Distributed Key-Value Store Utilizing CRDT to Guarantee Eventual Consistency

CPSC 416 Project Proposal

n6n8: Trevor Jackson, u2c9: Hayden Nhan, v0r5: Yongnan (Devin) Li, x5m8: Li Jye Tong

Introduction and Background

Ever since the CAP theorem was conjectured, practitioners of distributed systems in academia and industry have evaluated various ways of making trade-offs between consistency, availability, and network partition tolerance to strive for the best possible user experience. Internet scale data storage systems aim for high availability to serve a large number of concurrent requests, while being resilient to network partitioning caused by failures of various hardware/software components. Amazon's highly available Dynamo key-value store is a prime example of such a distributed system. Amazon engineers chose the eventually consistent model to provide a reliable "always on" experience to the users. Sacrifices in data consistency will manifest under certain usage scenarios. For instance, in their 2007 publication, the authors mentioned an anomaly in the Amazon shopping cart in which deleted items can reappear under certain conditions [1]. Although these anomalies are very rare and can always be corrected by the user, customer experience can be improved by minimizing these kind of anomalies.

Distributed systems use optimistic replication to achieve better availability and performance. Updates to data can be made at each replica without synchronization with other replicas. Conflicting updates at different replica are usually resolved through a background consensus algorithm and some updates may need to be rolled-back. The downside to this approach is that conflict resolution mechanism can be complex and error-prone. In the distributed systems literature, the concept of Conflict-free Replicated Data Type (CRDT) has been proposed to help with this problem. CRDT is a theoretical construct that ensures eventual consistency by design without the use of consensus. CRDT presents a theoretical-sound approach to eventual consistency that promises to remove the complexity in conflict resolution in existing implementations. However, the level of consistency can be weak. More specifically, CRDT will not provide sequential consistency which is often desired by application developers [2].

Overall Approach

We propose to implement an always writable key-value store inspired by Amazon Dynamo that uses CRDT to perform asynchronous background data reconciliation. The key-value store is intended to run over multiple nodes. This means group management will present challenges. To limit the scope of the project, we need to make simplifications to the

Dynamo architecture presented in reference [1] (similar to the ones made by the authors of reference [3]). In particular, we plan to address the following challenges:

Challenge	**Technique**	**Benefit**
Membership/failure detection	Gossip Protocol	Detection at run-time
Load Balancing	Consistent Hashing	Increased scalability
High Availability for Writes	CRDT	Simple data reconciliation
Node Failure and Rejoining	See “Node Failure and Rejoining” Section	Higher availability

By addressing these challenges, our implementation will contain no single point of failure, have scalable storage capacity, and guarantee eventual consistency. The main goal of our project is to evaluate the efficacy of using CRDT as a simple and coherent mechanism for data reconciliation when client writes are in conflict. Current production systems typically have complex data reconciliation mechanisms. The successful application of CRDT will provide a simpler, alternative mechanism. We will demonstrate our implementation using a collective ToDo/notepad application.

System Architecture

Gossip Protocol

In order to detect active nodes both at the initial start-up of the system and as the system continues to function, a gossip protocol will be used. The protocol, based on gossip that occurs in social networks, will have each node choose another node in the system at random, at set intervals. A node will then communicate all the nodes that it knows are currently inactive and active with the randomly selected node. Should the selected node fail to respond, it is marked as inactive. In this way, information about inactive nodes is passed from one node to the next until it has permeated the entire system. If an inactive node becomes active again, its presence will be initially identified by each node it randomly selects and communicates with.

