Project Milestone 4: DynamoDB

In this project, you are going to create a distributed key-value store based on Amazon's DynamoDB. This key-value store will have a gossip-based replication system, as well as a configurable quorum-type system for reads and writes. DynamoDB makes no effort to resolve conflicts in writes made by different nodes outside of direct causality, and will simply store and return multiple values if concurrent, conflicting writes are made.

DynamoDB consists of several servers, called Nodes, Each node exposes an RPC interface, and communicates with clients and other Nodes via that RPC interface. A client may read from or write to any node. Though the original variant of DynamoDB uses consistent hashing, we will not be using it in this project.

Resources for this project:

1. To read more about Amazon's original DynamoDB, this is the original DynamoDB paper (also linked in module 4 of canvas).
2. There is a video explaining how DynamoDB works by Peter Vosshall, one of DynamoDB's developers that is located in the classe's "Media Gallery" in canvas.
3. Patrick Liu created a video runthrough of the starter code (warning: 24 minutes long): located here (and also linked from Media Gallery in canvas).

Specification

Configuration

DynamoDB has the following configurable parameters

- **W** - The number of nodes that must perform a successful write
- **R** - The number of nodes that must perform a successful read
- **Cluster_Size** - The number of nodes in a cluster(also referred to as **N** in lecture)
- **Starting_Port** - The starting port number that the servers will listen on. There will be servers listening on ports in range [Starting_Port, Starting_Port + Cluster_size - 1]

The values of these parameters will be read from a configuration file in .ini format, similar to Project 2.

Vector Clocks

DynamoDB uses vector clocks to establish causality, similarly to how version numbers were used in SurfStore. Each element in the vector clock will be a pair (nodeId, version), and there can be potentially one element in a vector clock per node, for a maximum of Cluster_Size elements. This is a somewhat different representation than we used in lecture (which was an array of length Cluster_Size). Instead, we're adopting the representation used in the DynamoDB paper, where the array contains pairs indicating the vector clock number and the host ID, and any missing hosts can be assumed to have a vector clock value of 0. A vector clock must implement the following methods:

- **LessThan**(self, otherClock): Returns true if otherClock is causally descended from self.
- **Concurrent**(self, otherClock): Returns true if otherClock and self are concurrent, that is neither is causally descended from the other
- **Increment**(self, nodeId): Increments the version associated with NodeId.
- **Combine**(self, clocks): Returns a vector clock that is causally descended from all clocks.
- **NewVectorClock** - Returns a minimum vector clock.

Operations

A Node will expose (at minimum) the following functions via RPC:

- **Put**(key, context, value) - Puts a value with the specified context under the given key
    - If the context that the Node has already stored associated with the specified key is causally descended from the context provided to Put, i.e newContext < oldContext, Put will fail and the existing value will remain.
    - Put attempts to replicate the operation on the top **W** nodes of its preference list. If one is unavailable, it is skipped. If enough nodes are crashed that there are not **W** available nodes, Put will simply attempt to Put to as many nodes as possible.
    - Put will increment the vector clock element that is associated with this node
    - At the end of Put, all nodes that weren't replicated to should be stored so that a future Gossip operation knows which nodes still need a copy of this data
- **Get**(key) - Returns a list of (context, value) pairs associated with the given keys
    - Get will get (context, value) pairs from the top **R** nodes of its preference list. If one is unavailable, it is skipped.

- **Crash**(seconds) - Causes the server to emulate being crashed for the given amount of time, meaning that any RPC calls return an error condition indicating that the operation did not occur (note that we don't actually crash any of your nodes in this project--we just emulate a node being crashed with this call)
- **Gossip** - Replicates all keys and values from the current server to all other servers. If a given key is replicated to another node which already has a value for that key, then the vector clocks are used to either (1) garbage collect old values, or to (2) store multiple concurrent copies in the case of a version conflict.

You will likely have to expose more functionality via RPC to implement the project.

Basic Operating Theory

DynamoDB is an "eventually consistent" distributed database system. Our version consists of some number of nodes, running as RPC servers, storing keys of type string, and values of type byte[]. Upon startup, a DynamoDB node contains an empty key/value store and an empty preference list, but is assigned a node ID and given the **W, R, Cluster_Size** parameters. Then, a DynamoDB node is given a preference list in the form of a list of (address,port) pairs.

