ZombieKV: Scalable, Replicated and Cached Key-value Store using Consistent Hashing

Sen Yu

Abstract: This paper introduces ZombieKV, a scalable key value store with a replication factor of 3. ZombieKV stores data persistently on disk with the servers oriented in a hash ring topology, while maintaining an in memory cache that supports eviction strategies such as FIFO, LRU and LFU. It supports scaling of servers without service downtime, automatic failure recovery of servers, subscriptions and malicious server detection and recovery.

1. Introduction

Dynamo [1] is one of the few key value stores we studied in class. ZombieKV is deeply inspired by the Dynamo ring the authors of the Dynamo paper proposed at the beginning.

The key highlights of ZombieKV are:

- Scalable: The key-value store service need not to shut down for configuration changes such as add or remove nodes from the service
- Replicated: With a replication factor of 3, 3 specific servers need to fail concurrently for data to be lost
- Fast writes and eventual consistency: Writes are successful as soons as the primary server for that key (coordinator) writes it to disk. The coordinator then lazily tells its replicas about the write
- Cached: Not only is the data stored durable on disk, there is also a small cache in memory for fast reads
- Simple key-value store semantics: There are only 2 operations available to the client during normal operation, PUT and GET
- Failure detection and auto recovery: The servers exchange heartbeats to detect failures and the service automatically brings up backups to maintain the number of servers in the service
- Malicious server detection and replacement: Verification processes are put in place to check the validity of PUTs and GETs. If a server is behaving maliciously then it will be automatically be replaced
- Subscriptions: Clients can subscribe to any key and be notified when the value associated with that key changes

2. System Architecture

In this section, we will go over some key design decisions I made when developing ZombieKV.

2.1 Topology

The topology of ZombieKV is depicted in Figure 2 of the Dynamo [1] paper. Each server node (server) on the hash ring runs on its own machine. Every node maintains some metadata about the service. These metadata include the node's own IP address and port, and other node's IP address and port. The hash range each server is responsible for is computed by taking the MD5 hash or the node's own IP_address:port_number which results in a number to place the server at a certain position on the hash ring. The 2 successors of any server also replicates all the data in that server.

2.2 Communication Protocol

There are three roles (machines) in ZombieKV: client, server and admin. Communication is only possible between client/server, server/server and server/admin. There is no direct connection between the client and the admin, neither is there any scenario that warrants that kind of connection.

There are three cases which there are client/server communications. First, the client can connect to any single server it knows about to begin normal operation such as PUT and GET. If the key the client is operating on does not fall into the hash range the server being contacted is handling (i.e. not the coordinator), the server will return a message with status code SERVER_NOT_RESPONSIBLE along with some metadata that describes the current topology, i.e. which servers are responsible for which hash ranges. Then, the client can contact the appropriate server for operations. Another case of client/server communication is for malicious server detection. There will be detailed implementation of this feature in section 3.6. On a high level, during a PUT request, the 2 replication servers will connect to the client and ask if the clint indeed initiated such PUT request. During a GET request, the client will connect to the 2 replication servers and verify the key value pair it received from the coordinator is correct. The last case for client/server communication is for subscriptions. This is also discussed in depth in section 3.7. In short, the client tells a server the key it want to subscribe to and the port number it would like to receive notifications on. Whenever there is a change to the value of the key the client is subscribed to, the coordinator of that key will push a notification to the client at the designated port.

The communication between server/server is mainly for replication. When the coordinator receives a PUT request from a client. It immediately returns to the client with status code PUT_SUCCESS and in the background initiate replication requests to the 2 replicas that are successors to the coordinator. This is called eventually consistency as we studied in class. In addition, each server pings (heartbeats) its successor every 5s to make sure it is alive. If not, it will notify the admin about this situation and the admin will take appropriate actions. Lastly, the admin may instruct servers to transfer data to other servers. This is usually due to configuration changes such as adding and removing servers.

