International Institute of Information Technology, Bangalore

Distributed Cache

Satvik Ramaprasad | IMT2016008

May 14, 2020

Contents

1 Objective

The objective of this project is to apply and explore the concepts learnt in the last semester and build a distributed system.

The project chosen was to build a completely peer to peer distributed cache system. A cache is a high speed layer which stores a subset of data and increases performance, reduce load and decreases latency to meet the every increasing demand for scale and availability, modern cache systems are distributed in nature.

1.1 Main components

- **Mock Database** - A system which emulates a real distributed database. Acts as source of truth for correctfullnes of data.
- **Cache Node** - Node on consistent ring which caches portion of the data
- **Cluster Manager** - Creates the node cluster and monitors them. It also elastically increases or decrease nodes depending on load.
- **Application Driver** - Test application which fires queries rapidly on the system

1.2 Experiments

I had planned a series of experiments to execute. Some of them were:

- Performance comparison on different cache replacement policies with different request patterns
- Performance gain in response time depending on distribution
- Availability guarantees during partition or network failures

Some of the distributed concepts to use are gossip protocol, consistent hashing with sharding and membership lists. Apart from this, multithreaded programming, networking, latest C++ and design patterns will be explored in the project.

2 Design

2.1 Cache System

I adopted a simple design. The cache will be a simple key-value store. The default replacement policy will be Least Recently Used (LRU). Both the database and the cache expose the same network interface hence making them interoperable. In case of a cache miss, the cache will transparently look up the value in the database and respond.

The mock database will be implemented as a Postgres database and as a simple in memory C++ database.

2.2 Gossip Protocol and membership lists

The different nodes will be running gossip protocol in between them. The membership lists and consistent ring details will piggy back on the heartbeat messages. The membership lists will have the list of active nodes. The whole system can detect failed nodes and new nodes in short time.

2.3 Query Execution Strategy

Since this is a complete peer to peer system, there is no master. Therefore there is no single point of contact for a client application. There are broadly 2 design approaches -

2.3.1 Strategy 1 - Application knows detailed information

Here the application knows details of all the nodes and can therefore send the query to the exact target node. Advantage of this system is that there is lower latency as there is no extra hops. Another advantage of the system could be that the overall network load is lower. However disadvantage of the system is that the application needs to be part of the cluster as well participating in the gossip protocol even if its just for reading. Another disadvantage is that the application logic gets complex.

2.3.2 Strategy 2 - Application knows little information

In this strategy, the application is aware of just a few nodes in the cluster. The application can send a query to any of the nodes. The node then becomes the coordinator for the request and forwards the query to the target node as per the

ring. The disadvantage here is the extra hop. The advantage is the simplicity for the application. I have adopted this strategy.

2.4 Performance Measurements

As I will be running tests on processes in my local system, performance is a little tricky for many reasons. The main reason is that the local network in my host system is an inaccurate representation of a real network. Firstly because the network delays is very low. Second, postgres itself uses cache and therefore it will be difficult to measure performance gain. Lastly, since we are running the processes in the same system, there can in fact be an over all decrease in performance in distribution. Therefore, in order to correctly measure performance, I have chosen to artificially add delays in the database, cache and the network.

3 Implementation

Modern C++ has been used to implement the cache system. Efforts have been taken to make the system as flexible and robust as possible while keeping efficiency in mind.

3.1 Multi-threading

Multi-threading has been used extensively in the system. Each node is a separate process and it is multi threaded as well. One common requirement across the different type of nodes is to have multiple threads serve requests.

In order to enable reuse, I made a `MultiThreadedServerInterface`. Any server can become multithreaded simply by complying to the interface. In my system both `MockDatabase` and `MockDatabase` are multithreaded.

