Project 3: Paxos

Introduction

In this assignment, you will implement a simple in-memory database that is replicated across multiple hosts using the Paxos distributed consensis protocol.

You can download my 64-bit Linux implementing to play with:

- Example solution
- Example solution (macOS)

Save the file as `paxos-sol` and make it executable:

```
chmod 755 paxos-sol
```

The launch it using:

```
./paxos-sol 3410 3411 3412
```

This will launch a single instance that will listen on port 3410 (the first one listed) and will expect to find additional instances listening on ports 3411 and 3412, respectively. You can then launch additional instances:

```
./paxos-sol 3411 3410 3412
./paxos-sol 3412 3410 3411
```

You can also launch the instances on different machines by supplying IPv4 addresses and port numbers in the format `1.2.3.4:3410`

Requirements

Your system must implement the services to support Paxos, and it must support a simple command-line user interface. Each instance will run an RPC server and act as an RPC client. No background tasks will be necessary.

User interface

When running, and instance of your database should support a simple command-line interface. The user can issue any of the following commands:

- `help`: display a list of recognized commands

The main commands you will use are those related to finding and setting key/value pairs in the database. A `<key>` is any sequence of one or more non-space characters, as is a value.

- `put <key> <value>`: insert the given key and value into the database. If the key already existed, its value will be replaced. The instance will use Paxos to ensure that the value is only inserted if at least a majority of replicas in the system agree to insert it in the same order relative to other operations on the system.
- `get <key>`: find the given key and return its value. The instance will use Paxos to ensure that the read operation happens in the same order relative to other commands in the system.
- `delete <key>`: delete the given key and its associated value. The instance will use Paxos to ensure that the delete operation happens in the same order relative to other commands in the system.

In addition, another operation is useful mainly for debugging:

- `dump`: display information about the current node, including the complete sequence of operations that it has applied to the database and the current contents of the database. The contents of the database should be exactly the same on all replicas that have applied the same number of slots (described below).

The executable also supports a couple of command-line options:

- `-chatty=n`: control the level of logging that is output. n=0 gives the least output, and n=2 gives the

most.

- `-latency=<n>`: set the latency that is inserted into the message sequences. Whenever an RPC message is received, it will pause for a random duration between `[n,2n)` milliseconds before it acts on the value, and for another `[n,2n)` milliseconds before returning. This simulates slow transmission of the messages across the network. By default this number is 1000, which gives a delay of 1 to 2 seconds.

Replicas

Each instance participating in the system is called a *replica*. All of the replicas together constitute a *cell*. Each replica tracks a series of *slots*. Each slot represents a single command that should be applied to the database (a `put`, `get`, or `delete` operation). The goal of the system is to ensure that every replica agrees on the same operation in each slot, and applies them to the database in the same order. Since the database starts out empty and the operations are all deterministic, this ensures that each replica has the exact same database when the same number of slots have been applied.

Because of asynchronous messages, a replica may discover the decision for slot six before it has discovered the decision for slot five. If this is the case, it must wait until it learns the decision for slot five and apply it before applying the decision from slot six.

Note that `get` operations are synchronized just like `put` and `delete` operations, so the cell must agree on the point at which a `get` operation actually reads the database.

For simplicity, each replica should track a complete history if slots over its entire run. It should never garbage collect old slots. It is up to you to manage the process of applying operations to the database. Note that a simple Go `map` will be sufficient for holding the database itself.

Paxos

Each replica will implement all necessary roles in the Paxos algorithm.

Acceptor role

The simplest ones to describe and understand are those associated with the acceptor role. Each of these operations should be applied to a specific slot number, which should be part of the request data. As mentioned earlier, it is possible that the protocol will be working on multiple slots concurrently.

- Prepare(*slot*, *seq*) → (*okay*, *promised*, *command*): a prepare request asks the replica to promise not to accept any future prepare or accept messages for slot *slot* unless they have a higher sequence number than *seq*.