Upon receiving a request for a Put operation, a node will first attempt to put the value into its local key/value store. If the vector clock associated with this Put is causally descended from any previous values stored, then this value will replace them. Then, the node will attempt to replicate that value to **W - 1** other nodes, in order of preference, via RPC calls, skipping those nodes that do not succeed (because i.e. some of the nodes might be in an emulated crash state).

Upon receiving a request for a Get operation, a node will first attempt to get a (context, value) pair from its own key-value store. If multiple values are associated with this key (via values with concurrent vector clocks), all are returned. Then, **R - 1** other nodes are queried, in order of preference. Then, all these (context, value) pairs are combined such that only those pairs that have no causal descendents are returned.

Under normal circumstances, DynamoDB nodes would gossip periodically to share information. However, to ease testing, we expose Gossip as an operation that a client can request of a node. Upon receiving the Gossip request, a node will attempt to replicate its key/value store to the other nodes on its preference list. If a node on the preference list is crashed, it is simply skipped. Gossip should employ the same conflict resolution techniques as Put.

Similarly, under normal circumstances, nodes would crash and recover unpredictably. To ease testing, we also expose Crash as an operation. Upon receiving a Crash request, a node will immediately start returning error values for any RPC requests during the appropriate amount of time. When the duration has completed, the node should recover and function normally. We will not call crash() on a node that is currently crashed.

Conflicts

As stated above, DynamoDB makes no attempt to resolve conflicts. All conflicting items are returned to the user, and the user can resolve the conflict by making another call to Put, with a vector clock causally descended from each of the conflicting items.

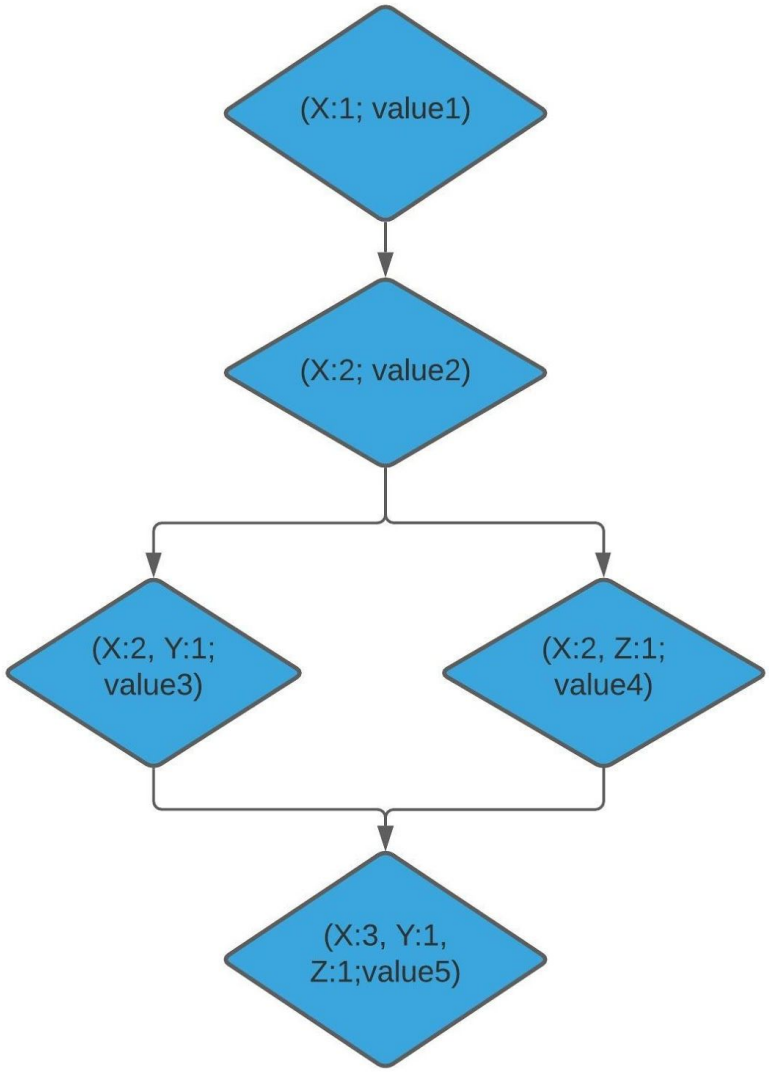

Figure 1: An example of conflict creation and resolution taken from the paper

Let us consider an example for the case **W** = 1**, R** = 1, as seen in Figure 1 above**.** Say a user Puts a value value1 on key k1, using server X. A user then Gets that value, and Puts another value value2 on the same key, again using server X. A subsequent Get on server X would return value2 instead of value1, since causality can be established. Now, suppose that server X gossiped to servers Y and Z, and the client Put value3 on key s1 using server Y, and Put value4 on key s1 using server Z. Once servers Y and Z have gossiped, there are now two conflicting versions values on key s1. A Get at this point would return both value3 and value4, with their corresponding vector clocks. To resolve this conflict, a client must perform a Put operation with a vector clock causally descended from both the vector clock