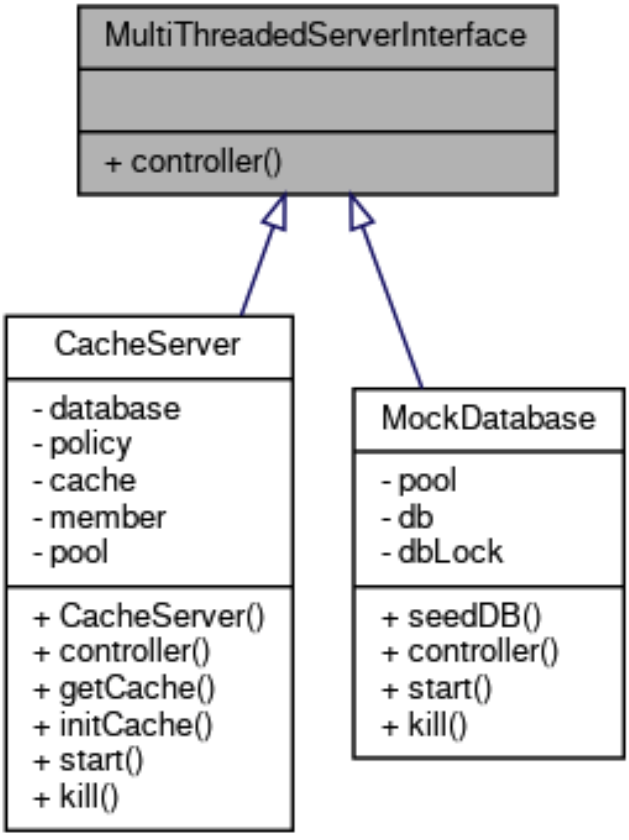

Figure 3.1: MultiThreadedServerInterface

`ServerThreadPool` implements the consumer-producer model for serving requests using multiple threads.

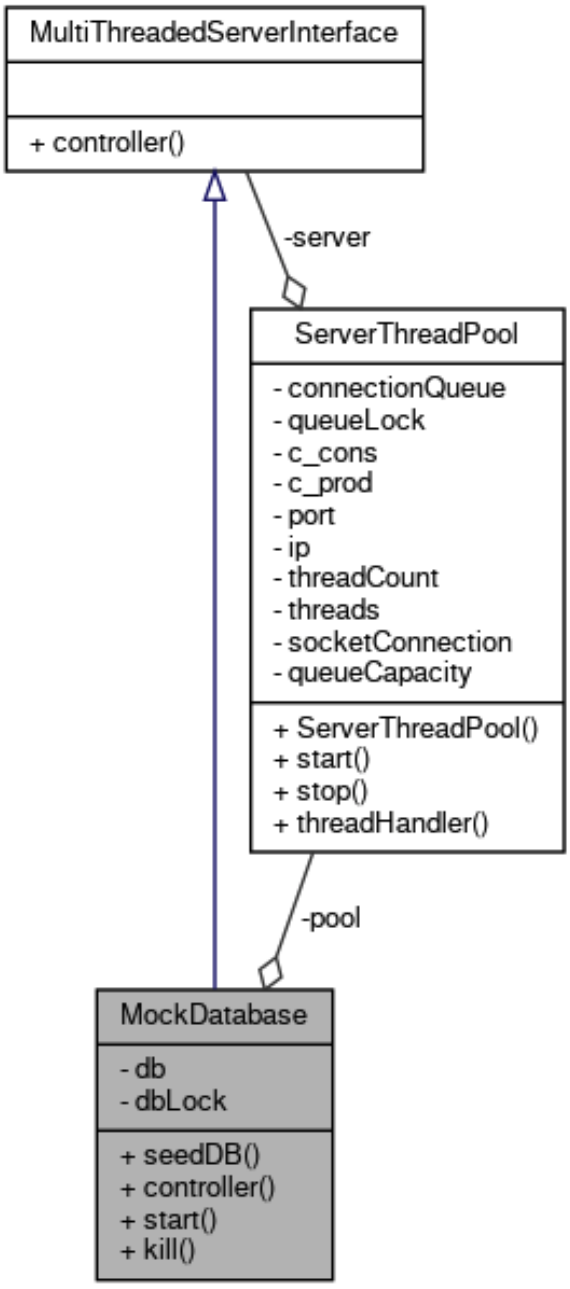

Figure 3.2: Mock Database MultiThreaded using ServerThreadPool

3.2 Cache System

When it comes to the cache itself, there isn't much difference in the implementation between a single node and a distributed node. The following is the class diagram for the cache system. Note that this isn't the cache server itself, i.e. it doesn't serve any requests directly.

