Distributed Key Value Store

M.Eng. Project- Spring 2013

Project Advisor: Prof. Alan J. Demers

Team Members: Steeb Dsilva(smd323), Harsh Panchal(hpp8),
Jayanth Parayil Kumarji(jp972), Supriya Singh(ss2895)

Contents

Project Description

The distributed key value store has the following features:

a) Consists of a number of clusters, each of them being a key value store
b) Store can hold a string key and string value
c) Each coordinator has a URL which can be used to put and get key value pairs.
d) Coordinators maintain consistency with each other by exchanging key value pairs with each other using anti-entropy

The specifications of the distributed key value store are as follows:

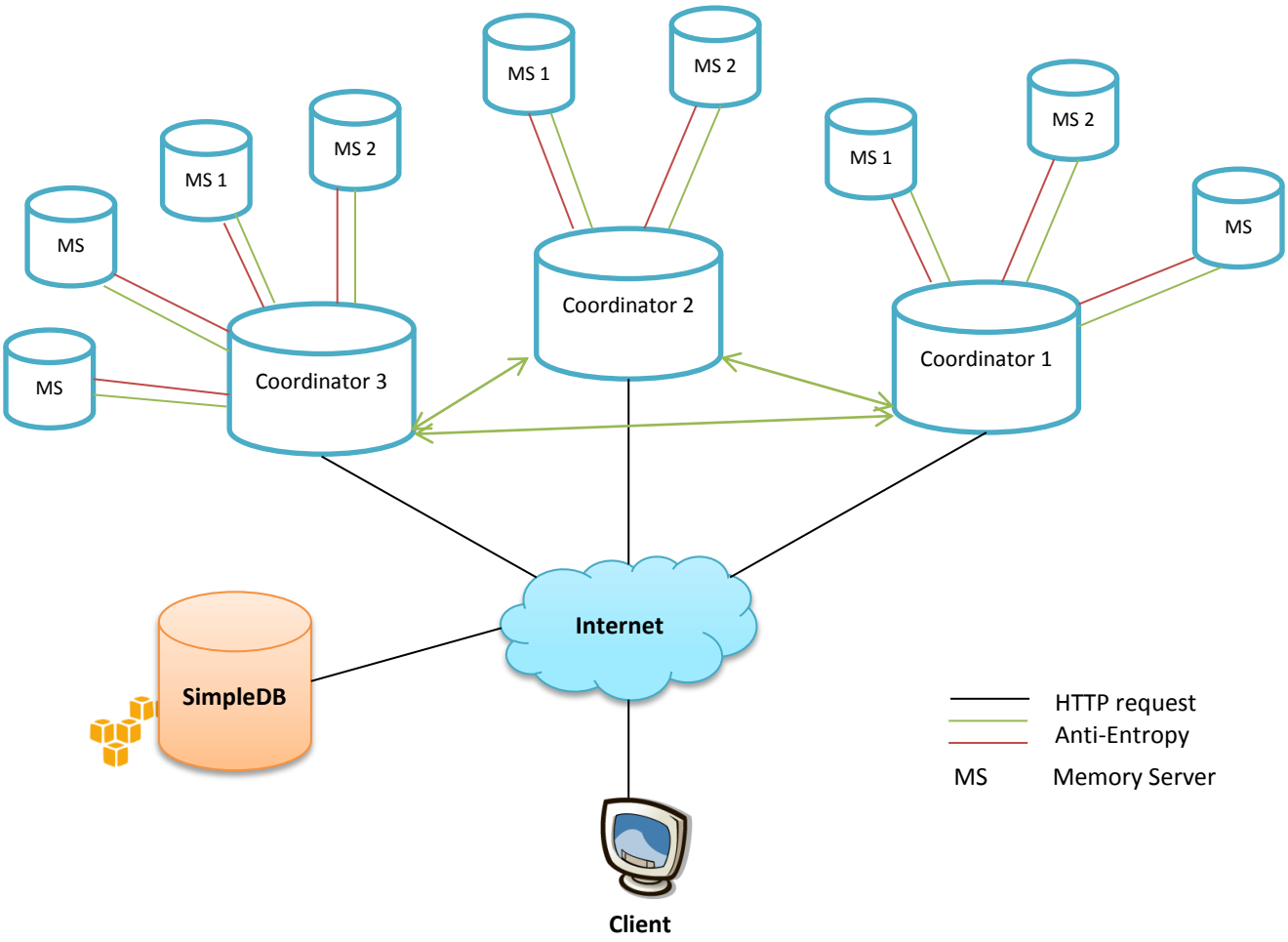
a) Virtual Clusters:
Key value pairs are stored in a virtual cluster consisting of a single coordinator and several memory servers. The coordinator does not store any key value pairs (hereby referred to as K-V-pairs). They are stored on one of its memory servers by using a hashing function on the key along with a unique timestamp given by the coordinator. If the user makes a batch put request, all the K-V-pairs in that put request are given the same timestamp.

