In-Memory Distributed Key Value Store

Group Number: 11
Group Members: Abhishek Satyander Lingwal (**aslingwa**), Glen Ashwin Menezes (**gmeneze**), Vipul Kashyap (**vkkashya**)

Date: 11/27/2016

Abstract — The idea is to create a distributed in-memory key value storage system. Key value store gives us advantage in terms of speed of operation, Single Point of Failure (SPOF) avoidance, better support for large amounts of unstructured data, lower total cost of operation (sysadmins) and incremental scalability over traditional structural databases. Structural databases lack these features as they emphasize on ACID properties which makes them slow and limits scalability. In-memory key value stores have applications for scenarios such as maintaining transient data (e.g. session info), caches and message broker. Our approach was inspired by Amazon's Dynamo paper [7] and Memcached [8].

1. INTRODUCTION

Distributed Key Value Store has become an important component of most companies' operations. It's an integral part of their platform to improve performance, reliability, efficiency and scalability. For example, Amazon's platform serves millions of customers at peak times using thousands of servers located in many data centers around the world. To support continuous growth, the platform needs to be highly scalable and reliable. Any down time will have significant financial consequences. Hence there is always need for storage technology that are highly available.

In this paper, we present design and implementation of a simple Distributed In-Memory Key Value store which is scalable and highly available. It has many use cases, Amazon, for example uses such a store [7] to manage the state of services that have very high availability requirements and need tight control over the trade-offs between availability, consistency, cost-effectiveness and performance.

Most traditional production systems use relational databases. However, many services provided now-a-days only store and retrieve data by primary key and do not require the complex querying and management functionality provided by RDBMS. These traditional systems typically require expensive hardware and very skilled personnel to operate, making them inefficient. In addition, the available replication technologies for RDBMS are limited and choose consistency over availability. Also, it is not easy to scale-out databases or use smart-partitioning schemes for load balancing.

Our solution uses a synthesis of well-known techniques to achieve scalability and availability. Data is partitioned and replicated using consistent-hashing, eventual consistency is achieved by versioning the objects using time-stamps. It is a completely decentralized system with load-balancing achieved by a proxy server. Storage nodes can be added or removed without requiring manual partitioning or redistribution.

2. ARCHITECTURE AND IMPLEMENTATION

a. *System Assumptions and Requirements:*

As is the case with any system design, there were a few assumptions and tradeoff decisions to keep in mind while implementing our solution. These assumptions were guided by the challenges and opportunities offered while keeping in mind the priorities and feasibility of the project.

i. *Query Model*:
Simple read and write operations to a data item that is uniquely identified by a key. No operations span multiple data items and requires no relational schema. Most services using this data store would need it to store large quantities of small data.

ii. *ACID Properties*:
ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are carried out reliably. Transaction, in database vocabulary, is a single logical operation on the data. Generally, data stores that provide ACID guarantees tend to have poor availability [6]. Our system operates with weaker consistency as trade-off for higher availability. Our data store provides only single key updates at a time.

iii. *Node capacity*:
We assume homogeneity among nodes in terms of memory size. This makes our load distribution algorithm simpler as we don't need to factor in the individual capacity of each node.

iv. *Network reliability*:
The operation environment is assumed to be non-hostile and there are no security-related requirements like authorization and authentication.

v. *Node Deletion*:
Failure of a node does not delete it from the ring structure (explained in Partitioning Algorithm section below), however it has no impact on performance or availability.

b. *System Architecture*

i. *Overview*:

Fig. 1 gives an overview of the System architecture. It contains of several components: Client, Proxy Server, Request Coordinators and Memory Nodes.