Strategy Pattern has been used here. `CachePolicyInterface` refers to the cache replacement policy while `DataConnectorInterface` refers to the source database.

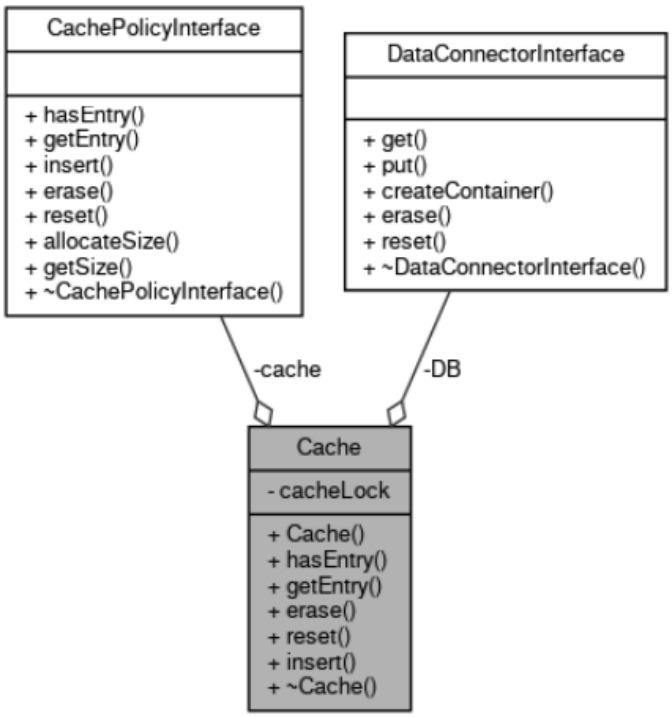

Figure 3.3: Cache System

3.3 Gossip Protocol

The gossip protocol is arguably the most essential feature of any distributed system. It is the de-facto dissemination protocol used. It works as follows.

If a system has N nodes, each node sends messages periodically to K random nodes ($K << N$). The gossip protocol states that the information is propagated in the network within $log_K(N)$ time. This is a protocol resilent to process or network failures. It is also allows new nodes to join and leave the cluster seamlessly.

For the gossip protocol to work, each nodes needs to have a list of other active nodes in the cluster. For this, it maintains something called a membership list.

The membership list itself needs to remain up to date.

The gossip protocol has been implemented using something similar to observer pattern. Any information that needs to be disseminated via gossip has to comply to `GossipArtifact` interface. I called it observer pattern because whenever an update comes, the artifact gets notified via an `update` call. Another interesting to node is the `getPayLoad` function. The gossip system calls this function to serialize the artifact.