b) Maintaining group membership in SimpleDB:
Each cluster coordinator has a local list of the other coordinators in the system. This local list is periodically updated with the master list of coordinators that is stored in SimpleDB. If a coordinator cannot reach another coordinator, it marks it as suspicious and removes it from SimpleDB the next time it accesses it. This prevents coordinators from accessing the SimpleDB at the same time when they detect that a server is down.

c) Update propagation using Anti-Entropy:
To keep the system consistent, the coordinators exchange K-V-pairs with each other using Anti-Entropy. Each coordinator periodically picks one other coordinator from its local list of coordinators to perform Anti-Entropy. During Anti-Entropy, they share the most recent updates from their own stores while simultaneously comparing the checksums of the K-V-pairs in their memory servers. A 32-bit hash on the key and the timestamp of the K-V-pair is calculated. The hash for all the keys is XOR-ed that will be the checksum for the database. When the checksums of both the coordinators match, they stop anti-entropy.

d) 1-resilience:
Even though the system maintains consistency by regularly performing Anti-Entropy with other coordinators, some updates may be lost if a cluster fails just after it updates its store and before it can perform Anti-Entropy with any other coordinator. To prevent this, we store the K-V-pair in one other coordinator as backup. This backup is randomly chosen from the local list of coordinators. The only time a coordinator does not write a K-V-pair to a backup is when only one cluster is up, or when its local list is empty.

Architecture



The system consists of multiple components.

1. Virtual clusters:
   Each cluster consists of a coordinator and one or more memory servers and behaves logically as a single data store. The client requests are addressed by the coordinator and the memory server act as data stores addressing coordinator requests.
2. SimpleDB:
   The group membership of the system is maintained by SimpleDB
3. Channels:
   - **Coordinator and Memory Servers:** Each cluster within it maintains 2 sets of TCP connections between the coordinators and the memory servers. One set of TCP connections are used for the GET-PUT requests and the other is used to perform Anti-Entropy.
   - **Coordinators:** The client can send its PUT/GET requests to any cluster (coordinator) through TCP channels. Each cluster propagates its changes to other clusters to ensure eventual system consistency. The system achieves consistency using Anti-Entropy, which we shall be discussing later.

User API

The system provides the user with GET and PUT requests to access the key value store.

a) PUT:
   The user can provide a single or multiple key value pairs to be entered in a store through a web page. When a single value is entered it will be treated as a 'Single Put Request' and in case for multiple keys it will be treated as 'Batch Put Request'.
   The status of the put message will be sent to the user.

b) GET:
   The user requests for the value of a key using the web page.
   The response for the request will be the value for the key or 'KEY NOT FOUND'

SimpleDB

The well-known location in SimpleDB is saved as follows:

address: The field would contain the IP address of the coordinator and a timestamp as the cluster initializes.
The default value of this field is 'NONE'

list: This field would be used to maintain the group membership and contains the list of all the coordinator IP addresses present in system.

itemName()	address	list
Coordinator		

Virtual Data Clusters

To bring up a cluster of 1 coordinator and N memory servers, i.e. (N+1) machines are started up and the project is loaded onto each of them. Each machine on start-up does the following:

a) Check the well-known location for the presence of coordinator IP in SimpleDB.

b) Coordinator

1. If the read value of the coordinator address is 'NONE', it means that currently there is no coordinator for the cluster and the machine registers its own IP address as the coordinator along with the current timestamp.
2. It then waits for a set of memory server to connect with it, to form the cluster.
3. Every time a memory server registers itself, the coordinator initializes two TCP channels between the coordinator and each memory server (one for anti-entropy requests and the other for client get and put requests). These channels are persistent throughout the lifetime of the cluster.

