No(de) Conflicts. The Pacifist Approach To Distributed In-Memory Key-Value Store

Authors

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

Abstract

A primary property of good user experience in a shared collaborative editing program is high availability. As dictated by the CAP theorem, the focus on availability means a trade-off with consistency and network partition tolerance. In our report, we attempt to solve the problem of consistency in a distributed system that supports a shared To Do List key/value system. We apply the concept of Conflict-free Replicated Data Types (CRDT) to implement eventual consistency. We designed our system to have nodes that accept requests from clients. Our system is able to handle node failures and restarts. Finally, we developed a web application to handle multiple client inputs.

Introduction

Collaborating editing has become a valuable tool for people to work together remotely on shared resources (e.g. Google Docs). The challenge in developing these systems is the strategies to balance trade-offs between the three pillars of consistency, availability, and partition tolerance (CAP theorem). Practitioners of distributed systems in academia and industry have discussed many ways to do this in order to provide the best possible user experience. DeCandia et al's [1] paper on Amazon's highly available Dynamo key-value store and their design of an eventually consistent model became the motivation for us to design our system.

Our goal was to build a distributed system that can effectively manage the storage of key/value pairs with respect to node failures. We can demonstrate this by a To Do List web application that can be shared and edited by multiple users in a similar, but simplified, manner to that of Google Docs. We aimed to provide the best possible user experience by implementing a system that can be highly available and eventually consistent. The system will address conflict resolution by applying the concept of Conflict-free Replicated Data Type (CRDT) as discussed by Shapiro et al [2].

This system is capable of handling restarts and node failures based on a replication factor. Each node also starts an http server to handle incoming requests from a web client.

We apply the concept of CRDT and implement an Observed-Remove (OR) Map to resolve conflicts, sacrificing strong consistency to allow for high availability.

We designed a RESTFul To Do List web application in which multiple clients can get, add, and remove items from the list.

Assumptions

1. Nodes can be trusted
2. All possible Node membership is agreed upon and known by every Node before runtime.
3. Nodes can rejoin the system after a failure
4. Network failures do not occur.
5. Nodes are not synchronized based on any type of clock
6. Nodes will detect failures of other nodes.
7. Node names or IDs are positive integers or zero, with each node's ID being one larger than the previous one. The first node's ID is zero.
8. Nodes will not be in a partitioned system

Design

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.

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.

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 was used. 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 noticed as it communicates with nodes in the system. Since this method of communication requires information to be passed node to node, there is a delay between when a node becomes inactive and when other nodes in the system determine this. This delay had to be taken into account when designing the rest of the system, specifically we rely on replicas to provide the information of the failed node. We used the memberlist library to implement this feature.

Node Failure

In the event that a node fails, the gossip protocol will eventually discover it through lack of communication from the node. Each node calls NotifyLeave() once they deem the node has died, and within this function the failed node is assigned to a map of inactive nodes. A proxy node is chosen with the following: (failed node ID + replication factor), and is stored in a proxy node map. The proxy node is used to redirect requests originally intended for the failed node to one that is currently available. The nodes will also perform a check to see if the proxy node chosen is in the inactive nodes map, and if that is true, it will continue iterating through the hash ring until it finds an active node.

The choice of using this method to handle node failure provides a simple and effective way to redistribute work among the remaining nodes, but the impact of doing so is finding proxy nodes in a larger system, say with 1000 nodes, can take a much longer time to perform if the system must iterate through many nodes. Another impact of this design choice would be that the workload may not be evenly distributed between all nodes, as some of the nodes may take more traffic than others.

