DynamiteDB: A Dependable and Distributed Data Store

Tyler Lisowski, Yakshdeep Kaul, Ramon Sua, Satyajeet Gawas

Georgia Institute of Technology, Atlanta, GA
{tlisowski3,ykaul3,rsua3,sgawas3}@gatech.edu

1. Introduction

Team DynamiteDB designed an eventually consistent distributed key-value store database that allows clients to store and retrieve data tied to a unique key. The system has 3 replicas for a given key distributed to 3 database nodes. An overview of the system is shown in Figure 1.

The client sends his request to a load balancer, which then forwards the request to a master node. The master node is then responsible for finding the replicas that store the key. The master then queries the database nodes that store the replica to write the value specified by the client or return the newest value for the requested key to the client. The newest value is found by analyzing the responses of the database nodes. The client issues requests with the specified key in the message to initiate a read request. For a write request, the client specifies the key and value it wants to write. The master will then carry out the client's request and notify the client of the action taken by DynamiteDB. In order to ensure eventual consistency, an anti-entropy protocol is running periodically to synchronize replicas across database nodes. During the execution of the anti-entropy protocol, keys are exchanged between replicas to ensure they both have the same up-to-date copies.

2. System Architecture

There are 3 main components in our system: The client, master, and database nodes.

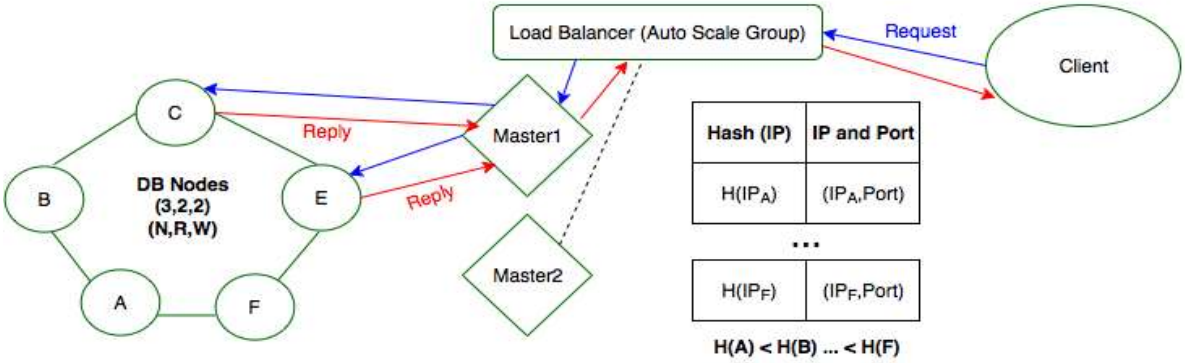

Figure 1. Workflow overview of DynamiteDB.

2.1 Client

The client is the user node of the systems that generates GET or PUT requests for key-value pairs. It interacts with the master node through a custom protocol that allows the client to retrieve the value of a key or write a key with a specific value. This is done by passing JSON messages with specific values set in the METHOD field (GET for reads, PUT for writes), with each request specifying the key that is being operated on.

2.2 Master

The master acts as a proxy between the client and the database nodes. The client requests are relayed to the master by an Elastic Load Balancer that serves the request to multiple master nodes in a round robin order. The master is a threaded implementation that spawns a new thread for each client request thereby ensuring scalability with respect to the number of client connections. We have successfully tested the master to handle 50 parallel client connections repeated in succession 50 times.

The master has access to a configuration file that stores all the necessary routing information to find the servers that are responsible for the requested key. This file stores the IP of each database node and the SHA-256 hash value of the IP. On startup, the master reads from this configuration file and stores the corresponding data in lists for faster accesses during IP lookups. Before carrying out any processing on the client's request, the master first executes its consistent hashing protocol which involves hashing the client's key followed by an IP lookup (from the lists previously described) on the hashed key to get the corresponding servers that are responsible for the corresponding key range. The requests from the client come in as a JSON object with METHOD (GET or PUT), KEY & VALUE fields. Since each request can be served by multiple masters, each master is responsible for ensuring that the request is updated with the latest timestamp by including (if it's the first request for that key) or updating the TIMESTAMP field which is later used for conflict resolution. Also, since for a read request, the master will have to examine the values returned by the database nodes and pick the latest one, the master includes (if it's the first request for that key) or updates the VECTOR CLOCK field in the request. The VECTOR CLOCK field is simply a dictionary of key-value pairs where the key is each master's IP and the value is a count which represents how many times a master node has acted upon a request corresponding to the key in the KEY field.