The admin (a machine) communicate with the servers to exert control over the service. During service startup, the admin ssh into the server machines and spin up a JVM to start the server application. The admin also gives the START and STOP

message to servers to allow and disallow them to communicate with clients. When adding a server or removing a server from the service, the admin has logic to compute which data needs to be moved and instructs the servers that are involved in data movement to 1) Lock for any writes 2) Instructs the sending server to send the data over to the receiving server 3) Update the hash range for servers that are affected (only local to a few servers) 4) instruct the new server to START if adding a new server or to SHUT_DOWN if removing a server. Another form of control the admin exerts to the servers is through failure detection and auto recovery. As noted in the server/server communication discussion, when the admin is notified from a server that another server is dead, the admin will first PIN the presumed dead server to check if it indeed failed. Then the admin will launch a "recovery procedure" by doing 1) Identify a idle server listed in admin.conf 2) Start it up by ssh into the machine and launch the server application 3) Identify the data the new server should manage, locate them in the replicas that are online, and instruct the replicas to send the data over to the new server 4) Update affected servers with new metadata. Note, the admin referred in this paragraph is the machine that is running the admin code. The administrator (a person) has access to an admin client that only has high level commands such as initService, start, stop, shutDown, addNode and removeNode. All the detailed logic and instruction to servers are handled by the admin machine (code).

3. Implementation

3.1 Message Encoding

Messages are encoded in JSON and there are four fields. For example {"statusType": ADMIN_START, "arg1": null, "arg2": null, "arg3": null} is a valid message in the the service. The statusType dictates what kind of message this is. For example, when the statusType starts with ADMIN, it means that these are coming from the AdminClient instead of a normal client. If the statusType is ADMIN_UPDATE, then it is expected that arg3 would be the new metadata the server should be saving.

3.2 Socket Programming and Threading

On the client side, there is only one active connection with a server at any given time. A Java TreeMap is used to speed up lookup of the server responsible for a given hashed key. Each client will connect through their own instantiation of Store (a Java Class in code), which acts as a client library. If a new storage server connection needs to be established, for example upon receiving SERVER_NOT_RESPONSIBLE, the Store will replace current socket with a new socket. This way, I was able to keep the memory usage at a minimum at the expense of increased latency in initializing sockets and input/output streams. I also decided to use TreeMap to keep track of the server hashes, so that given a new hashed key we can perform a fast O(logn) lookup of the matching storage server.

On the server side, there is a main thread called WelcomeThread that listens to incoming connections. Both the client and the admin (machine) can connect to the server. I created a new thread for each client or admin connection to the server. All of the threads share a couple of static volatile variables from the WelcomeThread which are used for the admin activities. To switch off the server's ability to accept client requests, I used a shared lock among the main WelcomeThread and all the other client connection threads to prevent race conditions. It works as follows: to turn off the client requests, we need to first make sure that there are no ongoing client request processing. To ensure that, we first close down the isOpenToClientRequests volatile variable, then we try to acquire the shared lock before we actually declare that we have successfully closed the client requests.

The admin communicates with the server in a similar way as a normal client but with different statusType field in KVMessage. There is a full set of "API"s in AdminStore class that is used by the admin for communicating with the server as a library just like the Store class for the client. Admin (code) runs locally on the administrator's (human) machine and gives instruction to individual servers. The admin (machine) maintains the state of the distributed system. The admin is able to take in 6 different high-level request from the administrator, For example, addNode, and make calculations as to where the new node should be placed on the hash ring. After that it computes what each node has to do, for example, a node might need to transfer data to another. Finally, the admin send all the instructions in the form of ADMIN messages to the servers.