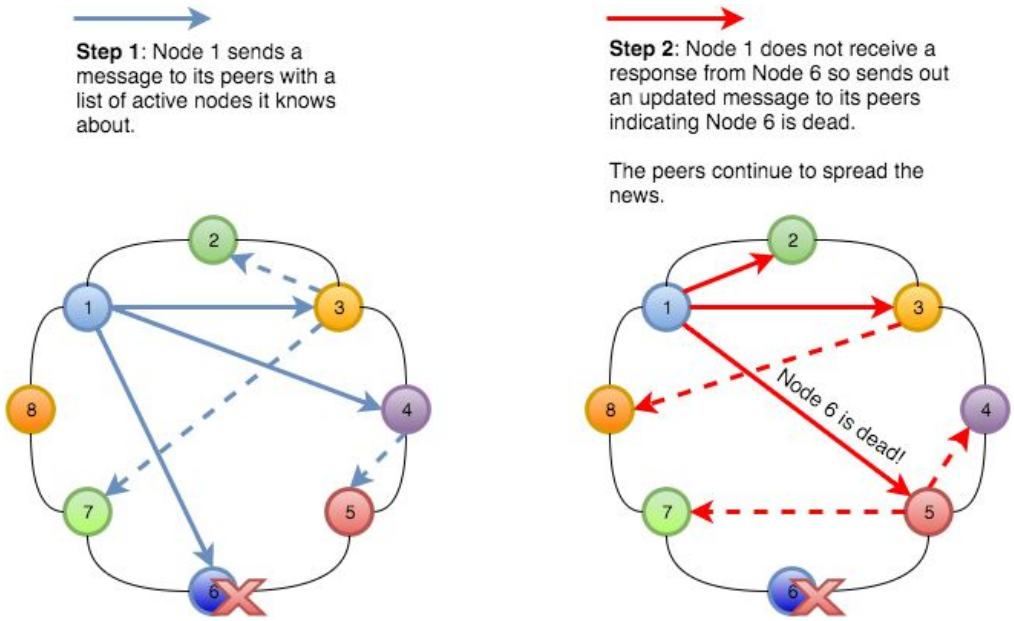

Figure 1: *The Gossip Protocol implemented across the network. Initial messages are solid lines while subsequent messages echoed are dashed lines.*

Consistent Hashing and Replication

Consistent hashing is a powerful mechanism for load distribution in a distributed environment. Any given key can be hashed to a position in the ring. The ring will be divided into partitions of equal size and each node in the system will be assigned roughly the same number of partitions to store. Replication factor N is a parameter of the system configuration on start-up and it means a key belonging to a given node will be replicated to its N-1 successor nodes in the ring. Nodes will have a unique identifier, taking on a value of “Node#”, # being sequential positive integers starting from 1.

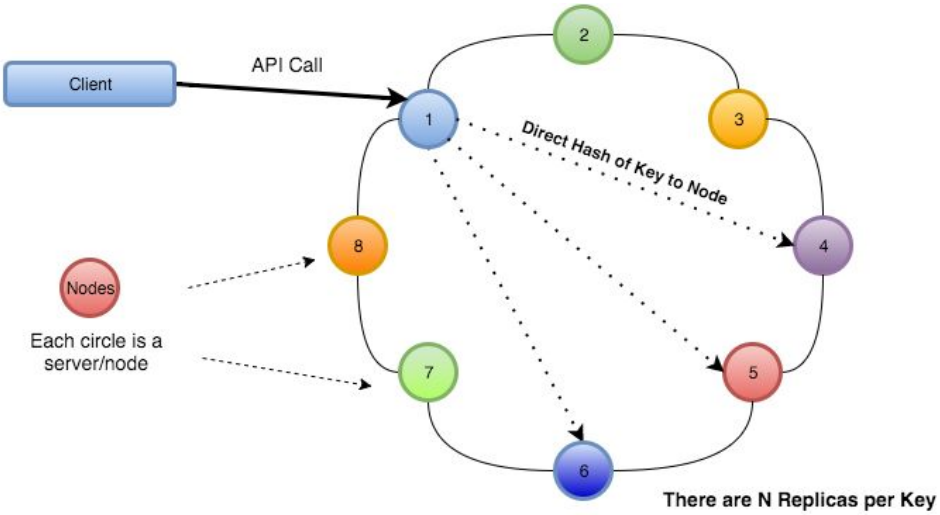

Figure 2: *The client sends a call to Node A which uses consistent hashing to assign the keys to specific nodes (D, E, F).*

Replication happens asynchronously in the background to increase the availability of the system to client requests. This design decision sacrifices strong consistency for high availability but the system can achieve strong eventual consistency with the use of CRDT. The use of CRDT guarantees eventual consistency and simplifies data reconciliation process in cases when write operations are in conflict.

Node Failure and Rejoining

In the event that a node fails, the remaining nodes with the failed node's keys will replicate the failed node's key value pairs among the remaining nodes based on the replication factor. Re-replication of the key value pairs will be performed by incrementing over the ring of nodes and copying over key value pairs as the system travels over the ring.

The Consistent Hashing algorithm will not be affected by a failed node. If a node attempts to access/modify a key value pair on a failed node, it will instead place them on the next available node using the same system as above. If a failed node rejoins, the keys that it previously held will be assigned to it to rebalance the nodes.

For example, Node 1, 2, and 3 have the same key value pairs and the replication factor is 3. Node 2 and 3 die, Node 1 upon detecting the failures will attempt to replicate its key value pairs to the next node, Node 4. Node 1 will then replicate its key value pairs to the next node, Node 5. There will now be 3 nodes with the key value pairs, satisfying the replication factor. (See Figure 3 and 4).