2.3 Database Node

The database node is responsible for storing the key-value pair, processing requests from the master, and updating replicas using the anti-entropy process. We use a file to store the value with the key as the file name. Upon receiving a PUT request, a KeyValueStore object is created from the received JSON string and this object is flushed to disk as an Object Stream. This allows the object to be recreated directly upon a read. The database node also runs a background task to execute the anti-entropy process. Each request is handled in a new thread.

3. System Operation

To model real life usage scenario, the clients generate random GET and PUT requests which are then processed by the master. The master redirects the request to the appropriate database node based upon the IP lookup as explained below. Our system implements a N=3, R=2, W=2 protocol. This means the client is required to write to or read from 2 out of the possible 3 database nodes storing a key in order for a given write or read operation to complete. These values for N, R, and W were chosen based on the data provided by Amazon in the Amazon Dynamo paper, which states it was a common configuration used for its distributed database implementation [1]. In case of a GET request, the master contacts two of

the three database nodes for the value. If one of them fails, it contacts the third database node responsible for that key range. In the scenario where two of the three database nodes fail, an error message is sent to the client. From the two responses received, the latest response is chosen by analyzing the VECTOR CLOCK field. A conflict might occur when the vector clocks returned don't have a causal relationship between them. Conflicts between the vector clocks are resolved by choosing the one with the largest (most recent) timestamp. This response is then forwarded to the client. Each PUT request is preceded by a GET requests as in the Amazon Dynamo paper [1]. This is done to get the latest value of the TIMESTAMP and VECTOR CLOCK associated with the key in the request. The master then appropriately updates the VECTOR CLOCK field and writes the value to two database nodes and waits for an acknowledgment from both nodes. If one of the acknowledgments is not received, the master writes the value to the third database node responsible for that key range. If two of the three acknowledgments are not received, an error message is sent to the client. This 2+1 methodology in reading from and writing to the database nodes was adopted to keep the number of TCP connections from the master as low as possible.

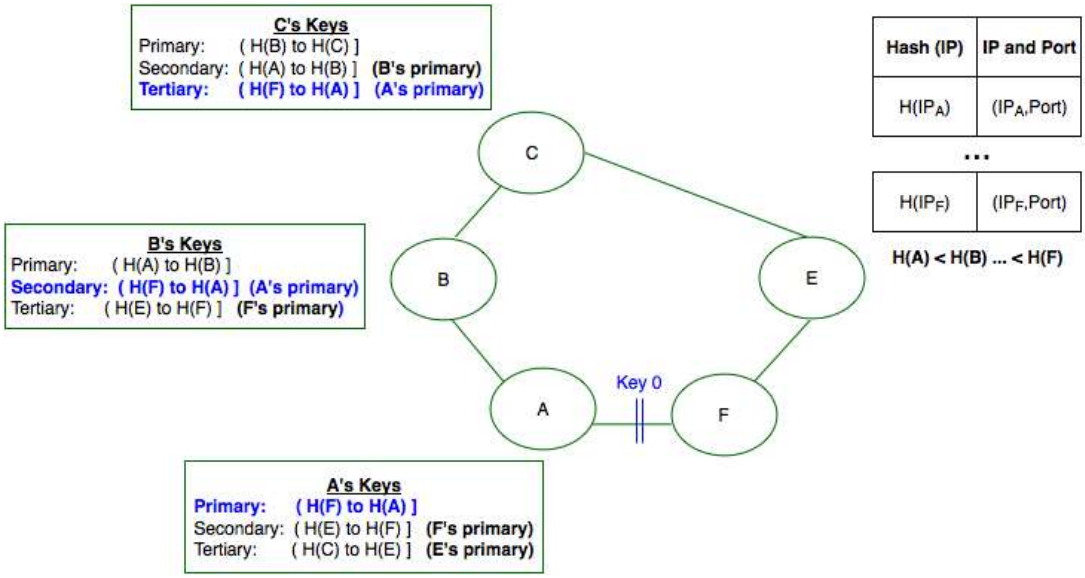

Figure 2. Key partitioning in DynamiteDB with consistent hashing and replication.