3.2 ZombieKV: Failure Detection and Auto Recovery

In ZombieKV, each server establishes a connection with its successor in the ring topology and send "heartbeat" messages every five seconds. If its successor does not reply to the "heartbeat" request message in five seconds, then the server that initiates the "heartbeat" request will conclude that its successor is dead and it will then establish a connection with the admin to inform it with the dead server's serverId (IPAddress + PortNumber). After admin receives the message, it will pin the dead server to confirm that it is dead. After it confirms the server is indeed dead, it will perform a removeNode operation on the dead server. After the removeNode operation is complete, it will perform an addNode operation from a pool of idle servers to add back to cover the loss of the dead server. Note that the removeNode and addNode operation are not the same as when an administrator types them in by hand, the detail of that design and implementation is discussed later in 3.4 System Reconciliation.

The "heartbeat" mechanism among the servers described above was invoked when admin issues the initService command. (i.e. Each server will establish a connection with its successor). After that, whenever there is a new metadata update which contains the new arrangement of server nodes, the "heartbeat" mechanism will get invoked but only the servers with a new successor will tear down its old connection and establish a new

connection with its new successor. This design will ensure that the number of connections between the servers is the same as the number of nodes. It can handle multiple node failure despite that there is only one connection between each pair of servers. For example, if there are 5 nodes (A, B, C, D, E) in our distributed system, and A sends heartbeat to B, B to C, C to D, D to E and E to A. If node A and node B both go down at the same time, although that at the moment A can't inform admin that B is down because itself is down, A will get detected by E to be dead and E will inform admin about it. Admin will then handle the death of A by performing removeNode operation first and update the metaData. Once the metaData is received by each server (metadata now only contains B, C, D, E because A has been removed), new heartbeat connection will get established between E and B due to that E's successor used to be A and now should be B based on the new metadata. So E will try to establish a connection with B, but since B is dead, the connection will not get established and E will inform admin that B is dead, which means that both the failure of A and B can be detected by our failure detection design. Note that there are cases where there are multiple messages with the same dead server to admin due to that the reassignment of connections between servers is performed after each removeNode and each addNode operation. To resolve the problem of removeNode and addNode the same server multiple times in this case, admin will perform a checkup/pin on the dead server to determine whether it will perform the operations to recover failure. If admin gets a response from the dead server, it means that admin has already recovered it due to a previous message.

3.3 Replication Mechanisms

In order to replicate, each server will determine its two replicas from the metadata. Each time a write operation is received by the server, it will write to its own persistent disk first and then establish connections to its two replicas and send a special server to server put request (which is a different request than the client to server put request that will bypass the write hash range check). This operation is non blocking which means that the data on the replicas may be stale but eventual consistency is guaranteed. Write requests can only be served by the coordinator for a given data item while read requests can also be served by the two replicas.

3.4 System Reconciliation

System Reconciliation will take place with every addNode and removeNode call by the administrator. The reconciliation is handled in the Metadata Java Class' addNewServer and removeServer methods. These methods will make adjustments to the inner treemap structure of metadata and return back a *map* in which the keys are server names and values are hashRanges of which data is needed to be added or removed from the server's key range. For adding new nodes, the resulting *map* will contain the next three successors of the new node and the hash ranges of the data need to be moved from the successors onto the new node. If failure is encountered when moving data, admin will pick other replicas that has the data and copy the data from that replica onto the newly added node. For removing nodes, the resulting *map* will contain the next three successors of the removed node and the hash ranges of the new data needed by these successors. Admin will then find servers that have copies of the data corresponding to the hash range and attempt to move data from these servers to the needed server until success.

Moving data is achieved by transferring key value pairs within the range given by the admin from the source server's persistent storage to the target server's persistent storage one by one over the network. First, the admin sends a ADMIN_MOVEDATA message to the server that contains the data to be transferred, while specifying the destination server and the range of interest. The source server, after receiving the message, will establish a connection with the target server and use the ADMIN_FILETRANSFER message status type to let the target server know that it is not a put request from client but a file transfer request so that it will bypass the target server's own hash range check and directly go to the persistent file. Once the target server finishes, it sends a message to the source server (ADMIN_FILETRANSFER_COMPLETE) and the source server will send the admin a response message to let it know that the file transfer process has completed.