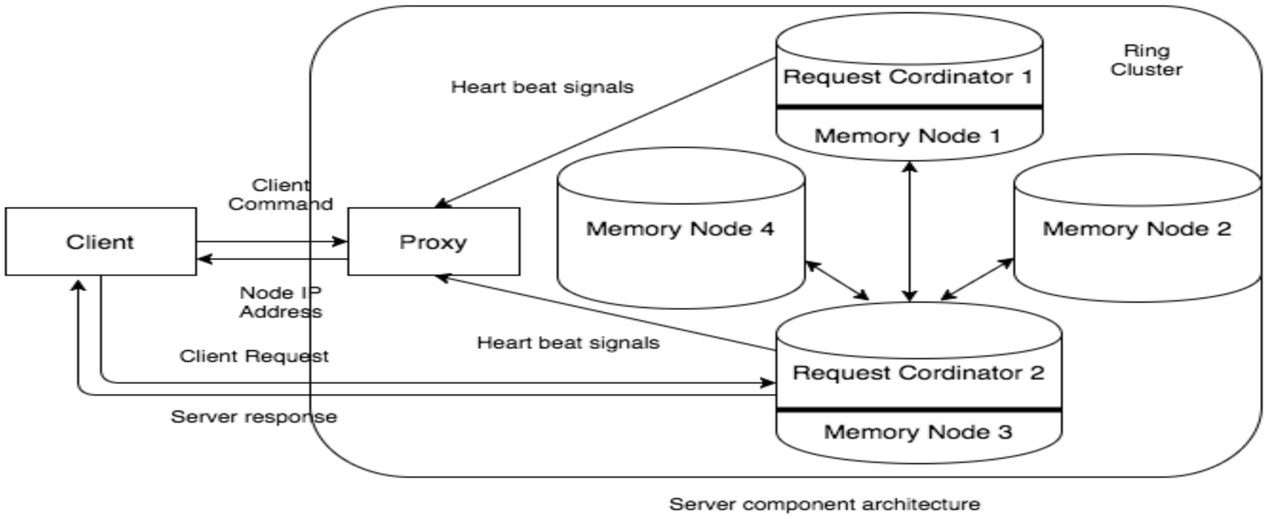

Fig. 1: System Architecture

Fig. 2 shows how these components interact with each other for a sample get/put request under normal circumstances.

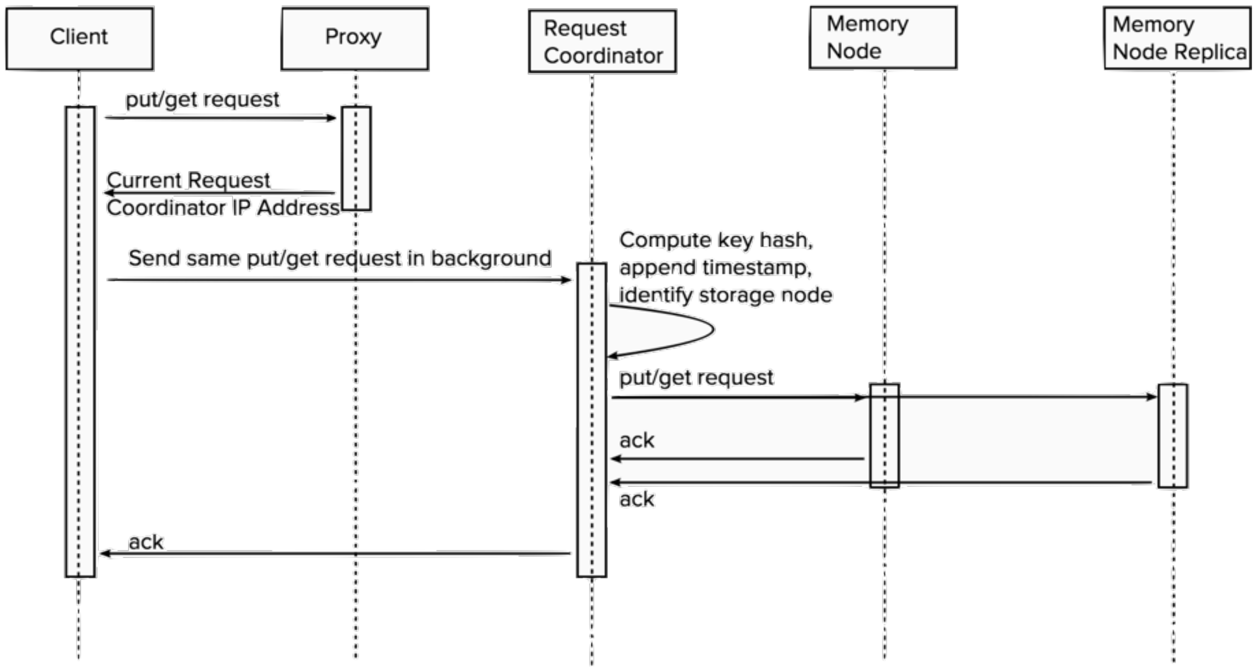
Client
Proxy
Request Coordinator
Memory Node
Memory Node Replica
put/get request
Current Request Coordinator IP Address
Send same put/get request in background
Compute key hash, append timestamp, identify storage node
put/get request
ack
ack
ack

