Go_Team01-1

Forked from an inaccessible project

Update from 5
Administrator authored 5 months ago

Name	Last commit	Last update
code-samples	Init commit	1 year ago
data-samples	Init commit	1 year ago
datasets	Init commit	1 year ago
materials	Update from 5	5 months ago
misc	Init commit	1 year ago
src	Init commit	1 year ago
.gitignore	Update from 3	1 year ago
CHANGELOG	Init commit	1 year ago
LICENSE	Init commit	1 year ago
README.md	Update from 4	7 months ago

README.md

Team 01 - Go Boot camp

Warehouse 13

Contents

Chapter I

General rules

- Your programs should not quit unexpectedly (giving an error on a valid input). If this happens, your project will be considered non functional and will receive a 0 during the evaluation.

- We encourage you to create test programs for your project even though this work won't have to be submitted and won't be graded. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded.
- If your code is using external dependencies, it should use Go Modules for managing them

Chapter II

Rules of the day

- You should only turn ink `\*.go`, `\*_test.go` files and (in case of external dependencies) `go.mod + go.sum`
- Your code for this task should be buildable with just `go build`
- All your tests should be runnable by calling standard `go test ./...`

Chapter III

Intro

— Oh come one, Artie, this is ancient! - Lattimer was about to pull his own hair. - It's a 21st century, nobody uses pen and paper to catalogue stuff anymore!

— What do you want me to do, Pete? It's been like that since forever!

— Well, we have a computer, don't we? It's pretty ancient, but we can install...

— No, we can't! You know that Warehouse has to remain top secret, don't you? We don't download or install any software here.

— Okay, so you want us to write our own database implementation? Will that work?

— Hmm, it MAY work...

— Perfect! So, I'll ask Myka to implement one for us!

— Wait, I thought you were talking about implementing it yourself...

— Nah, I'm no good with coding. Anyway, let's design it! What information should we store?

— Every artifact is assigned its own unique ID and then we have to store some metadata about it in a structured format. Also, everything should be accessible through a command line interface, as I don't get these modern GUIs...

An hour later...

— What? You want me to write a fully functional key-value storage for working with JSON documents? From scratch?

— I know, I know! But you're not alone! Here, I've made you some coffee in an Andy Warhol's coffee mug, so it's pretty much and unlimited resource of caffeine superpower.

— But, Pete! We have to solve multiple issues! What if the data is corrupted? What if we can't access some artifact data when we need it the most?

— Don't worry, Myka, you can do it! I'll also sit here and help. Let's just go through the issues one by one.

Chapter IV

Task 00: Scalability

After some time, the blackboard was covered in writings.

"Access through a command line - there should be a separate application that will provide REPL and will connect to a running instance via network, even if it's just local host and port".

"We should be able to kill any instance (process) of the database and it should keep running and providing responses to queries. That means, one of the configurable parameters for instance should be a replication factor, meaning how many copies of the same document do we store. For testing purposes 2 is probably enough"

"Client should perform heartbeats checking if current database instance is accessible. If it stops responding, it should automatically switch over to another instance"

"Also, for simplicity, let's assume for now being scalable means client should be aware of all other nodes. Every heartbeat response from a current node should include all currently known instances' addresses and ports along with current replication factor"

So, here we need to implement two programs - one being the client and one being an instance of a database. Whenever you are starting a new instance, you should be able to point it to an existing instance, so after receiving a heartbeat it will send over its host and port to all other running nodes, and everybody will know the new guy.

If the instance node is started with a replication factor different from existing nodes, it should detect that and fail automatically without joining the cluster. This means replication factor should probably be included in heartbeat as well.

You can use any network protocol you like for this - HTTP, gRPC, etc.

Whenever replication factor is more than a number of running nodes, information about this problem should be included in a heartbeat and shown in every connected client explicitly. You can see an example of a user session in Task 01.

Actual working with documents will be implemented in next task.

Chapter V

Task 01: Balancing and Queries

— Okay, so let's use UUID4 strings as artifact keys. We also need to implement some balancing to provide fault tolerance...

Our simple database should only support three operations - GET, SET and DELETE.