3.5 Client Handling of Dead Servers

The client (machine) will keep a version of the metadata similar to the one admin has in memory. When a client encounters SERVER_NOT_RESPONSIBLE error and with it new metadata from the server, it will update its local version of metadata of the current service topology. If during put and get calls, the client encounters a dead server, a Socket Exception will be thrown and caught by the client which triggers it to remove the dead server from its metadata, and a new call will be initiated. Since each put/get request will perform a lookup for the server to send requests to based on the metadata, the dead server can no longer be found. The dead server's successor will most likely be found and the request will be directed to the new server. The same handling will be issued until there are zero servers left in the metadata, in which case the client application will returned an error to the user (human).

3.6 Detection and Recovery of Compromised Servers

The detection of compromised servers feature involves two types of detection, the detection of "compromised get" and the detection of "compromised put". A "compromised get" is when a compromised server attempts to return a wrong value to a client's get request. A "compromised put" is when a compromised server attempts to update key-value pairs (replication process) on other servers that were not initiated by the client or initiated by the client but with the wrong value. The following is the detection and recovery of compromised

get and detection and recovery of compromised put protocol summary.

Detection and Recovery of Compromised Get Protocol Procedures:

1. Client initiates a GET request to Server A,
2. Server A will reply to the Client with a key-value pair
3. Once the Client has received the key-value pair from Server A, it will silently (non-blocking background operations) send the Key,Value pair to the other two replicas(call them Server B and C) that also know about this key-value pair
4. Server B and Server C, once received the key-value pair from the client, will validate the key-value pair using their own persistent storage. If the value is wrong, or the pair does not exist, Server B and Server C will notify the client and admin.
5. Once admin receives both messages from Server B and Server C that Server A is compromised, which means the quorum(majority) is reached, it will take action against the Compromised Server A by taking it down and deleting its data and then bring it back up with the data from its replicas.
6. Once the client receives both messages from Server B and Server C that Server A is compromised and with the actual read value (quorum is needed), it will notify the user (human) in the UI that the previous value that the user got was incorrect and what the actual read value should be.

Detection and Recovery of Compromised Put Protocol Procedures:

1. Client initiates a PUT request to Server A. Each of our client application has a server socket that is used for the cases where servers need to contact the client for validations of PUT request.
2. Server A will reply to the Client with a PUT_SUCCESS message
3. Once client receives the reply from Server A that the PUT request is successful, it will store the put request in its action log
4. Server A will proceed to send Server B and Server C (its replicas) the client PUT request. In our protocol, each replication not only needs the key-value pair but also the origin of the request (which client is the request from).
5. Server B and Server C will attempt to connect to the client that is provided in the replication message and send the key-value pair along with Server A's name(origin of replication process) to the client for confirmation
6. Once the client receives the messages from Server B and Server C, it will compare the key-value pair with its action log mentioned in step 3. If such key-value pair by Server A put operation is not found in the action log, or the value is wrong, it will message back Server B and Server C that the client has never requested such put operations. And If such key value pair put operation indeed exist and is by Server A, then it will just silently ignore the message.
7. If Server B and Server C receives messages from the client that Server A's put operation was wrong, they will tell admin that Server A is compromised.
8. Admin will take action against Server A by shutting it down and replacing its content with its replicas' data and bringing it back up.

If the reader wants to try this functionality out. I have specifically encoded any server running on port 60008 to be a compromised GET server and port 60009 to be a compromised PUT server for demonstration purposes.

3.7 Subscription and Notifications

ZombieKV allows the user to subscribe and unsubscribe to data mutations. To use it, the user may enter the commands "subscribe" and "unsubscribe" followed with the key of interest. First, the client opens a client side ServerSocket (listening socket) to listen for incoming notification in the future. During a subscription, the client (machine) will find out which server is the coordinator for the key of interest by exchanging messages with the servers it knows and update its metadata if necessary. Then, the client will send the subscription request which contains the key and the IP address & port of its ServerSocket to the coordinator server. The coordinator server will then save the request in memory. In the future, when this server receives a new put request or put update to this key, it will lookup the ServerSocket that was subscribed to this key (the client) and send a notification out to the client.