Node Rejoining

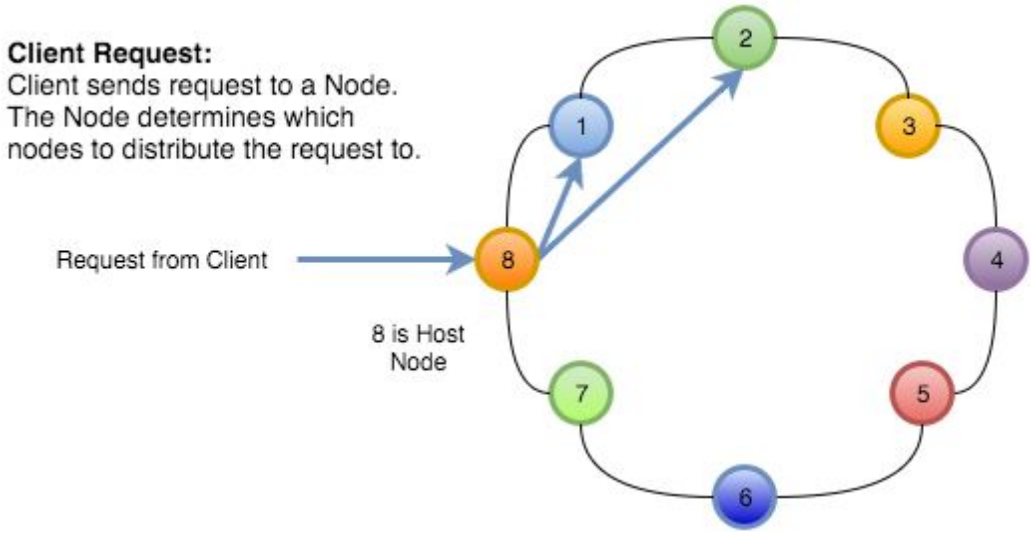
When a node (the calling node) initially starts up (assuming parameter three is 1), this calling node will determine which nodes are its replica(s) using its id, the replication factor and the list of nodes in the system. The calling node will then contact these replicas (or their alternatives if the replica node has failed, see Node Failure above). The replica nodes receive this request and search through every key in their ORMap, checking if the key hashes to the calling node. All key value pairs that do hash to the calling node are then returned. As each replica returns the key value pairs that hash to the calling node, these pairs are merged with the calling node's ORMap.

When all replicas have returned (or the 5 second timeout has been reached) the calling node will contact the nodes in which it was the replica. Again the nodes id, replication factor and list of nodes in the system are used to determine this. These nodes will go through each key in their ORMap, checking if the key hashes to themselves. All key value pairs that do hash to themselves are sent to the calling node. As each node returns the key value pairs that hash to it, these pairs are merged with the calling node's ORMap.

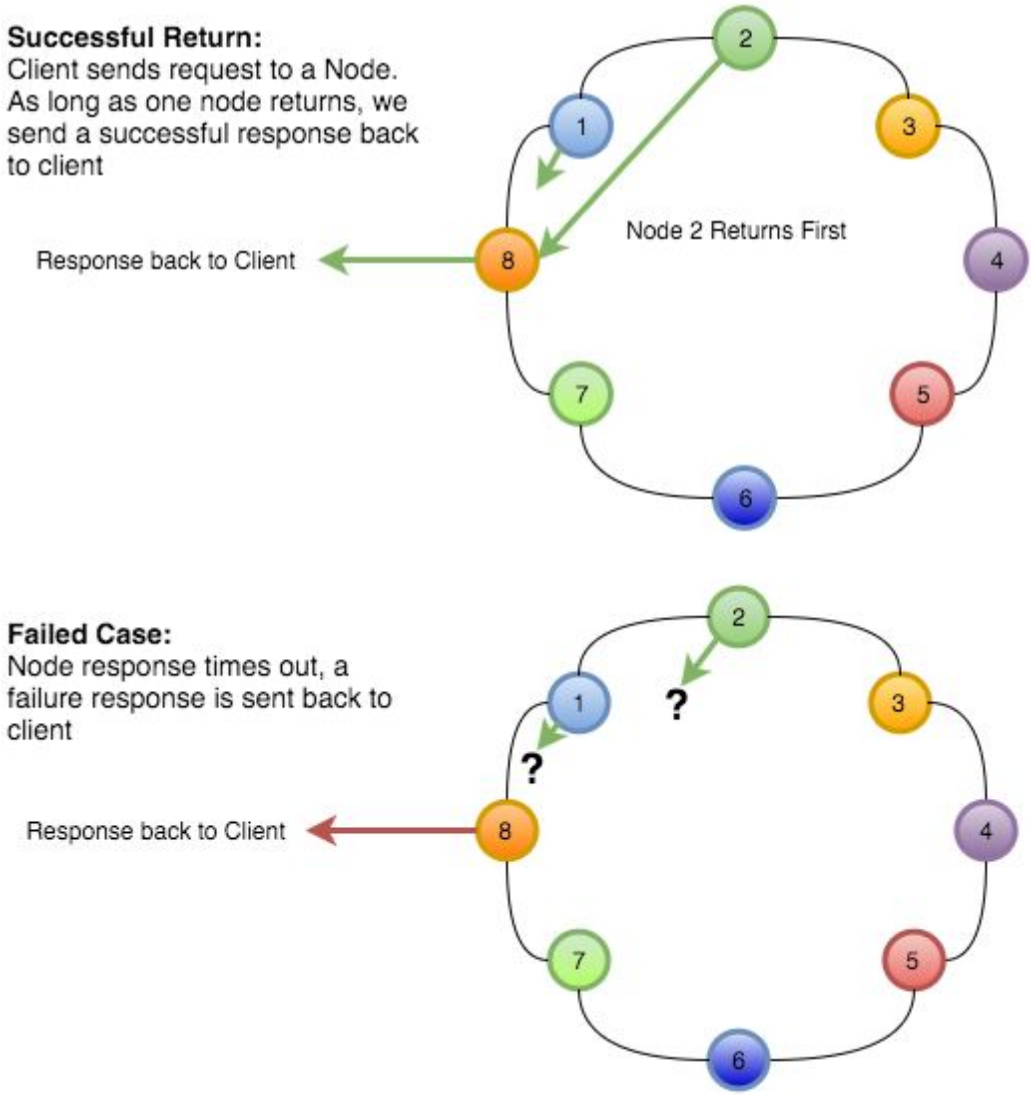
When a node rejoins, the other node's running the gossip protocol library: memberlist, will detect the rejoining. Memberlist will then call the NotifyJoin() function which will check if the new node is contained in the inactive node map. If the new node is found then it is removed from the inactive node map and from the proxy map.