4. Each coordinator has two listeners (one for Anti-Entropy requests and the other for backup-write requests).
5. Once the required number of memory servers are connected, the coordinator erases its IP address from SimpleDB and adds its value on to the list of coordinators also maintained in the SimpleDB

c) Memory Servers

1. If the value of the well-known location of the coordinator IP address in SimpleDB already contains a value, the machine checks the timestamp of that value. There are two possible scenarios:

    - If the time elapsed is more than SIMPLEDB_TIMEOUT duration (currently set at 5 minutes), it indicates that the coordinator or the memory server failed while booting up and left its IP in the location. In such a case, the machine will force write its own IP into the coordinator location and wait for memory server connections.
    - However, if the timeout duration has not passed, it would imply that there already exists a coordinator that is waiting for a set of memory servers to complete the cluster.

2. It reads the IP address value and registers itself to the coordinator and two TCP channels as mentioned above are opened for communication. It then waits for requests from coordinator.

Group Membership

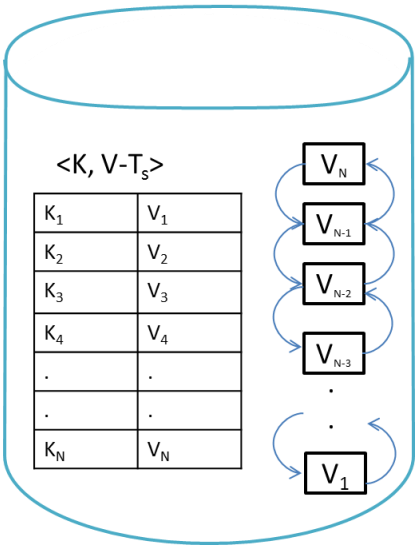
The coordinator of each cluster maintains its own local member list. This list contains a set of clusters (coordinator IP) which this cluster has knowledge of. At any point if the coordinator is unable to reach any of the cluster during its operations i.e.: backup write, Anti-Entropy; it marks the corresponding cluster (coordinator IP) as 'suspected'.

However, if the coordinator is able to connect to the 'suspected' cluster during another set of operations, then the corresponding 'suspected' flag for the cluster is removed.

After a certain duration (every 30 seconds), the coordinator flips an N sided coin (where N is the number of entries in its local list) and if a certain value is obtained, then the coordinator updates its local member list with membership list present in SimpleDB. The N sided coin is flipped using random numbers. Also it removes the cluster entries from the SimpleDB membership list that is has marked suspected.

The membership list at SimpleDB acts as the source for other clusters to update their member list. Under a perfect scenario, with no failure, the membership list on SimpleDB and the member list present on each cluster would be identical.

Key Value Store Data Structure



For each put request, we compute the timestamp which is a combination of the actual timestamp (generated by the coordinator when the Put request was received) and the coordinator IP address. The timestamp is computed at the coordinator.

Timestamp value = new Timestamp (Time, coordinator_IP_address)
This would enable us to differentiate the most recent timestamp values. In an event when the time value matches, the copies are distinguished based on the coordinator IP address.

The components of the Key value store are as follows:

Checksum: There is a 32-bit checksum maintained at each coordinator to keep track of the K-V-pairs the cluster currently holds, which is updated with every PUT request.

HashMap: The HashMap stores the K-V-pair by using the key of the K-V-pair as the corresponding key in the HashMap, and the corresponding value in the HashMap is the value of the KV pair with its corresponding timestamp value and a reference to the corresponding node in the Doubly Linked List for faster access.

Doubly Linked List: The Doubly Linked List consists of the key with the corresponding timestamp, the list being in reverse timestamp order; therefore the most recent item is at the top.

Anti-entropy Pointer: There is a pointer maintained for the list in order to indicate which element in the list is the next one (most recent) to be used in an Anti-Entropy request. In the case of a new Anti-Entropy request the pointer is always re-initialized to the top of the list.

Anti-entropy status bit: It is used to indicate whether the particular machine is under Anti-Entropy or not.

Client Requests

Put Requests

When the coordinator receives a put request the following things happen:

1. The coordinator computes and assigns a Timestamp Value to every new request.

2. It then computes the checksum of the key and the corresponding timestamp as mentioned in the previous section.

3. The checksum at the coordinator is recomputed by XOR-ing the old value with the checksum of the key to be added.

4. The memory server that will hold this K-V-Pair is decided by applying a hash function on the key. This K-V-pair is then sent to the corresponding memory server.
   Hash function: (Summation of the ASCII values of every character in the key) % (# memory servers)
   In order to do so, the coordinator needs to acquire a lock on the TCP channel for GET-PUT requests. Once the lock is acquired, data is sent to the memory server.

