NoSQL Database, June 2012.

[5]. Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In 13th Symposium on Operating System Principles, pages 226–238, Pacific Grove, CA, October 1991.

[6]. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshaman, A. Pilchin, S. Sivasubramanian, P. Vosshall, "Dynamo: Amazon's Highly Available Key-value Store", SOSP' 07, October 14-17, 2007

[7]. W. Vogels (Amazon.com), "Eventual Consistency".

[8]. S. Cribbs, "Data Structures in Riak (NoSQL Matters Cologne 2013", April 26, 2013

[9]. http://guide.couchdb.org/draft/consistency.html#cap

[10]. http://blog.mongodb.org/post/498145601/on-distributed-consistency-part-2-some-eventual

[11]. J.M. Wozniak, B. Jacobs, R. Latham, S. Lang, S. W. Son, and R. Ross, "C-MPI: A DHT implementation for grid and HPC environments", Preprint ANL/MCS-P1746-0410, 2010

[12]. S. Burckhardt, A. Gotsman, H. Yang, "Understanding Eventual Consistency", MSR-TR-2013-39, March 25, 2013.

[13]. G. Grider. "Parallel Reconfigurable Observational Environment (PRObE)," available from http://www.nmc-probe.org, October 2012.

[14]. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In Proceedings of the Twenty-Ninth Annual ACM Symposium on theory of Computing (El Paso, Texas, United States, May 04 - 06, 1997). STOC '97. ACM Press, New York, NY, 654-663.

[15]. Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In Proceeding OSDI '06 Proceedings of the 7th symposium on Operating systems design and implementation. Pages 335-350.

[16]. Lamport, Leslie.Paxos Made Simple. ACM SIGACT News Distributed Computing Column 32, 4 Whole Number 121, December 2001 51-58.

[17]. Lamport, Leslie. The part-time parliament. ACM Transactions on Computer Systems (TOCS) Volume 16 Issue 2, May 1998. Pages 133 - 169.

[18]. Patrick Hunt and Mahadev Konar and Flavio P. Junqueira and Reed Benjamin. ZooKeeper: wait-free coordination for internet-scale systems. Proceedings of the 2010 USENIX conference on USENIX annual technical conference, 2010.

[19]. Lisa Glendenning and Ivan Beschastnikh and Arvind Krishnamurthy and Thomas Anderson. Scalable consistency in Scatter. SOSP '11.

[20]. https://developers.google.com/protocol-buffers/.

[21]. http://git-scm.com/.

[22]. https://github.com/skxie/zht-eventual-consistency

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