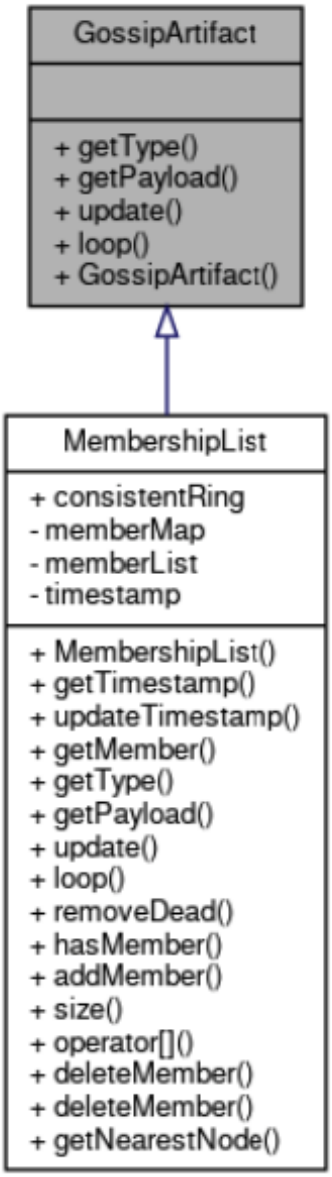

Figure 3.4: Gossip Artifact

3.4 Membership Lists

Membership list represents the active nodes in the system. It should get updated both in case a new node enters and if a new node leaves the cluster. This is done by heartbeats. Each node sends heart beats to other nodes letting them know its alive. When a heart beat from a new node is received, the receiving node adds the sender node to its list of active nodes. However, to increase the speed at which membership lists get updated, along with heartbeat, the entire membership list is also sent. This dissemination of heart beats is done by Gossip Protocol.

In case a node doesn't hear from a node for T_FAIL time, it stops sending information of that node to other nodes during heartbeats. If it doesn't hear from a node for T_REMOVE time, it removes the node from the membership list entirely. $T\_REMOVE > T\_FAIL$ because if a node is removed prematurely, it will get added soon via heartbeats as a new node and in effect the dead node will never be removed from the cluster.

In conclusion, gossip protocol requires membership lists to work and membership lists need gossip protocol to work and hence forming a cyclic dependency. That is why the `MembershipList` is an `GossipArtifact`.

The figure 3.5 shows the class diagram of `MemberNode`. Any entity which wants to be a node in the system needs to contain `MemberNode` object. This object can then be used to register gossip artifacts. The node internally will be running gossip protocol, maintain membership lists and disseminating information of the artifacts via gossip.

Note that when a node joins the cluster, it has to be introduced to an introducer node.

3.5 Consistent Hashing

Consistent hashing is one of the ways to shard a system. In this method, the nodes all fall on a ring numbered from say 0 to N. When a request comes, the position for the key on the ring is calculated. Then the nearest clockwise node is the node that needs to serve the request.

The position on the ring is calculated as follows $hash(key)\%N$. For this project, I found that `std::hash` worked fairly well.

Note that the information of the ring is also a gossip artifact.