However, the subscription is lost if the coordinator fails. I solved this challenge by making the observation that we already have a mechanism in place for propagating information around in the storage service - heartbeats. We can let the subscription information piggyback on the heartbeat messages that are already sent around the hash ring. When a server receives subscription information from its neighbour. It will compare it with the local subscription information and merge the information together. For example, if server A has subscription info "key1:[127.0.0.1:6000]" and server B has subscription info "key1:[127.0.0.1:60001], key2:[127.0.0.1:60000]" Then when one server send its subscription information to another, the receiver will update its local subscription information to "key1:[127.0.0.1:6000, 127.0.0.1:60001], key2:[127.0.0.1:60000]". By doing this, if the server that initially received the subscription request does not immediately crash before sending out any heartbeat, then it's the subscription will live as long as the service lives, even when new nodes are brought up by admin, it will be fed the subscription information that is being passed along the hash ring.

However, this had a side effect of subscriptions never being able to be deleted by contacting a single server alone. If a subscription is deleted from a server, it will be simply added by in the next heartbeat. This problem can only be tackled by admin as it is one who knows about all the servers that are active and only it can force the heartbeats to stop and resume. Thus, I implemented unsubscription by having the client library send the unsubscribe request to any server it knows in the storage service. Then, the receiving server will tell admin about the request. The admin will send a message to all active servers to temporarily halt all heartbeats. Then it will broadcast another message to let the servers remove the subscription in question. Finally, it will broadcast to resume the heartbeat process.

4. Performance

Performance test methodology: I first populated the key store with 1,000 entries of made up key-value pairs. Then I vary one variable at a time while hold other variables constant to test the performance of the storage service under different configurations. The variable that is varied is the one in each section title. The other constant variables are configured as follows: 20 clients, 10 servers, 500 cache size and LFU cache strategy. The numbers are chosen because they are the median value for each corresponding experiment. For 4.1, 4.2 and 4.5, I first record a baseline measurement for a replication factor of 1, i.e. no replication. Then, I do the experiment again using a replication factor of 3 and compare the results.

4.1 Number of Clients vs. Latency

Table 1. Performance Test Result of Varying Number of Clients. Replication Factor = 1. Unit: ms/request

	1	5	20	50	100
Read	8.284	14.47	33.512	132.34	182.52
Write	9.35	30.138	80.97	254.42	357.9

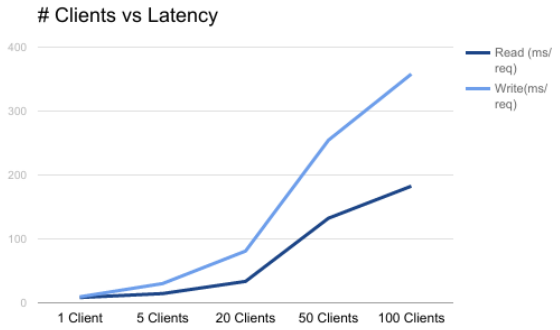

Figure 1. Number of Clients vs. Latency. Replication = 1

From Figure 1, we can see that as the number of clients increase the latency increases drastically as well. This is because as the number of client increases, each server has to handle more requests concurrently, resulting in more congestion and latency for each request. Furthermore, in general the write requests take longer than read requests. This is because write requests require that the server check the entire key store for duplication whereas read only need to find the first key match.