Fig. 2: Sample Sequence Flow

Client sends the get or put request to the Proxy server to retrieve the IP address of the request coordinator. Primary purpose of request proxy server is to balance the load of incoming requests to all the coordinators in a round-robin fashion. Upon receiving the IP of the current request coordinator, client now sends the request directly to the coordinator. Note that this process happens in the background and not visible to the client. This is done so that proxy server only does the work of load balancing.

Request coordinator now computes hash from the key to determine which memory node to choose from the ring structure (explained below) to store this key value pair. The request coordinator then sends the request to the primary memory node and to it's next N replicas in the ring at the same time. We follow quorum protocol to acknowledge successful read or write operation. After successful acknowledgement from all at least W or R memory nodes, request coordinator acknowledges the client about the success or failure of the request.

ii. System Interface

A client can interact with system via simple REST calls. A get request with the key would fetch the value for that key and a post request with a key value pair would set the key in the memory node. If a post request with an existing key is set, then it will overwrite the previously set value. These requests can be sent by simple curl commands or by using utilities like Postman. To make the task of sending these requests easier, we've integrated the client with Slack so that requests can now be sent by using Slack commands instead of typing commands on terminal.

iii. *Partitioning Algorithm*

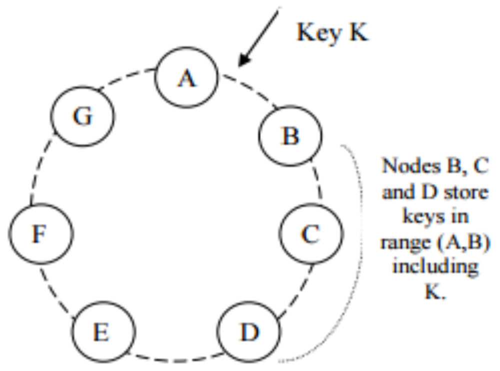

Fig. 3: Ring Structure: Insert key

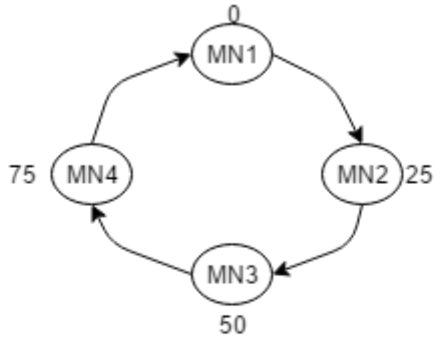

Fig. 4: Ring Structure: Node placement

One of the key design requirements for our solution was that it must scale incrementally. This requires a mechanism to dynamically partition the data over the set of nodes (i.e., storage hosts) in the system. Our partitioning scheme relies on consistent hashing to distribute the load across multiple storage hosts.

In consistent hashing [9], the output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a value within this space which represents its "position" on the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position. Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. The principle advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors and other nodes remain unaffected.

However, since consistent hashing randomly assigns positions to nodes in the ring, it leads to non-uniform data and load distribution. This problem is addressed using an approach similar to Dynamo [7]. We try to assign values to nodes such that the system load (range of values) is distributed among the nodes as evenly as possible.