5. At the memory server, a lookup is made into the HashMap of its K-V-Store based on the key and the corresponding response it sent to the coordinator. There are two possible scenarios:
   - In the case that it is not present, an entry is made into the Hash Map, and also at the head of the Doubly Linked List.
   - In the case of the value already being present in the HashMap, it is updated with the new value and timestamp only if the timestamp of the request is more recent than the timestamp of the key present in the HashMap. Also, an entry is made in the Doubly Linked List (at the appropriate location according to timestamp) and the old entry is removed.
   - In the case of an already present key in the memory server it responds to the coordinator with the old value of the key. The checksum for this old value is calculated and is then XOR-ed with the coordinator checksum to negate the effect of the old value.

6. The lock on the TCP channel is then released and an Http response with a Success message is sent to the user.

Batch Put Requests

The process is very similar to the above with the following change:

- The memory server number for each key is calculated and locks are acquired for those TCP channels in incremental order of the memory server numbers to ensure that no deadlock occurs due to multiple requests. This also allows requests to be executed in parallel on disjoint staging servers.

- All the K-V-pairs in a batch request are assigned the same unique timestamp consisting of the current time along with the coordinator's IP address.

Achieving 1-Resilience

For every PUT request, the coordinator selects a random backup coordinator from its local member list and tries to write the K-V-pair to the corresponding cluster. In case a cluster is unreachable, the coordinator marks the cluster as a suspect for removal from Simple DB later by the Group Membership update thread. Also, another non-suspect cluster is contacted until at least one other cluster is selected as a backup or until the list is exhausted in which case no backup is assigned.

Get Requests

In a GET request for a particular key, the coordinator computes the memory server at which the key will be present using the hash function and locks the particular stream.

On receiving the request, the memory server retrieves the corresponding value from its Hash Map. The memory server replies back to the Coordinator with a K-V-pair message containing the value for the corresponding key or 'KEY NOT FOUND' which is then returned to the user as an Http response.

Anti-Entropy

Flow

Each coordinator periodically (every 2 seconds) spawns a new thread that does the following:

a) Check the anti-entropy status of own cluster.
b) If anti-entropy is already being performed, the thread does not do anything.
c) If the cluster itself is not performing Anti-Entropy, then first changes its own Anti-Entropy status to true. Then it changes the Anti-Entropy status at every memory server to true and the Anti-Entropy pointer at each memory server is initialized to the top to the Doubly Linked List.
d) The thread now selects one other coordinator from its local list of coordinators at random. If the server is not available for Anti-Entropy, it is marked and not tried again. It does this till one of the coordinators accepts the Anti-Entropy request or till it has requested all the coordinators in the list.

Performing Anti-Entropy

a) Whenever a coordinator receives an Anti-Entropy request, it checks its own status to see if it is currently performing Anti-Entropy.
b) If it is not currently performing Anti-Entropy, then it compares the checksum in the request against its own checksum value.

c) If they do not match, it means that there is a difference in their K-V-stores. Only in such a case will it accept the Anti-Entropy request and change the Anti-Entropy status at its own self and at its memory servers to true.

Exchanging key-value pairs

a) After a coordinator accepts an Anti-Entropy request, the Anti-Entropy initiator sends his most recent K-V-pairs to him. The coordinator puts these K-V-pairs into its own memory servers using the Anti-Entropy channels.
b) In response to the K-V-pairs sent to it, the coordinator now tries to calculate the most recent key value pair that is contains.
c) To calculate the most recent K-V-pair, it asks all its memory servers for their most recent K-V-pairs and compares the timestamps of each of the received K-V-pairs. The most-recent of these if then sent to the initiator coordinator as a response.
d) The coordinator also takes care of concurrent user put requests to the store along with the on-going Anti-Entropy.
   While the Anti-Entropy is going on, it locally stores each of the K-V-pair puts done by the user. These puts that are stored locally are compared with the recent K-V-list obtained from each memory server. The most recent K-V-pair of these is then sent to the initiator coordinator as the Anti-Entropy response.
e) This process goes back and forth, every time checking its own checksum with the checksum of the initiator coordinator present in the request. Whenever the initiator or this coordinator realizes that their checksums are the same, its response will tell the other to stop Anti-Entropy.