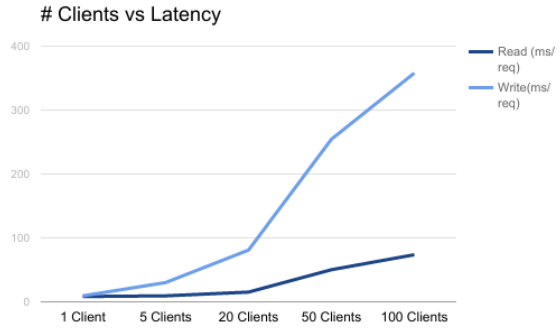

Figure 2. Number of Clients vs. Latency. Replication = 3

Compared to the result when replication factor = 1, this time the read speed decreased around 1/2 at high client counts whereas the write speed maintained the same. This is because during writes, the client only have 1 choice of server as before. But during reads, the client has 2 more choices so the read is distributed more evenly among the servers. In addition, in my implementation I used eventual consistency. An ack is sent to the client as soon as the coordinator has written the value to disk.

4.2 Number of Servers vs. Latency

Table 2. Performance Test Result of Varying Number of Servers. Replication Factor = 1. Unit: ms/request

	1	5	10	50	100
Read	52.79	42.77	32.85	25.23	23.79
Write	101.18	90.66	81.13	72.30	70.71

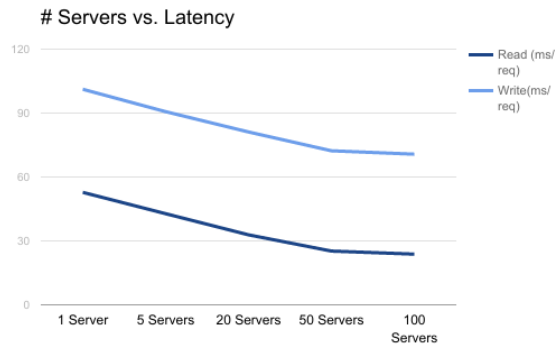

Figure 3. Number of Servers vs. Latency. Replication = 1

As shown in Figure 3, as the number of servers increase the latency decreases. However, the decrease in latency from 50 servers to 100 servers is not proportion to the increase in server count. This is because we only have 20 clients thus at this point pretty much every server is handling one request at a time.

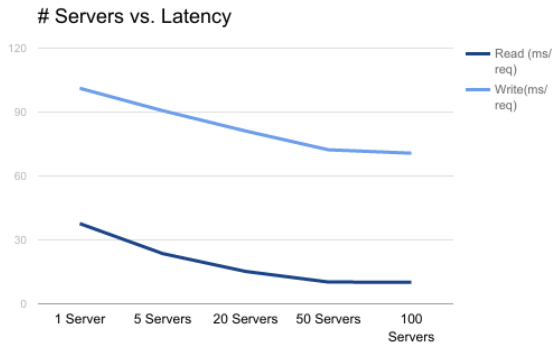

Figure 4. Number of Servers vs. Latency. Replication = 3

Compared to the result when the replication factor = 1, we see significant read performance gain due to the same reason as the previous test in section 4.1.

4.3 Cache Strategy Experiment

Table 3. Performance Test Result of Varying Cache Strategy. Unit: ms/request

	FIFO	LRU	LFU
Read	15.58	14.01	13.14
Write	81.65	72.85	66.52

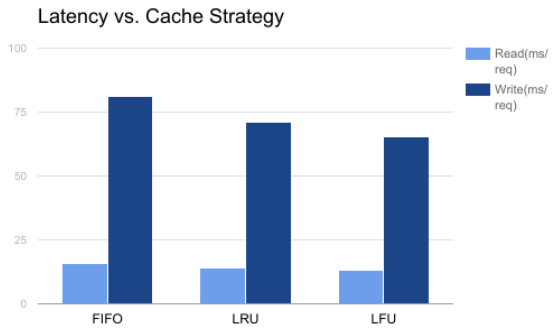

Figure 5. Cache Strategy vs. Latency

From Figure 5, we can conclude that LFU is the fastest cache. The reason LFU is the fastest is because I designed LFU myself with efficiency in mind. The FIFO and LRU were implemented using Java's built in LinkedHashMap library, which was not designed solely for caching.

4.4 Add and Remove Node Performance

Table 4 lists our add and remove node performance. Data is populated evenly across existing servers.