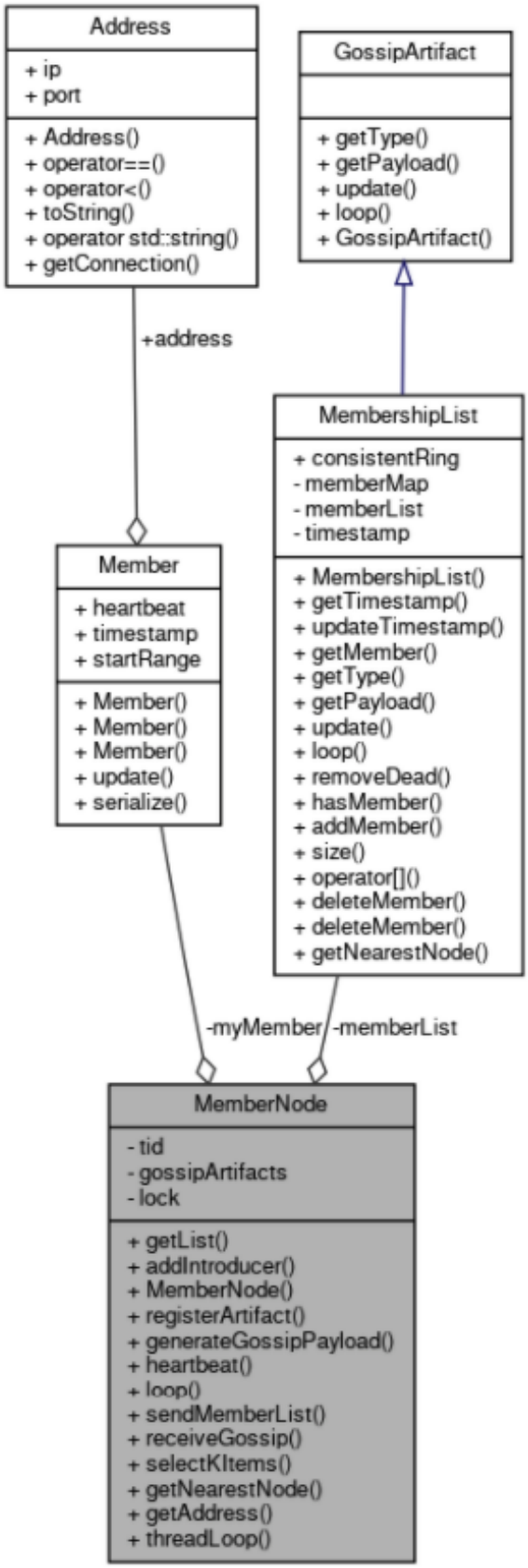

Figure 3.5: Member Node

3.6 Bringing Everything Together

The following is the class diagram of all structures involved in the distributed cache system everything coming together!