As shown in figure x, if there were 4 nodes in the system and the range of key hash values generated by the hash function were between 0 to 100 the nodes would be assigned values such that each node is responsible for1/4th (or 25) values.

iv. *Replication*

To achieve high availability and durability, our system would replicate the data N-way in our systems its 2 replications. By replicating the data, we make sure the system can handle 2 node failure for data loss. When a put request is made the system [Request Co-Coordinator] makes a put request to 3 nodes in total. The nodes to which the put request is being made is decide on the node selected by the consisting hashing technique. The next two nodes in the ring serve as a replicating node which will store the same data. Hence is X node was selecting by the hashing system then X+1 and X+2 serve as the node which will store this same copy of data

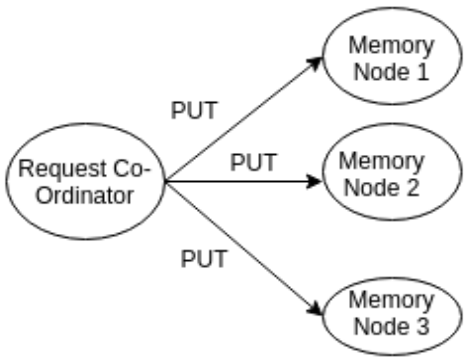

Fig. 5: Replication mechanism

for fault tolerance. When a get request is made all copy i.e. original and its replicating copies are requested. Based on the copies received it takes the latest time stamp copy. Thus, even one copy is corrupted or destroyed system is still able to serve the request using replicated copy.

v. *Consistency*

Our system guarantees eventual consistency. If the data is written across various replicated copies it will be consistent till any write is failed or node is not able to retrieve the stored key value from memory. If the write is corrupted for any reason the copies will be in inconsistent state. Hence forth any get request made will cause different copy to be retrieved from the data store. It will pick up the latest copy and serve the client. At the same time, it will send back the request to the old copy of data to be updated with the latest copy. Thus, resulting in the consistent state eventually.

vi. Auto Clean-Up

To avoid stale data in the system we have added the feature of cleaning up stale key value pair from the system which are older than pre-defined time from the system. This ensures auto system cleanup and avoid old unused data so that new data which are getting used can be served. For this purpose, a Time-Based Order List is maintained where head of the list contains recently used value while tail contains older data.

vii. *Addition/Removal of Nodes*

The system tries to efficiently redistribute the load whenever a new node is added. The ideal way to distribute load would be to reposition all nodes such that each node needs to handle equal set of values in the ring. But this process would need a lot of redistribution processing per node. This will take a hit on system performance especially when nodes are frequently added.

The tradeoff decision we made was to ensure that only the new node needs to sync up with neighbors whenever it is added to the system, all other nodes continue to function as is. The new node is placed at the midpoint between the pair of nodes separated by the maximum distance in the ring. This is because this pair of nodes handle the most load in the system and

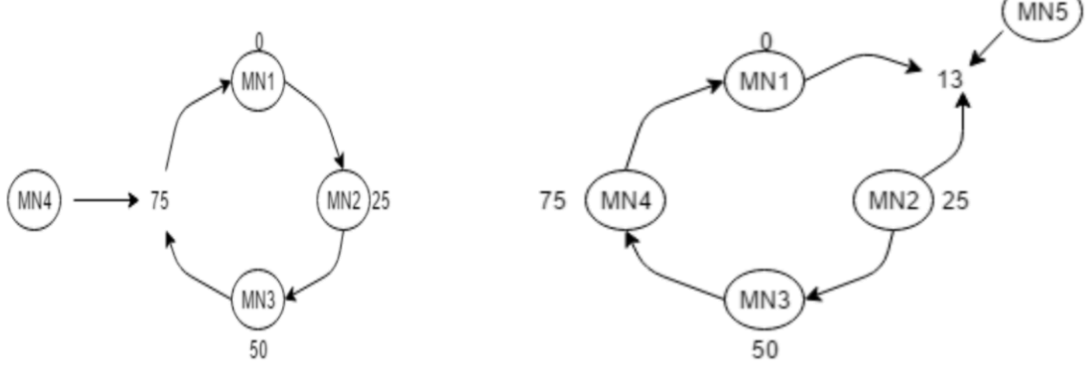

Fig. 6: Ring Structure - Insert node

the addition of new node effectively halves their load. This can be seen in the two examples in Fig. 6 above.