  If this is the highest *seq* number it has been asked to promise, it should return `True` for *okay*, *seq* as the new value for *promised* (which it must also store), and whatever *command* it most recently accepted (using the Accept operation below).

  If it has already promised a number ≥ *seq*, it should reject the request by returning `False` for the *okay* result value. In this case it should also return the highest *seq* it has promised as the return value *promised*, and the *command* return value can be ignored.

- Accept(*slot*, *seq*, *command*) → (*okay*, *promised*): an accept request asks the replica to accept the value *command* for slot *slot*, but only if the replica has not promised a value greater than *seq* for this slot (less-than or equal-to is okay, and it is okay if no *seq* value has ever been promised for this slot).

  If successful, the *command* should be stored in the slot as the most-recently-accepted value. *okay* should be `True`, and *promised* is the last value promised for this slot.

  If the request fails because a higher value than *seq* has been promised for this slot, *okay* should be `False` and *promised* should be the last value promised for this slot.

The sequence numbers (called *seq* above) should be pairs of values, each consisting of a number and an address. I recommend writing methods on this type to compare two sequence numbers, and also to convert them to strings for printing, e.g.:

```
func (elt Seq) String() string { ... }
```

Any type that has such a method can be printed easily using the various log and fmt commands.

For comparisons, define something like:

```
func (this Seq) Cmp(that Seq) int { ... }
```

where `Cmp` returns <0 if this<that, >0 if this>that, and 0 if they are equal.

Learner role

The learner role is even simpler:

- Decide(*slot*, *command*): a decide request indicates that another replica has learned of the decision for this slot. Since we trust other hosts in the cell, we accept the value. It would be good to check if you have already been notified of a decision, and if that decision contradicts this one. In that case, there is an error somewhere and a panic is appropriate.

  If this is the first time that this replica has learned about the decision for this slot, it should also check if it (and possibly slots after it) can now be applied.

Applying commands is generally straightforward. I recommend storing commands using a struct that tracks:

- The sequence number *n* when this command was first promised and accepted by any node. When a value is repeated in later rounds within the same slot, it should retain this sequence number.
- The command itself; a string should suffice.
- The address of the node that proposed this value.
- An int tag that was assigned to this command when it was first entered by the user.

Methods on these command objects to convert them to strings and compare them to each other (if the address and tag match, they should be considered equal) are also useful.

When a user enters a command from the shell interface, the command may not be executed until later, since it must wait until the command is decided upon in some slot. That decision comes in the form of a decide message, so you must have some way to communicate that it succeeded back to the user. I suggest doing the following:

1. When you issue a command, assign it a randomly-generated tag value to track it.
2. Create a string channel with capacity 1 where the response to the command can be communicated back to the shell code that issued the command.
3. Store the channel in a map associated with the entire replica. It should map the address and tag number (combine these into a string) to the channel.
4. Whenever a decision is applied, check if the command has a channel waiting for it by generating the same address/tag key and looking it up. If found, send the result across the channel, then remove the channel from the map and throw it away. If not channel is found, do nothing; the command was proposed on a different replica.

Proposer role

The proposer is a little more complicated. It is given a command to execute, and it must repeatedly propose it (in progressively higher slot numbers) until the command is decided upon by the cell.

It always tries to propose in the first undecided slot that it knows of. Note that it may be out-of-sync; the slot may already be decided, but this replica may now know about it.

It starts a single round of Paxos by picking a value *n*. If it has already attempted to propose something in this slot, or the acceptor handlers on the same replica have already received prepare and/or accept requests for this slot, it will know of a higher value of *n* than one. Otherwise it can start with one (combined with its address).

- Prepare: Once it has picked a sequence value *n*, it should sent a Prepare message out to the entire cell.

Use goroutines to do this concurrently.