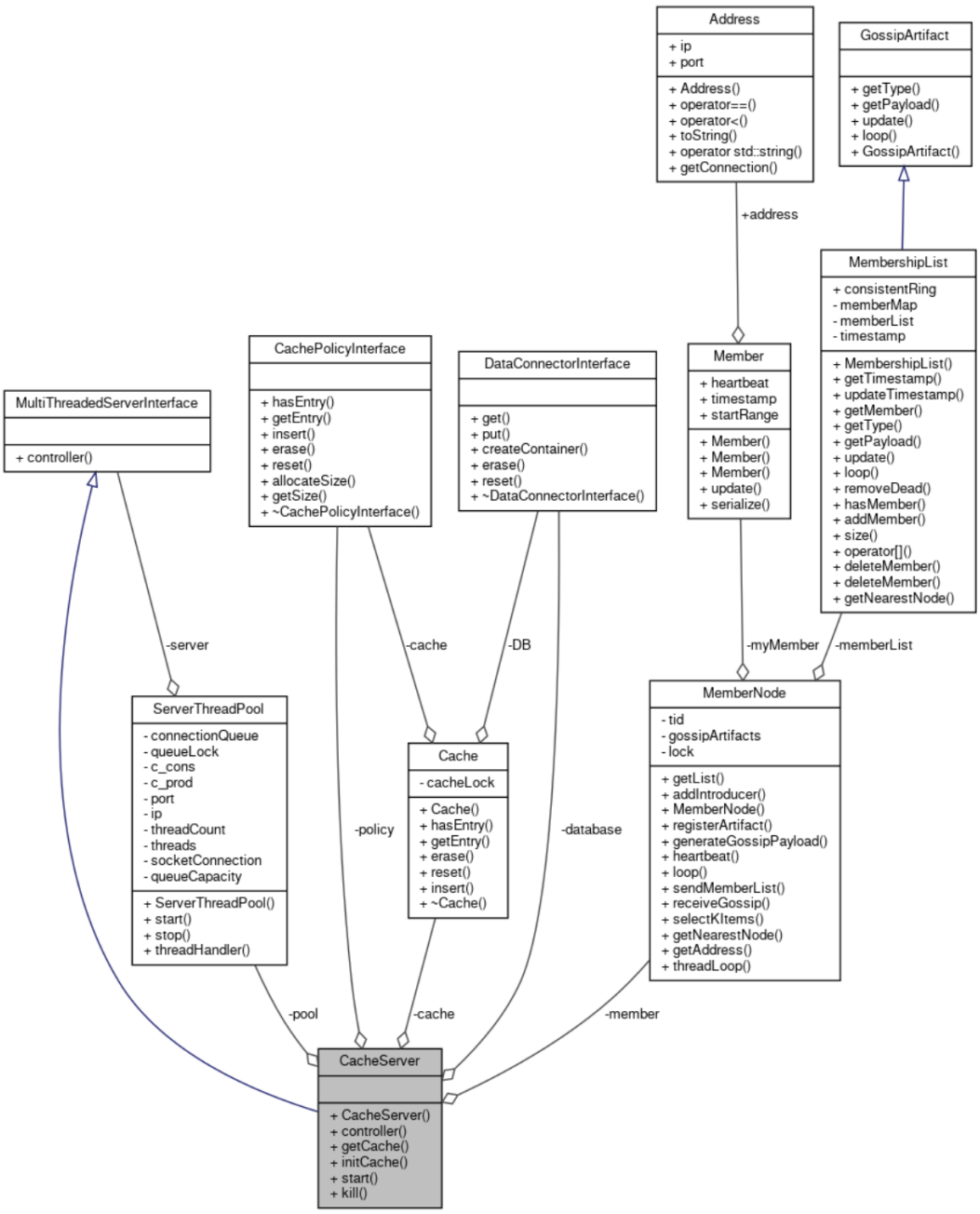

Figure 3.6: Big Picture

4 Experiments and Results

To run the experiments, I put a 50ms delay for the database and a 10ms delay for the cache system irrespective of the cache hit or miss. There was also around 3ms delay per query on the network.

Therefore for a single node system, If there was a cache miss, the response time was around 65ms. If there was a cache hit, the response time was around 13ms.

For a distributed cache system, if there was a cache miss, the response time was around 70ms. If there was a cache hit, the response time was around 15 ms. There is a slight overhead in a distributed node system as compared to a single node system because there is an extra hop involved in processing the query. Another reason could be increased network load.

The following is the graph of the response time as the queries progress.