All data transmissions to and from the master are carried out with the help of TCP sockets. A timeout of 1 second is set on each connection with the database nodes to prevent each thread and hence the client from going into a deadlock in case of issues during communication with the database nodes. In addition to that, we ensured that database node failures were handled gracefully by the master. We tested this by manually rebooting two database nodes while the master was serving requests from the client, and we were able to verify that the correct error message was displayed at the client on a failed request with no cases of a deadlock.

Database nodes form the distributed database and store the key value pairs for the system. Each database node hashes its unique IP address and is assigned a key range based on the difference between it and its predecessor node. This process is known as consistent hashing [2]. For example, if the hash of Node A's IP is 1023 and the hash of Node B's IP is 1000 with a fixed hash length of 10 bits (0-1023 are

the possible key values), Node A will have a primary key set of 1001 through 1023 and Node B will have a primary key set of 0 through 1000. The distributed database consists of 5 nodes and the system uses the SHA-256 hashing algorithm to hash the keys and IP addresses. The primary key set for a given server is also be replicated across the two successors of that server, shown in Figure 2.

Requests come into the system as JSON objects. By reading the METHOD field, the database node can determine the type of message request: read, write, or anti-entropy sync. On a write, the system checks to see if a version of the key data exists on persistent storage. If no version exists on persistent storage, the key data (key, value, vector clock, and timestamp) is serialized to disk using the Java Serialization mechanism. The filename of the data is a hex encoded string of the key. This allows for easy retrieval of existing keys in the system. If a version exists on disk, the system checks to see which version is newer based on the vector clock values and serializes the newer data. If there is a conflict in the two versions' vector clocks, the system serializes the version with the latest timestamp. For all writes, the system returns the up-to-date version of the key data to the master as acknowledgement that the write has succeeded. For a read, the node just checks to see if a version of the key exists on persistent storage. It returns the version of the key data it has stored as a JSON object, or if no version is stored, it returns a JSON object with the METHOD field set as "NOVAL".

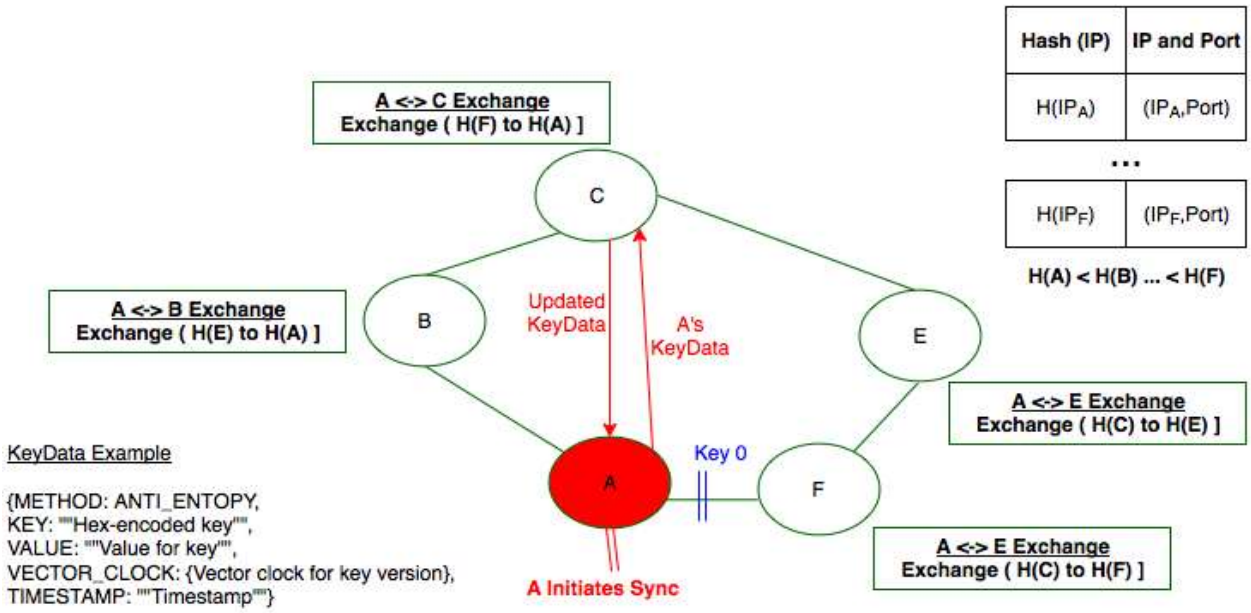

Figure 3. Overview of anti-entropy synchronization process.