NOTE: The coordinator also maintains a structure similar to the memory server, to store the response of the memory servers when it asks them for their most recent K-V-pairs and the concurrent client requests. This store is active only when the cluster is performing Anti-Entropy.

Evaluation

Before Code Changes:

Volume of PUT

Case 1: Best Case

Following is the performance of the system on different number of keys.

i) The keys used to inundate the server were unique and the requests were sent to a single server.
ii) Each PUT request achieved 1-resilience.
iii) 2 clusters were used for this test each with a single memory-server
[1 coordinator + 1 memory server]

No of Keys	Time (ms)
10	90
100	853
1000	8601
5000	53456
10000	132307
20000	185615
40000	442088

Case 2: Worst Case

Following is the performance of the system on different number of keys.

i) First unique keys were used to inundate the server. Then the same set of keys was sent to the server in the same order.
So, if keys 1-100 are inserted, after the insertion, key = 1 will have the oldest timestamp. By putting the key 1-100 again in the same order, the key that is put will always be at the bottom of the doubly linked list. Hence, it is the worst case scenario.
ii) Each PUT request achieved 1-resilience.
iii) 2 clusters were used for this test each with a single memory-server
[1 coordinator + 1 memory server]

No of Keys	Time (ms)
10	91
100	862
1000	9391
5000	51652
10000	112282
20000	174151
40000	403422

Case 3: Average Case

Following is the performance of the system on different number of keys.

i) Keys were generated at random and sent to server
ii) Each PUT request achieved 1-resilience.
iii) 2 clusters were used for this test each with a single memory-server [1 coordinator + 1 memory server]

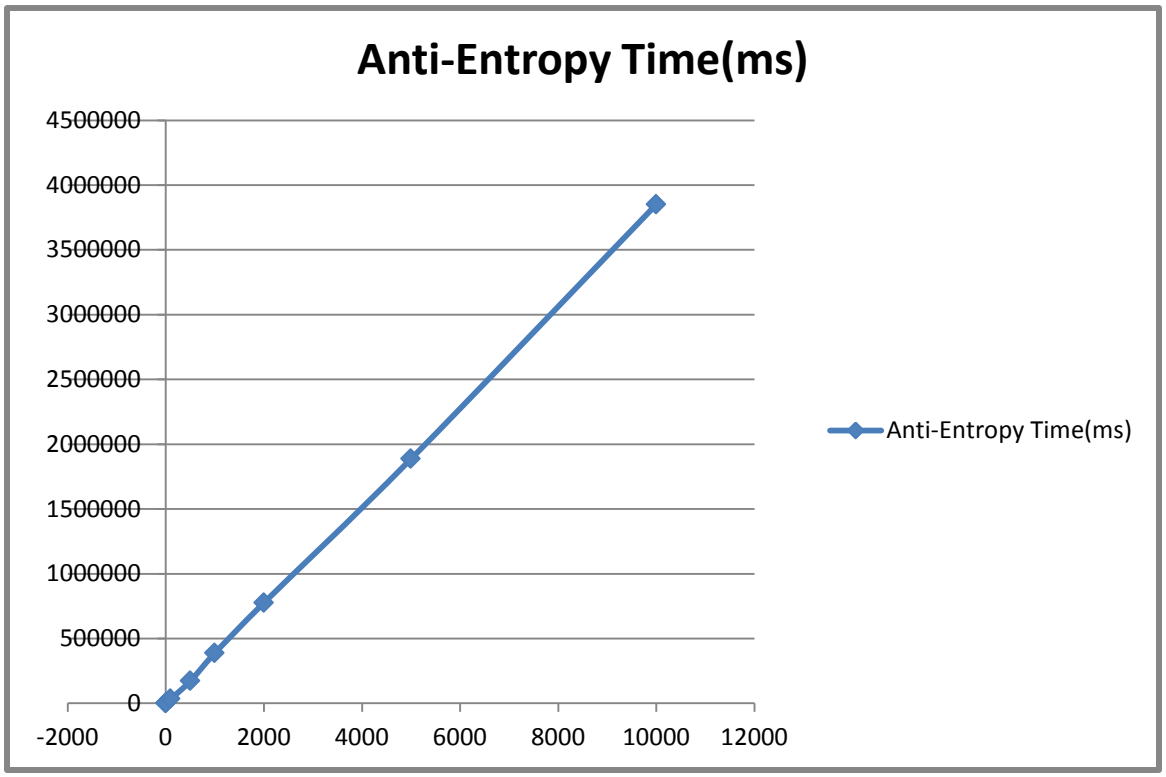
No of Keys	Time (ms)
10	70
100	938
1000	8554
5000	44954
10000	111828
20000	246734
40000	470851