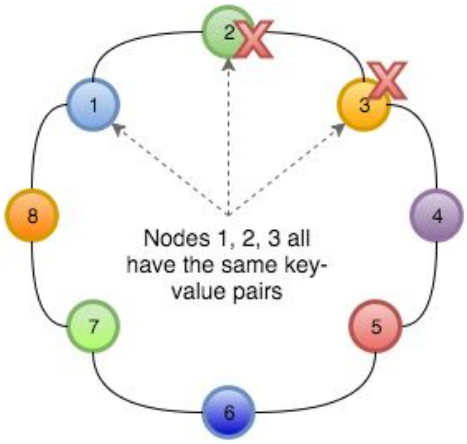

Figure 3

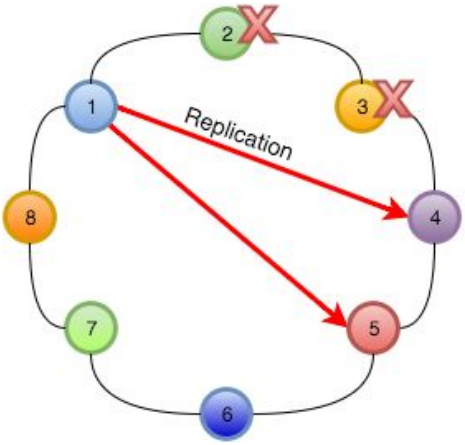

Figure 4

To continue our example from previously, Node 7 attempts to retrieve a key value pair on the failed Node 2, however, Node 7 has learned via the Gossip Protocol that Node 2 has died. Node 7 instead will try to retrieve the key value pair on Node 3. If Node 3 were to also have died, Node 7 would learn of the death and attempt to access the next Node, in this case Node 4. This process of checking the next Node would continue until either Node 7 ran out of nodes to check or it found a node to return the key value pair. (See figure 5)

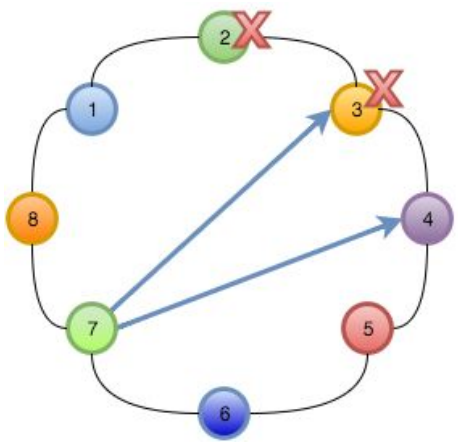

Figure 5

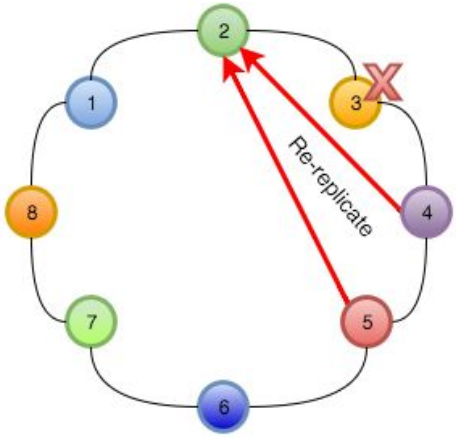

Figure 6

In the case where Node 2 were to rejoin, each Node in the system would learn about the rejoining via the Gossip Protocol and then run a check on its own key value pairs. The check would run the Consistent Hashing algorithm on the node's own keys and then determine if that key value pair could be moved to the new rejoined node. (See figure 6)

CRDT Usage in Our System:

The set data structure can be extended into a CRDT by attaching a unique identifier to each element. As illustrated in the figure below, the addition of an element will put a unique identifier (i.e. a_α is unique). The removal of an element will remove all elements known to the source replica (i.e. the blue replica in the figure). This is the so called Observed Remove Set (OR-Set). By ensuring the addition of an element is always delivered before removal of the same element, CRDT properties of OR-Set guarantees eventual consistency.