KeyData Example

{METHOD: "GET, PUT, ANTI_ENTROPY,NOVAL",
KEY: ""Hex-encoded key"" (64 characters total),
VALUE: ""Value for key"",
VECTOR_CLOCK: {Vector clock for key version},
TIMESTAMP: ""Timestamp""}

KeyData Example

{METHOD: "PUT",
KEY: "07afe379...",
VALUE: "HI, This is the value I am writing!",
VECTOR_CLOCK: {178.32.5.2: "5", 178.5.4.3: "6"},
TIMESTAMP: "2016-04-24 17:55:19.530393"}

Figure 4. Format for requests in DynamiteDB (JSON objects).

We have created an interface for daemon services that are scheduled at the required rate. The anti-entropy process runs as a daemon service in the background and synchronize the values across the replicas asynchronously. The process is outlined in Figure 3. In this process, a node connects to one of the four other database nodes in the system (excluding itself). Node connections are done in a round robin fashion. When a node is chosen, the database node determines the entire key range that the two nodes share. This is done by analyzing the same file that the master node looks at to determine the key range for each database node. Once the range is determined, the initiating node iterates through all the keys it has in its persistent store and exchanges keys in the shared key range with the node it has chosen to do the anti-entropy sync with. Each key is sent as a JSON object in the format outlined in Figure 4 with the METHOD field set as ANTI-ENTROPY. When an anti-entropy message is received by a node, it updates its persistent storage with the proper key data in the same process that occurs as writes when the node executes a write. The node then sends back the updated key data to the node that initiated the anti-entropy sync so the initiator can update its persistent storage. Every individual key exchange is its own TCP connection. If there is an error at all in the syncing of a key, the key is skipped and the node attempts to exchange the next key.

All the nodes in the prototype were deployed in Amazon Web Services [3]. The team created two custom OS images: one for the database nodes and one for the master node. The custom image contained all source code and start up scripts to launch the team's services on bootup. The images were based on Ubuntu 14.04 LTS. To ensure the master nodes were not a single point of failure, the team set an auto scaling group for the master nodes. This was done by creating an Elastic Load Balancer and setting up a simple HTTP server on the master nodes [3]. Every 10 seconds, the load balancer issues HTTP requests as a health check to each master node in the auto scaling group [3]. If two successive HTTP requests fail, the load balancer determines the instance as unhealthy and launches a new master node from the custom master image. This rule ensures that two healthy nodes are active in the system most of the time. There can be less than 2 healthy nodes during the time it takes to diagnose a node as unhealthy and launch the new node in its place.

4. Evaluation and Performance

To test and evaluate the performance of DynamiteDB system, we used testing scripts that model both normal system operations and extreme cases that may cause system failure. We considered three different metrics that best describes its performance in terms of consistency, dependability, and distribution. These metrics are the following:

1. PUT and GET request latencies with varying data size
2. Anti-entropy time i.e. time to propagate changes to all the replicas
3. Distribution of keys across database nodes

4.1 PUT and GET request latency

One of the main metric that measures the performance of client-based systems is the request latency. This is measured from the time that the client issues either a GET or PUT request, to the time it receives the response for the issued request. This metric was tested for both GET and PUT requests using varying value sizes. Each test is then repeated 10 times to reduce variability in the result. The mean and standard deviation are then calculated and shown in Figure 5. As expected, the GET and PUT requests increases as the data value size increases. For GET request, the increased latency is due to the increased time needed to send larger data. The same is the case for PUT request but with additional write time as well. For a 256-byte value, the DynamiteDB system is able to achieve a GET request latency of approximately a quarter of a second and a PUT request latency of approximately a second.