associated with value3 and the vector clock associated with value4. In our example, our client uses server X to perform this update, Putting value5. Once this change is gossiped, all servers have only value5.

Suggested TODO List

- Preliminary work
    a. Read the DynamoDB paper, with a particular focus on everything up to and including section 4.
    b. Watch Vosshall's talk on DynamoDB
    c. Check out the starter code, and watch Patrick's video describing the structure of the code
    d. Learn about unit testing in Go by reading chapter 11 of the Go book.
- Implementing the project
    a. Start by implementing VectorClock. This will be the basis of all your causality testing, so it's important that it works properly for all of your other code. We have given you a single unit test for VectorClocks--please add other tests so that you're sure that your implementation of the methods of VectorClock is correct.
    b. Implement only the local variants of Put and Get (that is, only worry about reading/writing from local storage, and not other nodes). You will likely want to expose these as RPC calls, as they are probably useful for implementing the full versions of Put and Get
    c. Implement Crash. All your RPC functions should check if the server is crashed before doing anything.
    d. Implement Gossip. For now, you can make it so it replicates to every node on the preference list. At this point, your DynamoDB should be fully functional for the case **W** = 1, **R** = 1.
    e. Implement Put and Get to handle **W** > 1, **R** > 1. That is, Put and Get should read from/write to **W** and **R** nodes, respectively.

Tips

- Test, test, test. More than ever, test coverage will determine which bugs are exposed.
- The majority of your causality checks should be in two places: Put to local storage, and Get from multiple nodes. Put to multiple nodes and Get from local storage should not need causality checks (why?)
- Gossip needs to replicate to nodes that Put hasn't replicated to yet. The straightforward way to do this is to maintain a data structure that keeps track of

which nodes haven't been written to for which (context, key, data) triples, which is appended to by Put, and consumed by Gossip. However, since multiple Puts and Gossip can happen simultaneously, the state of this data structure can vary depending on the order the list is modified. As a result, you will likely have to wrap access to the data structure in a Mutex or similar. Below is some example code for how you might do that:

```
import "sync"

l := make([]int, 10)
var m sync.Mutex //no initialization required

m.Lock() //Acquires lock for mutex, this will block if another
goroutine has the lock
l = append(l, 1)
m.Unlock() //Releases lock for mutex, allowing another goroutine to
acquire it
```

Testing

For testing, we advise you to use the `go test` unit testing framework, as it is the way that we will be testing your submission. Go testing is covered in Chapter 11. We have provided in the starter code a separate `mydynamotest` directory, containing two example unit test files. To run all the tests, navigate to the `mydynamotest` directory and run `go test`

Unit tests in the `go test` framework have a couple requirements. Unit testing filenames must be of the form `*_test.go`. Unit testing functions must have a function signature of the form `Test*(t *testing.T)` , and import the `testing` module.

Make sure to recompile your program before running `go test` if you've made any changes to it, as `go test` will not recompile for you.

Useful commands

- `go test -` Run all tests in the current directory
- `go test -v` Run all tests in the current directory verbosely

- `go test -run [testname]` Runs only the test named `testname`

To read more about testing, visit this link: https://golang.org/pkg/testing/ and consult chapter 11 of the book

Files you will modify in your solution:

- `Dynamo_VectorClock.go`
- `Dynamo_Server.go`
- `Dynamo_Utils` (if necessary)
- `Dynamo_Types` (if necessary)

Grading Rubric

- Basic put/get operations succeed
- Conflicting put operations result in getting all conflicting values
- Gossip correctly replicates key/value pairs to all other nodes
- Put/get with **w** or **r** > 1 query the correct number of nodes
- Values correctly to propagate to crashed nodes after they recover.

Starter code

A link to the starter code is available here: https://classroom.github.com/g/nu9uqx7l

###