Request Distribution Logic

For our system, we distribute work and requests among nodes by taking into consideration a replication factor that is set at system startup. Depending on the type of request received (put, get, or remove), the host node (node that client contacted) will call distribute() with an appropriate string to use as a switch case to execute the appropriate actions. The host node uses our consistent hashing function Find() to decide which node is the “owner” node of the key. Afterwards, a buffer is created to store which nodes the system will contact to perform the operations. Each node’s IDs are positive integers including zero, and our system increments the “owner” node ID up to (replication factor - 1) number of times. In the event of requests being mapped past the final node in the consistent hash ring, the request will be wrapped over and sent to the very first node. The host node will also check the chosen nodes against an inactive node map, and if they are found in that map, the proxy (alternative) node(s) for the failed node(s) is used instead.



After the nodes are added to the buffer the host node creates an appropriate request packet, and iterates through the buffer sending the request to each node in the buffer. Two go channels are then created, one to listen for the responses from the contacted nodes and one to send timeout responses. The host node waits for at least one successful response from the contacted nodes before sending a response back to the client. In the event that a timeout is reached, a failure packet is created and sent back as the response.

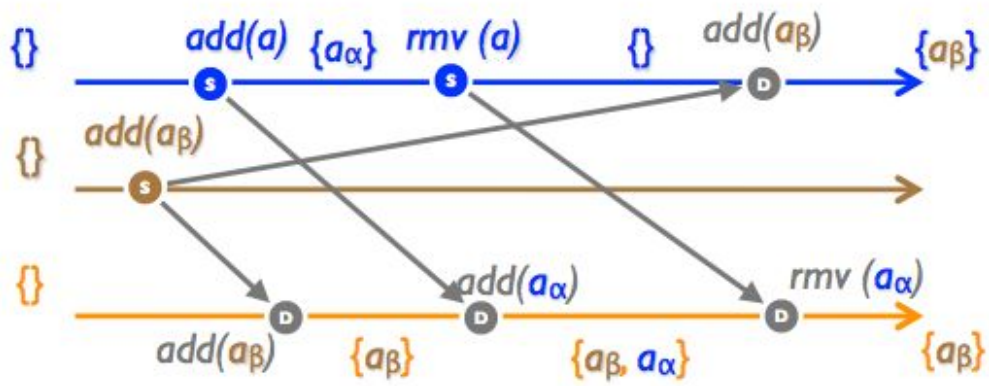


Using this method to distribute the requests among the nodes allows for faster replies to the client as we only have to wait for a single successful response. The design impact from this method would be that we have no guarantees that the other nodes were able to successfully store the keys as we are only sure the one that responded holds it. Another benefit would be that in the event that a node fails and we have a request for a key that it holds, we can always use its replicas to execute the request.