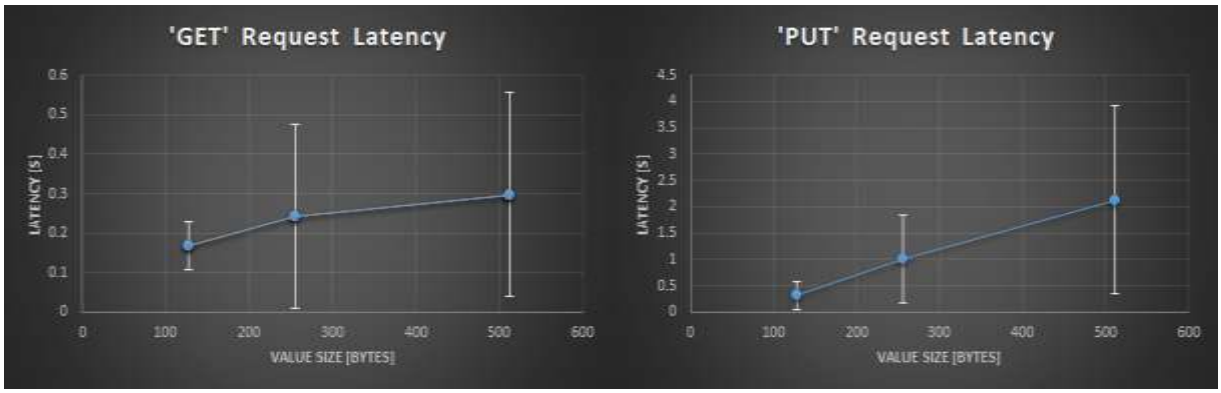

Figure 5. GET (left) and PUT (right) request latencies with varying value sizes

The PUT request latency is considerably longer than that of a GET request. This is caused mainly by two aspects of the design of DynamiteDB. First, when a client issues a PUT request, the master node first issues a GET request to the database nodes in order to obtain the latest vector clock before proceeding with the PUT request. As a GET request is a part of a PUT request, a PUT request would naturally take longer. Secondly, the DynamiteDB design mandates the database nodes to wait until the data that it receives for a PUT request is written completely to the hard disk before sending an acknowledgement to the master. This introduces delay in the PUT request as writing to hard disk is comparably slow. Despite these factors causing PUT request to be slower than a GET request, it remains an acceptable design if more reads than writes are expected in the system.

4.2 Anti-Entropy time

As discussed in the previous sections, DynamiteDB is an eventually consistent system. The master node only issues a PUT request to two out of the three database nodes that handles the key while the anti-entropy process handles eventual propagation of the data to the third node. For discussion, we define the anti-entropy time as the time is takes for data written to two database nodes to also be written to the third database node through the anti-entropy process. For a node that has failed, the anti-entropy time can be used as an estimate time from bootup to the reception of all updated keys.

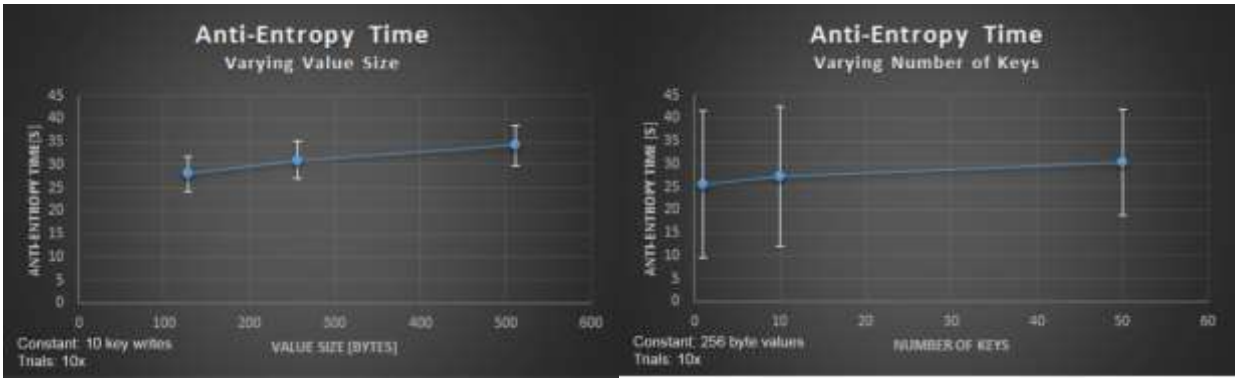

Figure 6. Anti-Entropy time with varying value size (left) and number of key writes (right).

To measure the anti-entropy time, one or multiple client PUT request with unique keys handled by the same subset of database nodes are issued. The third database node in the subset is then queried until all the key and value pairs written on the other two nodes appear on the third node. This metric is first tested using a constant 10 simultaneous key writes and varying key value sizes. It is then tested again using varying number of simultaneous key writes and a constant 256 bytes value size. The results are shown in Figure 6.

The anti-entropy time for 10 key writes and 256 byte values is approximately half a minute. This result was expected as the anti-entropy process is configured to exchange keys with other nodes approximately every half a minute. It can also be observed from the results that increasing the value size and/or key writes increases the anti-entropy time. This is explained by the increased data that is need to be exchanged between database nodes when either parameter is increased.