Anti-Entropy Time duration:

Following is the time required for the system [2 clusters] to achieve consistency after being inundated with different number of keys.

i) The tests were performed on EC2 machines.
ii) The keys used to inundate the server were unique.
iii) The PUT requests were issued to a single cluster.
iv) Each cluster used consisted of a single coordinator and three memory servers. [1 coordinator +3 memory servers]

No of Keys	Initiator cluster: Time(ms)	Contributor cluster: Time(ms)
0	540	262
10	3674	3586
100	33321	33190
500	172895	172798
1000	388450	388327
2000	776604	776467
5000	1887492	1887351
10000	3851465	3581600



Issues Addressed:

- Before: The Doubly Linked List was being accessed sequentially. Therefore, in the case of large number of keys in a data cluster and a Put operation of a key already in the data store, the delete of the old value will be very slow.
  Now: The value for each key in the HashMap also has a reference to the corresponding node in the Doubly Linked List for faster access.
- Before: A shuffle operation was being performed on the list of coordinators and was traversed to select another machine for Anti-Entropy. A shuffle operation is algorithmically expensive and will degrade performance.
  Now: A coordinator is being selected at random from the list. If it could not perform Anti-Entropy, it is marked and another coordinator is selected at random.

After Code Changes:

Volume of PUT

Case 1: Best Case

Following is the performance of the system on different number of keys.

iv) The keys used to inundate the server were unique and the requests were sent to a single server.
v) Each PUT request achieved 1-resilience.
vi) 2 clusters were used for this test each with a single memory-server
[1 coordinator + 1 memory server]

No of Keys	Time (ms)
10	112
100	853
1000	8416
5000	37829
10000	68646
20000	133461
40000	269605

Case 2: Worst Case

Following is the performance of the system on different number of keys.

i) First unique keys were used to inundate the server. Then the same set of keys was sent to the server in the same order.
So, if keys 1-100 are inserted, after the insertion, key = 1 will have the oldest timestamp. By putting the key 1-100 again in the same order, the key that is put will always be at the bottom of the doubly linked list. Hence, it is the worst case scenario.
i) Each PUT request achieved 1-resilience.
ii) 2 clusters were used for this test each with a single memory-server
[1 coordinator + 1 memory server]

No of Keys	Time (ms)
10	92
100	846
1000	8592
5000	39757
10000	70737
20000	139950
40000	279311

Case 3: Average Case

Following is the performance of the system on different number of keys.

iv) Keys were generated at random and sent to server
v) Each PUT request achieved 1-resilience.
vi) 2 clusters were used for this test each with a single memory-server [1 coordinator + 1 memory server]

