Distributed Systems: Final Project

The project for this course will be to develop (in stages), a distributed, reliable backend in support of a mildly complex (think twitter) social media application.

The development of this project is divided into 3 stages!

Summary

The stages are split up as follows:

1. A monolithic web application, with all logic and persistence happening on process. This process exposes a simple social media website.
2. The monolithic web application is now split into several services, with a *stateless* web server and *at least one* backend service, which persists data and communicates with the stateless web server via gRPC.
3. The backend service(s) are now stateless, persisting their state in a raft, replicated data store.

Stage 1

Deliverable

Summary

A simple web application, comprised of a web server written in Go, serving html files from a single machine. For this stage of the project, you do not need to persist data in files or a database – instead, keep everything in memory. Do not use a database for this application.

Features

This application needs to have a small number of very clear functions. The minimum set are as follows:

1. Creating an account, with username and password
2. Logging in as a given user, given username and password
3. Users can follow other users. This action must be reversible.
4. Users can create posts that are associated with their identity.
5. Users can view some feed composed only of content generated by users they follow.

Note that these operations *should* be accomplished efficiently, but *need* not be. You won't lose points for O(n) algorithms, where an O(logn) solution is possible. However, you're selling yourself short ;)

If you want to build a system other than this twitter example, that is totally OK. Just speak to the professor/TA first, to get approval. We just want to make sure that your application is of comparable difficulty and testability.

Tests

You *will* be expected to write tests for your functions!! If you encounter any issues with testing, don't hesitate to ping your TA!

If you want to use fancy testing frameworks (like ginkgo), that's awesome! Not required, but might be fun.

To be clear we're not setting an arbitrary code coverage metric, but tests are super important, so they'll factor into your team's grade. If you're nervous about whether your tests are good enough, speak to your TA about it. We just want to drive home that tests are super important, especially in this phase, as these tests will be your canaries when you split off the backend in step 2.

Frameworks and Vendoring

On that note, if you want to use frameworks such as buffalo or ginkgo, go ahead, but make sure you practice vendoring them as dependencies! It'll make it easier on your teammates, and easier for me to grade! That being said, you definitely will not *need* to use any of these frameworks. As our system is relatively small, they won't save you a ton of time. It's also worth learning how stl packages like `net/http` and `testing` work, as that's what all of these frameworks are built on.

Structure

As friendly advice, you should try to structure your application as follows. You don't *need* to, but it will definitely make stage 2 easier!

```
cmd
|-- web
|   `-- web.go --> build target (the code that actually runs the server)
web
|-- web.go
|-- config.go --> config for web, determines port to bind to, etc.
|-- cmd        --> implementation of build target
`-- auth
    |-- auth.go   --> authentication module, creates a new auth object,
    |                  starts it, stops it.
    |-- config.go --> token validity duration, hash difficulties, etc.
    |-- errors.go --> errors that auth object can reply with
    |-- *.go       --> implement auth; use contexts, generated
    |                  proto, and storage packages
    |-- *_test.go --> don't forget tests!
    |-- storage
    |   |-- storage.go --> storage interface for auth
    |   `-- memory
    |       |-- config.go
    |       `-- memory.go --> *threadsafe* implementation,
    |                          using maps and lists
    `-- authpb
        `-- models.proto --> used to generate all data
                              primitives used by auth module
```

The auth package is enumerated as an example. All of auth's sister packages should look pretty similar, and `web` should compose these packages, hiding access to them behind http handlers, as was shown in class.

If you want, you can totally abandon this structure and follow your own path!! It's totally up to you, but please pick something sensible!! :)

Resources

The following resources might be helpful:

1. This open source book, which explains how to build a go web server from fundamentals. Note that there is way more information in this link than you'll need, but it's a great overview. Some of the snippets in Chapter 3 might be especially useful.
2. This blog post explains a popular method of organizing projects, which is quite similar to the one your TA demonstrated in-class!
3. This exploration of Kubernetes source, which your TA found on the day he was writing this doc, is really cool, and gives you an intimate sense of how the project works.
4. This example travis script for those of you who want to play around with automated builds and tests using Travis!
5. For leveraging contexts, check out this blog post, which goes over the subject way better than I could!!
6. This might be a bit late to slide in, but this talk on concurrency patterns is great.a
7. For faking http responses, this package is great.