4.3 Key distribution across database nodes

One of the goals of the DynamiteDB system is to provide the service in a distributed manner where each database node has an equal share of responsibility. Equal distribution is important to prevent overloading of a single node and ensure a single database node failure has tolerable effect. As discussed above, the system used consistent hashing in order to distribute keys among database nodes. To evaluate its effectiveness, a sample of 175 random keys were ran through the system. Figure 7 shows the distribution of the keys across the 5 database nodes as a result of consistent hashing. Note that the result also includes the duplicated keys stored in the nodes.

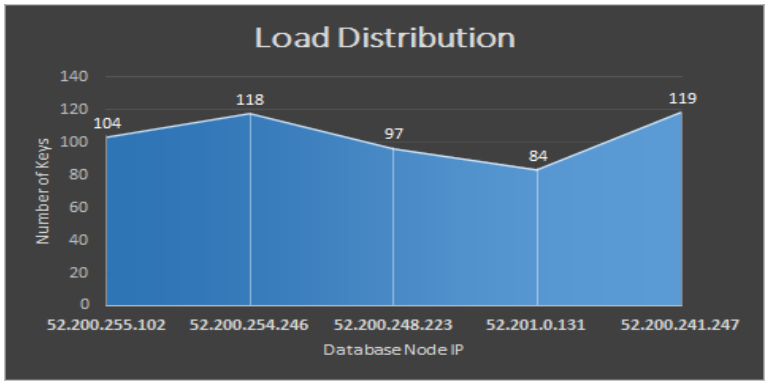

Figure 7. Distribution of keys across database nodes.

While the load is not completely even across database nodes, there is not one database node that serviced majority of the keys. If needed, the distribution of keys can be further improved through the addition of more database nodes or through the use of virtual nodes.

5. Conclusion and Future Work

Overall, all the required specifications outlined in our proposal were met in this prototype of DynamiteDB. The team succeeded in creating a distributed database with replication and distribution of keys using consistent hashing. Eventual consistency is ensured by the anti-entropy process that the team implemented. Clients are able to store and fetch data based on a given key. The team also was able to design a master node that implemented the (N=3,R=2,W=2) protocol for the database. Additionally, the team was able to avoid having the master as a single point of failure by creating a load balancer and assigning the nodes to an auto scaling group.

If the team had time to continue improving the design, the team would implement a process in which database nodes could actively join and leave the system. The team would need to create its own protocol to allow nodes to broadcast that they are leaving or joining the system, and allow the nodes to redistribute the proper keys to the new node. In addition, instead of the anti-entropy process exchanging every key in the shared range between two nodes, the team would like to just send a Merkle tree of all the keys to detect inconsistencies in the data at the two nodes. The Merkle tree would greatly improve the efficiency of the process by reducing the amount of keys that need to be exchanged between the two nodes, since after exchanging the Merkle tree the nodes would only have to exchange keys that both nodes do not have the same data for. Another addition that would be helpful is the use of caching during writes. Currently, the acknowledgement of a PUT request is delayed until the data is written completely into a file on the hard disk. With a cache system, the data can be written into a cache and the acknowledgment can be sent while the database permanently store the cached data into the hard disk. With less delay on sending write acknowledgments, the PUT latency will be reduced.

Work Distribution

Team Member	High Level Responsibility
Tyler Lisowski	Key-Value Store Object Serialization and Anti Entropy Mechanism
Satyajeet Gawas	Framework for implementing Database node services and background daemons
Yakshdeep Kaul	Master Implementation
Ramon Sua	Client Implementation and Testing scripts

Borrowed Code/Service

1. Amazon Web Services (AWS) Instances
2. AWS Load Balancer

Submission

1. dynamiteDB – source code for database node i.e. key value store
2. StartupScripts - scripts used to configure amazon instances
3. Python_Master_Client - source code for master node and client
4. Test - testing scripts and performance results

6. References

[1] G. DeCandia et al., "Dynamo: Amazon's Highly Available Key-Value Store," *Proc. 21st ACM SIGOPS Symp. Operating Systems Principles* (SOSP 07), ACM, 2007, pp. 205-220.

[2] 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.

[3] "Amazon Web Services", *Amazon.com*, 2016. [Online]. Available: https://aws.amazon.com/. [Accessed: 24- Apr- 2016].