CRDT

The set/map 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 Map (OR-Map). By ensuring the addition of an element is always delivered before removal of the same element, CRDT properties of OR-Map guarantees eventual consistency. In our system, when an Add is performed, a unique tag is attached to the key and if the key is already known, the system does an observed removal. When a Remove is

done, if the key exists in the Add Map, it is copied to the Remove Map. In this scenario, Add actions take precedent over Removes and therefore, we recognize that a client may still get a key that it has removed.



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].

Client Facing API

The application that will utilize our key-value store will be a collaborative To Do 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 To Do objects to only be key/value strings up to a fixed maximum length. 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 can concurrently access the system using the follow API calls:

- **Add**(key, value): store a new key/value pair into the system
- **Get**(key): retrieve the value associate with the key
- **Remove**(key): removes the key and the value associated with it from the system

Implementation

General

The backend is written in Go while the frontend is written in Javascript, Jquery, Bootstrap, HTML

Backend

The backend consisted of a distributed system of key-value nodes which ran across a range of machines (generally 8). The entire backend was developed in the Go programming language with various third-party libraries from Github. Specifically we utilized the following:

Gossip Protocol

To implement this we used the memberlist library (https://github.com/hashicorp/memberlist), making changes to the NotifyJoin() and NotifyLeave() functions.

Consistent Hashing

The consistent hashing used by our system was based off of the groupcache library (https://github.com/golang/groupcache). We created our own version of consistent hashing to distribute the workload among nodes in our system.

GoVector

Logging in our system used this library to provide vector clocks and message logging for visualization using the ShiViz application. (https://github.com/arcaneiceman/GoVector)

Gorilla Mux

An http handler/router to handle client side requests.Used in our handler.go to deal with our ToDo list client sending HTTP requests to the server. (http://www.gorillatoolkit.org/pkg/mux)

Frontend

A simple front-end web client was developed using HTML and Javascript utilizing the jQuery library. The client uses RESTful Ajax calls to send and retrieve JSON data from the backend.

Evaluation

Testing Plan

- Bash scripts for individual get, put, and remove operation.
- Sequence like: put, get, remove, put
- Two clients was be used to issue concurrent writes to the same key on different nodes in the system.
- The client API was tested by having a client issuing a series of commands (listed above)
- We tested 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

For example the sequence of put, get, remove, and put operations are executed against the system (8 nodes). GoVector and ShiViz is used to visualized the UDP communication between all nodes. The key:value pair is logged in the message section of GoVector log. By visualizing the log with ShiViz we confirm the key:values are stored in the replica. The

desired effects of the sequence of operations are worked out ahead of time on paper and check against actual execution shown in the log.

The system was first tested on local environments with 8 process and then scaled up to run on 8 of the ugrad linux machines (e.g. lin10.ugrad.cs.ubc.ca).

Integration of ShiViz

GoVector logs all communication between nodes except for communication by the memberlist Gossip Protocol library. Also all keys and values that are being sent by nodes are also logged.

GoVector and ShiViz was used to visualize the behaviour of our system. In particular, the value