When examining the replies, you should make note of the highest *n* value that any replica returns to you (so you can generate a larger *n* if necessary for a future round). You should also track the highest-sequenced command that has already been accepted by one or more of the replicas.

If you get a majority of responses that promise your sequence number, select your value. Note that once you receive a majority of "yes" votes, you can stop counting. You do not necessarily need to wait for all of the votes to come back. If one or more of those replicas that voted for you have already accepted a value, you should pick the highest-numbered value from among them, i.e. the one that was accepted with the highest *n* value.

If none of the replicas that voted for you included a value, you can pick your own.

In either case, you should associate the value you are about to send out for acceptance with your promised *n*.

If you do not get a majority vote (a majority of all replicas in the cell, not just a majority of those you heard back from), you should sleep for some random amount of time and then start over. Note that you can stop counting votes as soon as you receive a majority of "no" votes or errors; you do not necessarily have to wait for every vote to come back.

To pause, pick a random amount of time between, say, 5ms and 10ms. If you fail again, pick a random sleep time between 10ms and 20ms, etc., waiting longer each time (but always a randomized amount of time).

- Accept: With a value *command* and sequence number *seq* in hand, you can proceed to the accept round. Send an accept request to all replicas and gather the results. As before, in addition to counting the votes, track the highest *seq* value that any other replica reports back to you in case you need to pick a higher *seq* when you try again.

  If a majority accept your request, then the value is decided. In this case, send decide messages to all replicas in the cell, including yourself (no need to treat yourself as a special case).

  If you fail to gain a majority, sleep and start over.

Each time you start over, check to see if the slot has been decided. While you were running through the steps, a decision may have been reached and you may have been notified about it. In this case, you must move on to the next undecided slot before you start over. Or, if the value decided (while you slept) is the value you were trying to propose, you should not re-propose it in another slot.

Testing

Here are a few suggestions of scenarios to test and what to expect.

- Basic case. Start all of your replicas, then issue a `put` command on one. Once all of the replicas have settled down, do a `dump` on each one and compare the results. They should all know about the decision.
- Dualling proposers: set a high latency value. Type in conflicting `put` commands in each window, then move from terminal to terminal and hit Enter on each one as quickly as possible. You should see each replica trying to propose its value. It may take multiple rounds before they settle on one (the exponential backoff delays between attempts make it all but certain that one will eventually prevail). Dump each node and make sure they all agreed on the final decisions. Make sure that all of the commands were eventually decided.
- Kill a replica: start the cell and run a few commands. Then kill one of the replicas. The others should stil be able to make progress and decide on values. If you have 5 replicas, kill a second and make sure the remaining 3 can still decide values and remain consistent with each other.
- Start with missing replicas: repeat the kill experiment, but this time don't even start all of the replicas. Only start 4 out of 5, or 3 out of 5. The ones that are running should still run just fine.
- Kill and restart: run commands, kill a replica, run more commands, then restart the replica. On one of the replicas that you did not kill, issue a new command, say "put go gopher". The restarted replica should happily participate in the voting and receive the decision. When you dump it, it should *not* have the "go" key in its database. It should have the decision in the appropriate slot, but it should not have applied it yet.

- Kill, restart, and catch up: run commands, kill a replica, run more commands, restart the replica, run more commands on other replicas, then run a command on the restarted replica. It should start by trying to propose in slot 0 (since it is unaware of a decision for that slot). Through the normal Paxos mechanism, it will be forced to re-decide the value that was already decided for slot 0, which it will then learn about and apply.

  This process will continue until the replica finally finds a slot number that is not decided, and succeeds in getting a decision for the command you typed in.

  Note that this is a slow recovery mechanism if you have many slots, but it should get there eventually.

- Intermediate values: while running some of these test cases, run “dump” repeatedly on one of the replicas that is not busy proposing a value. You should be able to see slots that are promised, but not accepted, accepted but not decided, and decided. If you time it right (this is easy if the latency value is high enough) you should be able to see all of these stages.