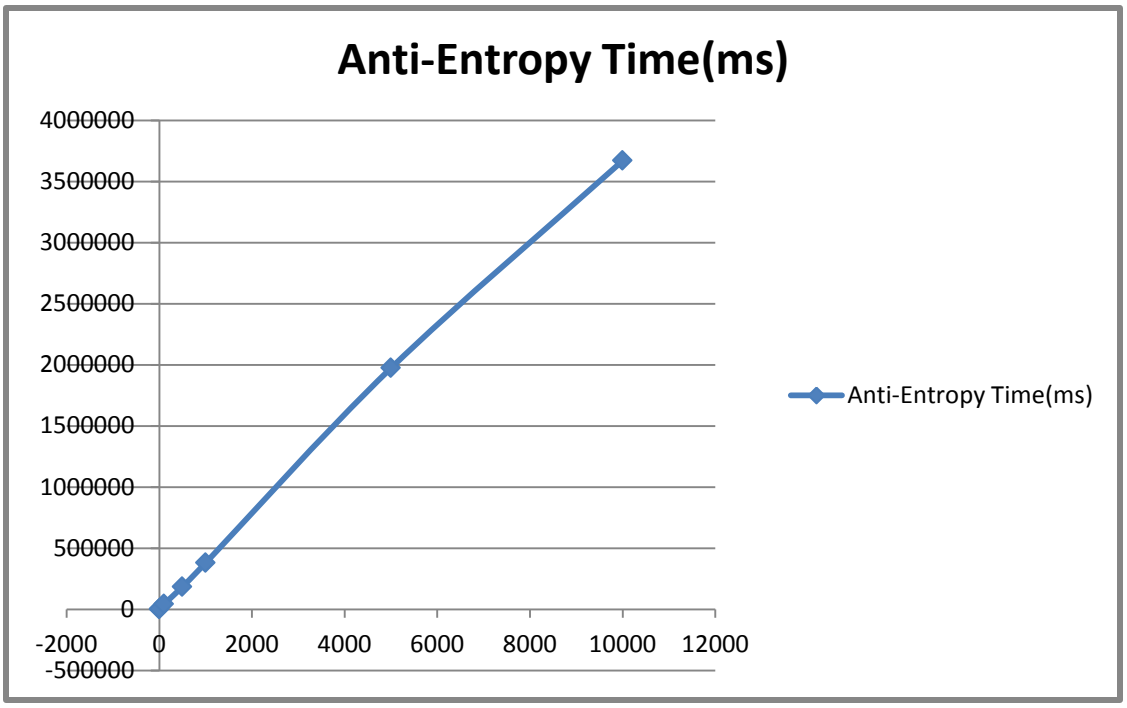
No of Keys	Time (ms)
10	103
100	937
1000	8723
5000	40256
10000	78311
20000	146819
40000	278326

Anti-Entropy Time duration:

Following is the time required for the system [2 clusters] to achieve consistency after being inundated with different number of keys.

i) First unique keys were used to inundate one server. Then the same set of keys is used to inundate the other server and Anti-Entropy is done. Hence, the keys that are received by a server during Anti-Entropy will overwrite the keys that are already present.
ii) Each cluster consisted of a single coordinator and three memory servers. [1 coordinator +3 memory servers]

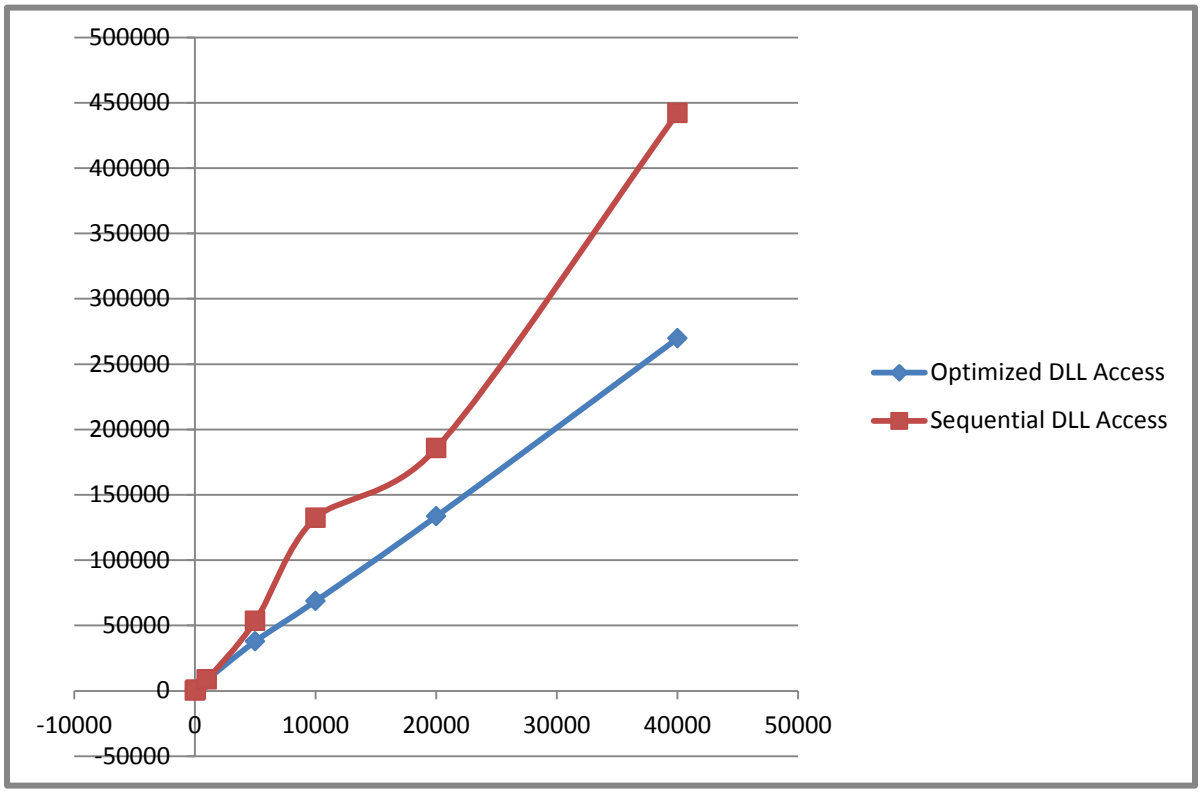
No of Keys	Initiator cluster: Time(ms)	Contributor cluster: Time(ms)
0	239	625
10	3916	4071
100	42974	43113
500	184415	184697
1000	379871	379676
5000	1973884	1972410
10000	3669633	3669900