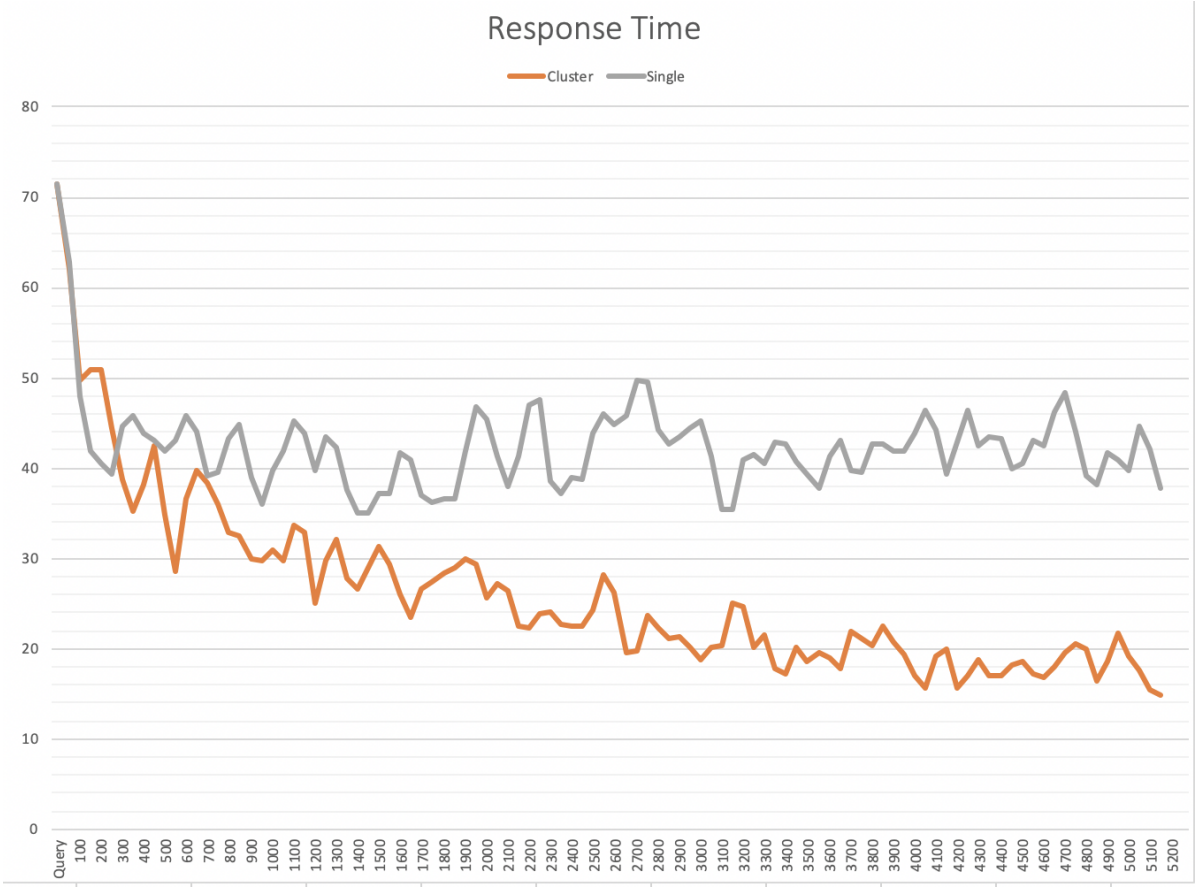

Figure 4.1: Graph of Result

We can see that the single node system stagnates relatively quickly in performance. However, the distributed system continues to improve.

5 Conclusion

5.1 Summary

I have successfully designed and implemented a distributed cache system. I have used several distributed concepts such as Gossip Protocol, Membership Lists, Sharding and Consistent Hashing. I have managed to demonstrate an improvement in performance in a distributed cache compared to a single node cache. I have also learnt how to work with threads, mutexes to write efficient multi-threaded programs in C++.

5.2 Success Factor

I had initial success in my system. However, once I started running experiments by firing queries in parallel instead of sequentially one by one, the system got stuck. The whole system became unresponsive for a while and then automatically resolved itself.

Lot of effort was put into drilling down on the issue and fixing it but didn't have much success.

The whole system went to a halt without any error, therefore I first suspected that some sort of deadlock was happening. However, it always seems to "recover" on its own after sometime. But still I went through the whole code to see if anything is missed. Also to confirm that deadlock wasn't the issue, I profiled the amount of time it took to acquire a lock, it was always < 3ms. So I concluded deadlock wasn't the issue.

I noticed that some network calls were blocking, so I profiled network operations, the read/write message into socket was always fast < 2-3 ms. However, the getSocketConnection() suddenly took so much time and it even timed out. Now either this was a network issue or the server was not accepting a request because it was busy. I logged the server queue size and the number of used threads and this wasn't the case, after this it was a dead end, not sure what to do from here.

A lot of experiments and tests planned couldn't be executed due to this issue.

Event though this project met a roadblock, I wouldn't call it a failure, I learnt a lot. I implemented the gossip protocol and the consistent ring successfully. The basic distributed LRU cache works properly as well. This was the first time I did a project with significant multithreading and networking aspects. Apart from that I also learnt a lot of C++ and got a chance to use some design patterns as well. The gossip protocol itself seems to be extremely well implemented and worked well even when this issue happened.

5.3 Future Work

In this project, I implemented some algorithms, concepts and techniques to create a distributed cache. A lot of experiments can be run to test the robustness of the system during failure and network partition. Further experiments can be done on to measure performance of difference cache strategies in different conditions. On a side note, while I have understood the distributed algorithms and concepts well, I have a lot to learn on how they are actually implemented in real systems. For this, I should study the code of open source products like Cassandra and Reddis.

References

[1] Boost documentation. https://www.boost.org/doc/.

[2] Caching. https://en.wikipedia.org/wiki/Cache_replacement_policies.

[3] Cloud computing concepts 1 - gossip protocol. https://www.coursera.org/learn/cloud-computing/lecture/5AOex/1-2-the-gossip-protocol.

[4] Cmake documentation. https://cmake.org/documentation/.

[5] Consistent hashing. https://en.wikipedia.org/wiki/Consistent_hashing.

[6] Database sharding. https://www.digitalocean.com/community/tutorials/understanding-database-sharding.

[7] Distributed caching. https://www.youtube.com/watch?v=U3RkDLtS7uY&t=16s.

[8] Gossip and epidemic protocols. http://disi.unitn.it/~montreso/ds/papers/montresor17.pdf.

[9] Gossip protocol. https://en.wikipedia.org/wiki/Gossip_protocol/.

[10] V. W.-H. Luk, A. K.-S. Wong, C.-T. Lea, and R. W. Ouyang. Rrg: redundancy reduced gossip protocol for real-time n-to-n dynamic group communication. *Journal of Internet Services and Applications*, 4(1):14, May 2013.