In the first example, nodes MN1 and MN3 handle the most load, so the new node (MN4) will be added such that the range of values handled by the two nodes (50-100) is halved. In the second example, all nodes handle equal range of values. Thus, MN5 partitions the first pair of nodes.

c. *Results*

Test Case Scenario	**Expected Result**	**Status**
Client sends a first time put request	Request is successfully processed with client data getting stored in the memory nodes	PASS
Client sends a get request for data present	Client request should get successfully processed and pass the value back	PASS
Client sends a get request for data not present	Client request should get successfully processed and return appropriate message to show no key found	PASS
One of the memory node fails which contains the client data	Client request is still served as this data is replicated across different memory nodes	PASS
One of the request co-ordinator fails.	Proxy which receives back the heart beat result will successfully remove this from its list.	PASS
A new memory node is added to the system	Successfully addition along with this propagation back to Request co-ordinator and the added memory node loading relevant data.	PASS
Data become inconsistent due to write corrupted or update failed.	Request co-ordinator will identify this and sends back a request to update the specific key to the memory node.	PASS

3. RELATED WORK

a. *Semi-structured data, peer to peer configuration*

There are several peer-to-peer (P2P) systems that have worked upon the problem of data storage and distribution. The first generation of P2P systems, such as Freenet and Gnutella, were used as file sharing systems. These were examples of unstructured P2P networks where the overlay links between peers were established arbitrarily. In these networks, a search query is usually flooded through the network to find as many peers as possible that share the data. P2P systems evolved to the next generation into what is widely known as structured P2P networks. These networks employ a globally consistent protocol to ensure that any node can efficiently route a search query to some peer that has the desired data. Some systems like Pastry [1] and Chord [2] use routing mechanisms to ensure that queries can be answered within a bounded number of hops. To reduce the additional latency introduced by multi-hop routing, some P2P systems employ O(1) routing where each peer maintains enough routing information locally so that it can route requests (to access a data item) to the appropriate peer within a constant number of hops. Various storage systems, such as Oceanstore [3] and PAST [4] were built on top of these routing overlays. Oceanstore provides a global, transactional, persistent storage service that support serialized updates on widely replicated data. To allow for concurrent updates while avoiding many of the problems inherent with wide-area locking, it uses an update model based on conflict resolution. Conflict resolution was introduced in [5] to reduce the number of transaction aborts. Oceanstore resolves conflicts by processing a series of updates, choosing a total order among them, and then applying them atomically in that order. It is built for an environment where the data is replicated on an untrusted infrastructure. By comparison, PAST provides a simple abstraction layer on top of Pastry for persistent and immutable objects. It assumes that the application can build the necessary storage semantics (such as mutable files) on top of it.

b. *Master Slave configuration:*

Redis is an in-memory key value data store and supports master-slave asynchronous replication. Redis provides simple APIs to communicate with redis server which acted as an inspiration for our client side implementation.

4. CONCLUSION

This paper described a distributed in-memory key value data store which achieves desired level of availability and performance which successfully handles node failures and network partitions. It is incrementally scalable allowing even distribution of load. What we have achieved to show in this paper is that eventually consistent storage systems can be a building block for highly-available applications.

5. ACKNOWLEDGEMENT

We would like to thank Professor Vincent Freeh for his support throughout the semester for providing us with this exciting opportunity to learn something interesting. We are also grateful to authors of Amazon's Dynamo paper which acted as inspiration for our project.

6. REFERENCES

[1] Rowstron, A., and Druschel, P. Pastry: Scalable, decentralized object location and routing for large-scale peerto-peer systems. Proceedings of Middleware, pages 329-350, November, 2001

[2] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications (San Diego, California, United States). SIGCOMM '01. ACM Press, New York, NY, 149-160

[3] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Wells, C., and Zhao, B. 2000. OceanStore: an architecture for global-scale persistent storage. SIGARCH Comput. Archit. News 28, 5 (Dec. 2000), 190-201.

[4] Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. Proceedings of Symposium on Operating Systems Principles, October 2001.

[5] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (Copper Mountain, Colorado, United States, December 03 - 06, 1995). M. B. Jones, Ed. SOSP '95. ACM Press, New York, NY, 172-182.

[6] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. 1997. Cluster-based scalable network services. In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (Saint Malo, France, October 05 - 08, 1997). W. M. Waite, Ed. SOSP '97. ACM Press, New York, NY, 78-91

[7] Dynamo: Amazon’s Highly Available Key-value Store Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

[8] https://memcached.org/

[9] 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.