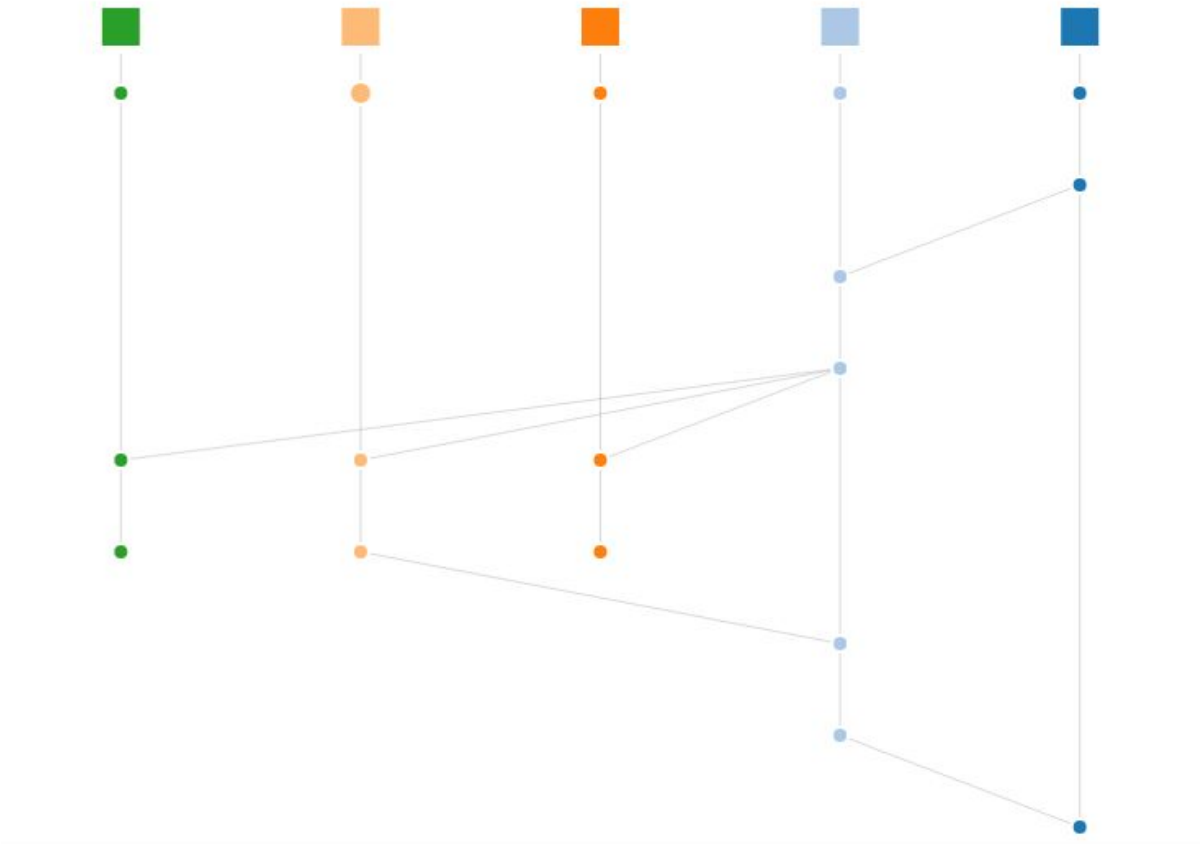


Figure: Example of a simple Get() operation visualized with ShiViz

Limitations

The most obvious limitation is the trade-offs dictated by the CAP theorem. Sacrifices in data consistency will manifest under certain usage scenarios. DeCandia et al [1] mentioned an

anomaly in the Amazon shopping cart in which deleted items can reappear under certain conditions [1]. In our system, if two clients have

Because our design decision that all nodes know about all other nodes, this introduces the loss of scalability. As more and more nodes enter the system the load on each node and subsequently the system increases.

A limitation of our sample web application is that in order for a client to see a list of key/value pairs that the system has, the client must already know what the key is. This was a design

When a node rejoins it requires that other nodes in the system do checks of every key single key value pair they have. As the number of keys increases this can become an expensive operation.

Discussion

An interesting challenge we had was deploying our system across many machines. We generated bash scripts and ran our system across eight separate machines on the UBC Lin Servers.

The most challenging portion was our decision to change our communication protocol from RPC to UDP. We did this so that we could integrate GoVector properly. This ended up causing a lot of issues as it required us to build checks around data transmission. Specifically, we needed to handle and test the parsing of the data and the way we listened to and wrote to connections. Another challenge in the same context was the communication between the client API and the backend. The implementation took longer than expected and required testing around the encoding and decoding of JSON data.

Ultimately, we found the work of developing a distributed system challenging but worthwhile. It provoked us to think critically when discussing concepts of replication, failures, and overall system designs.

Allocation of Work

Yongnan (Devin) Li: Consistent Hashing, OR-Map, Client Facing API (HTTP)

Hayden Nhan: OR-Map, Frontend, Client Facing API (HTTP)

Trevor Jackson: Node Rejoining, Gossip Protocol, GoVector & Shiviz, Node Communication

Li Jye Tong: Node Failure, Node Replication/Distribution Logic, Node Communication

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.