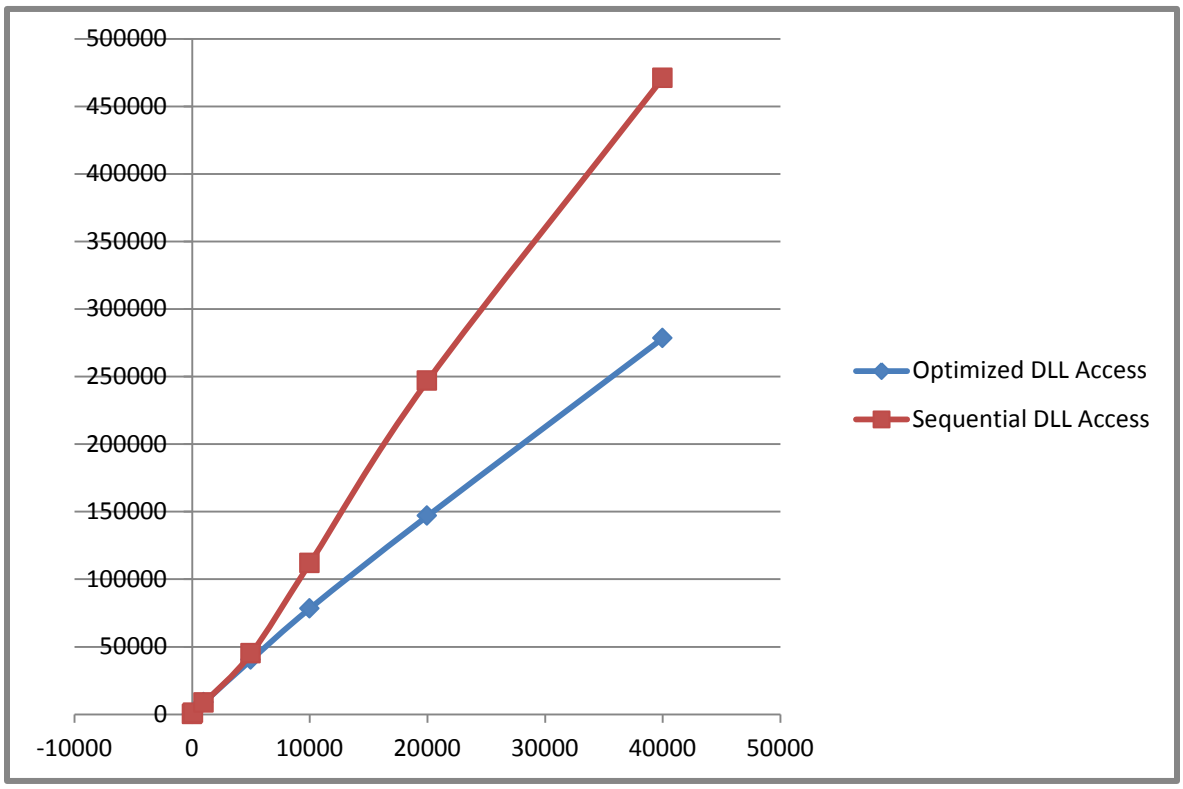
Comparison of PUT Requests

Below is the comparison of the best, worst and average cases of put before the DLL optimization and after the optimization.

Case 1: Best Case



Case 2: Average Case



Case 3 : Worst Case