Table 4. Add/Remove Node Completion Time. Replication Factor = 1. Unit = Seconds.

# Servers	1	5	10	50	100
+1 Node	2.582	3.011	2.796	2.816	2.97
-1 Node	1.593	1.475	1.350	0.142	0.011
+5 Node	18.431	16.311	14.596	12.798	11.264
-5 Node	9.124	8.747	7.6	0.289	0.025
+10 Node	40.378	36.382	32.034	28.723	23.028
-10 Node	9.284	7.849	7.844	1.927	1.252

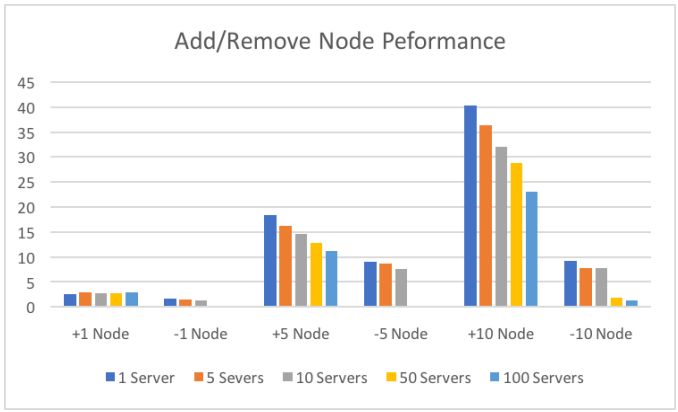

Figure 6. Add/Remove Node Completion Time. R = 1. (s)

From the results above, we can conclude 1) Adding node takes more time than removing a node. This is because ssh into the remote machine and wait for the application to boot up takes time, whereas removing a note need only terminate the remote process 2) As the node increases, the time it takes to add/remove node decreases, this is because less data is transferred between the nodes as each node is responsible for less data.

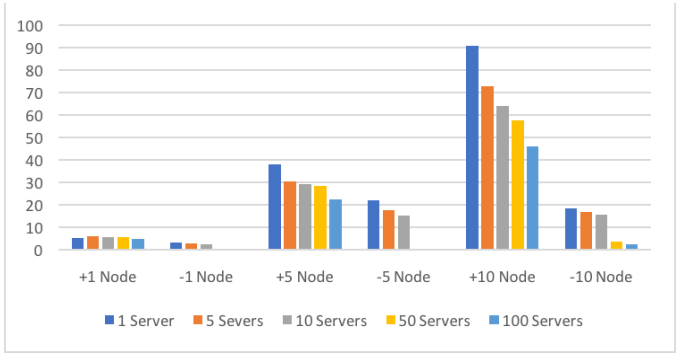

Figure 7. Add/Remove Node Completion Time. R = 3. (s)

Compared to the result when the replication factor = 1, the time it takes to add and remove and add node increased dramatically by around 2x. This is because of the introduction of replication. Now, whenever there is an add node or remove node operation, there are at least 4 nodes involved: the node in question and its 3 successors. Thus, more data is moved around hence more latency.

5. Conclusion and Future Direction

This paper described an implementation of a key-value store based on lessons learned from Stanford class CS244b. There are three weaknesses of my design that I am aware but did not have time to address. First, there is a single point of failure at the admin. If admin goes down, the service can still function during normal operation, but a lot of the features would be unavailable such as add/remove nodes, malicious server

detection and failure detection and auto recovery. Ideally, the admin should be highly available like Chubby or ZooKeeper. Second, when moving data, key-value pairs are sent one by one in separated messages. This is purely due to implementation easiness. A better solution here is to wrap all key-values to transfer into a single message. Lastly, in order to unsubscribe the admin has to send messages to all servers. A better solution is to pass unsubscribe messages around like subscription messages.

6. References

[1] 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 Proc. 21st ACM Symposium on Operating Systems Principles (SOSP), Oct. 2007.