Here's how a typical session should look like, with comments (starting with #):

```
~$ ./warehouse-cli -H 127.0.0.1 -P 8765
Connected to a database of Warehouse 13 at 127.0.0.1:8765
Known nodes:
127.0.0.1:8765
127.0.0.1:9876
127.0.0.1:8697
> SET 12345 '{"name": "Chapayev's Mustache comb"}'
Error: Key is not a proper UUID4
> SET 0d5d3807-5fbf-4228-a657-5a091c4e497f '{"name": "Chapayev's Mustache comb"}'
Created (2 replicas)
> GET 0d5d3807-5fbf-4228-a657-5a091c4e497f
'{"name": "Chapayev's Mustache comb"}'
> DELETE 0d5d3807-5fbf-4228-a657-5a091c4e497f
Deleted (2 replicas)
> GET 0d5d3807-5fbf-4228-a657-5a091c4e497f
Not found
>
# if current instance is stopped in the background
Reconnected to a database of Warehouse 13 at 127.0.0.1:8697
Known nodes:
127.0.0.1:9876
127.0.0.1:8697
>
# if another current instance is stopped in the background
Reconnected to a database of Warehouse 13 at 127.0.0.1:9876
Known nodes:
127.0.0.1:9876
WARNING: cluster size (1) is smaller than a replication factor (2)!
>
```

If a key specified in SET already exists in a database the value should be overwritten. If it doesn't, then SET operation should provide read-after-write consistency, meaning immediate reading should give you proper value.

When updating an existing value or deleting it, an eventual consistency should be implemented, meaning immediate (dirty) reads can (but not "should"!) give you old results, but after a couple of seconds the data should be updated to a proper new state.

You can implement key-hash-based balancing so your client could explicitly calculate for every entry the list of nodes where it should be stored according to a replication factor. This will also come in handy for deletion.

If a current node is killed during writing, your client should automatically perform another request to another available node. The only case when user should see the error like "Failed to write/read an entry" is when ALL instances are dead.

Chapter VI

Task 02: Long Live the King

Let's upgrade the logic from Tasks 00/01. Now, we introduce concepts of a Leader and a Follower nodes. This leads us to a list of important changes:

- from now on, client ONLY interacts with a Leader node. The hashing function to determine where to write replicas is now *on Leader*, *not* in client
- all nodes (Leader and Followers) keep sending each other heartbeats with a full list of nodes. If node doesn't respond to heartbeats for some specific configurable timeout (for testing purposes you should set it to 10 seconds by default)

- if the Leader is stopped, remaining Followers should be able to choose a new Leader among them. For simplicity, each of them can just order the list of nodes by some other unique identifier (numeric id, port etc.) and pick the topmost one. From that moment all heartbeats will include a new elected Leader
- if not able to connect to a known Leader, a client should try and connect to Followers to receive a heartbeat from them. If a Leader is killed, this heartbeat will include a new elected Leader

Chapter VII

Task 03: Consensus

NOTE: this task is completely optional. It is only graded as a bonus part

You may have noticed that a lot of things could go wrong in a schema provided above, specifically race conditions and ability to lose some data due to replicas not being re-synced automatically between instances after some of them die.

You can try and solve that for some extra credit by either using an existing solution or writing some workaround yourself. Here are some options:

- Using existing Raft implementation (https://github.com/hashicorp/raft) or writing minimal implementation by yourself (https://www.youtube.com/watch?v=64Zp3tzNbpE)
- Utilizing external tools, like Zookeeper (https://zookeeper.apache.org/) or Etcd (https://etcd.io/)
- Choosing some other way, like Paxos (https://github.com/kkdai/paxos), some blockchain implementation (like https://tendermint.com/) or your own hacks

...Hopefully, now Pete and Myka won't be looking through a mess of paperwork everytime they need to find something. Probably Artie will do it anyway, because sometimes it's really hard to challenge the force of habit.

But I think it was an interesting journey, during which we've found some cool artifacts on the way. Do you?

Chapter VIII

Reading

MIT Distributed Systems - RaftIntro MIT Distributed Systems - Raft1 MIT Distributed Systems - Raft2