Stage 2

Deliverable

In the next stage, you will split off your backend into a separate service. It can be a monolith (i.e. one service that performs all of the tasks related to your system's state management), or a set of services that each perform one small task, as described in-class.

This service (set) must communicate with the web server using an RPC Framework like gRPC (strongly recommended!) or Thrift. If you run into any issues getting these frameworks set up, don't hesitate to contact the TA!

At this point, your webpage service (i.e. the service that is rendering templates / serving web pages) should be *stateless.* In practical terms, you should be able to horizontally scale your web service without causing any concurrency problems on the backend, or UX problems on the front end. The Web Service should no longer be persisting any data (even session data!!), but instead be fully reliant on the backend service. If this isn't the case, you still have work to do :)

Depending on how you solved stage 1, this could be REALLY challenging – you might need to totally re-write your service, in some cases. If you followed the model your TA showed in-class, however, this might be the least time-intensive stage. If you have extra time and want a challenge, you might want to implement more features, or split the backend into more distinct services.

Resources

The following resources might be helpful:

1. This document is a super concise explanation of installing gRPC and protobufs (the transport serialization used by gRPC). It also references an example, which is a great resource in and of itself for learning protobuf syntax. In particular, it contains snippets for creating, registering, and dialling with gRPC servers/clients, which you can directly use in the project.
2. Review the example rpc slides, and the accompanying repo, which are posted on piazza
3. Adam's gen proto script :)
4. Your TA! If you have any issues!

Stage 3

Summary

In stage 3, we are finally going to give our system a much-needed upgrade –
persistence!! And not just any kind of persistence – we're talking about highly-
available, replicated persistence!

To accomplish this, we're going to use one of the two popular open source Raft
implementations out there, either the one from CoreOS or Hashicorp. You can
choose to use whichever one you prefer; in your final presentation, you to give
some rationale for why you picked the one you did! :)

Deliverable

This is a relatively small step, but it requires learning how your raft implmentation
of choice looks like!! It should end up being more of an ops problem than a dev
problem. Again, if you've followed the example architecture, it shouldn't be a
huge headache. If you find that it is, contact your TA!!

You'll need to:

1. Write code that stands up raft nodes (on your local machine is ok!!)
2. Replace your tried and tested storage implementations with a new one,
   which wraps a raft client for your raft implementation of choice. This will
   live alongside your old implementation in the `storage` directory, if you're
   following the recommended structure.
     1. This involves code that establishes a connection with your raft cluster.
        Now, you'll find that your storage implementation's `config`, `start`
        and `shutdown` actually do some heavy lifting!!

Details

Each service should have its *own client* wrapper for contacting the raft cluster,
but it is totally OK if they are all targeting the same cluster. If you want, you
can target multiple clusters – good luck getting all those to run on your laptop,
though :).

Don't worry about hosting this solution anywhere other than your laptop. If
you want to host it on AWS, heroku, or what have you (just to show off!), that's
totally OK. Just make sure that you have a configuration that will also work
on your local machine. One hack for accomplishing this in the recommended
structure, is to build out `cmd` directory as follows:

```
cmd
|-- local
|   |-- web
|   `-- backend
`-- aws
    |-- web
    `-- backend
```

In the above example, your config initializations under `local` would look very
different from your config inits under `prod`. Alternatively, you can read config
objects values from environment variables, have different env-setting shell scripts
for aws/local, and keep a single `cmd` folder. It's really up to you!

Errata

Did you find some bug with this document? Want to improve it? Go ahead and submit an issue on Github.