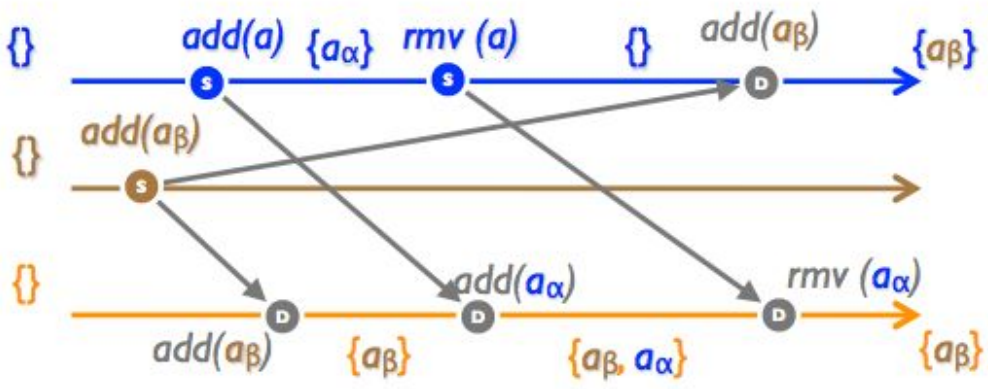

Figure 7*: Operation-based OR-Set. Operations on the three replica propagate and eventually reach the same state with the set containing one element. Adapted from reference [2].*

Application to be Built Utilizing Our System

The application that will utilize our key-value store will be a collaborative ToDo list. In a similar vein to Google Docs, our app will be able to handle multiple users editing the same document from different machines and locations. We will restrict the list of ToDo objects to only be strings up to a fixed maximum length. Each numbered object will correspond to a key (ToDo1, ToDo2, ToDo3, etc.), while their string description will correspond to their value to be stored. Each application running will act as a client for our server nodes to communicate with. With the usage of our key-value store, multiple users accessing and changing the same list will be able to do so concurrently while also receiving the changes made by the other users.

In order to access the key-value store, the client(s) will be able to use the follow commands:

- **get(key)**: Send a request to the server to retrieve the value associated with the key given
- **add(key, val)**: Send a request to the server to store the key value pair given
- **update(key, old_val, new_val)**: Send a request to the server to check the value associated with the key, if there is a match with the old value given, replace the old value with the new value.
- **remove(key)**: Send a request to the server to remove the key value pair stored

Assumptions

Assumptions we can make:

1. Nodes can be trusted.
2. All possible Node membership is agreed upon and known by every Node before runtime.
3. Network failures do not occur.

Assumptions we cannot make:

1. Nodes are synchronized together.
2. Nodes will communicate failures.
3. Nodes will not rejoin the system after failure.

Testing Plan

- The client API will be tested by having a client issuing a series of commands (listed above)
- Two clients will be used to issue concurrent writes to the same key on different nodes in the system.
- We will test the system for node failures by:
    - force killing nodes and observing that keys are replicated correctly
    - rejoining killed/nodes and observing that keys are re-assigned back to this original node
- GoVector and ShiViz will be used to visualize the behaviour of our system.

Timeline

March 3	Specifics of the API design, protocols, languages, and frameworks worked out. Learn about GoVector and ShiViz.
March 10	Implement the Gossip Protocol
March 17	Complete CRDT and Consistent Hashing
March 18	Project update to TA
March 24	Replication and Node Failure/Joining
March 31	Testing with GoVector and ShiViz
April 7	Build Application and Testing
April 11	Final Report and Final Changes

SWOT Analysis

Strengths	Weaknesses	Opportunities	Threats
- All team members are comfortable with programming in golang - Application will use tools which team members are very proficient and comfortable with. - Team members have flexible schedules to allow more accessible meeting times	- Assignments, exams, other projects, and other commitments can hinder the completion of this system - No one knows how to use golang unit test framework - We need to learn to use GoVector and Shiviz. - Scope of project may be too large for us to fully implement	- Implement a system to resolve a current issue, and create an application to utilize it - Many tutorials for building Restful APIs in golang - Possible Gossip Protocol library, may be useful. - Consistent Hashing Protocol library may be useful.	- Tight time constraints - The CRDT proposed was taken from a highly theoretical paper and may take a great deal of time to understand and implement. - Golang is a relatively new language, future changes may cause our system to become deprecated

References:

1. DeCandia, G., et al. Dynamo: Amazon's Highly Available Key-value Store. www.allthingsdistributed.com/2007/10/amazons_dynamo.html accessed Feb 26, 2016.
2. Shapiro, M. et al. A comprehensive study of Convergent and Commutative Replicated Data Types. http://hal.upmc.fr/inria-00555588/document accessed Feb 26, 2016.
3. Scott Sallinen and Brittany Roesch. SASSI Simple Available Scalable Storage Implementation. sample proposal 2 provided by instructor on piazza.