Module lab::lab3 [−][src]

[−] Welcome to Lab 3! The goal of this lab is to take the bin storage that we implemented in Lab 2 and make it fault-tolerant.

Lab 3 can be submitted in teams of **up to** 3 people.

Get Your Repo

This is the the first assignment you will complete in groups. As a team you will create a shared Github repository. Have one group member follow the GitHub classroom link available through the Lab 3 class announcement. Once one group member has made the group the others can select it. **Choose your teammates carefully prior to accepting a Github group**, the choice is binding. Once you have a repo select one of your group members to push their code from lab2 to the freshly created lab3 starter repo for your team. See this Github Documentation for how to perform the push.

Note that we don't provide great unit tests to test fault tolerance (as it's hard to spawn and kill processes from within unit tests). Make sure you test this sufficiently using a testing mechanism of your own design.

Backends
1) 1 ~ 300 backends online
2) at least 30 seconds between each join/leave (to migrate storage)
3) no data loss if at least 3 backends are online at all times

Keepers
1) 1 ~ 10 keepers online
2) at least 60 seconds between each join/leave
3) Everything is lost when a process leaves.
4) join with the same index but receive a different ID

System Scale and Failure Model

There could be up to 300 backends. Backends may join and leave at will, but you can assume that at any time there will be at least one backend online (so that your system is functional). Your design is required to be fault-tolerant where if there are at least three backends online at all times, there will be no data loss. You can assume that each backend join/leave event will have a time interval of at least 30 seconds in between, and this time duration will be enough for you to migrate storage.

There will be at least 1 and up to 10 keepers. Keepers may join and leave at will, but at any time there will be at least 1 keeper online. (Thus, if there is only one keeper, it will not go offline.) Also, you can assume that each keeper join/leave event will have a time interval of at least 1 minute in between. When a process 'leaves', assume that the process is killed– everything in that process will be lost, and it will not have an opportunity to clean up. Why do we need the 1 minute interval?

When keepers join, they join with the same index as last time, although they've lost any other state they may have saved. Each keeper will receive a new id in the KeeperConfig.

Initially, we will start at least one backend, and then at least one keeper. At that point, the keeper should send `true` to the ready channel and a frontend should be able to issue BinStorage calls.

index => the index of the keeper in the keeper list
id => a non-zero incarnation identifier for this keeper, indicating when this keeper was created relative to other keepers

Frontend would only be started after the first backend and keeper.

Consistency Model

To tolerate failures, you have to save the data of each key in multiple places. To keep things achievable, we have to slightly relax the consistency model, as follows.

`clock()` and the key-value calls (`set()`, `get()`, and `keys()`) will keep the same semantics as before. The Storage used in the back-end will return every clock() call in less than 1 second.

When concurrent `list_append()`s happen, calls to `list_get()` might result in values that are currently being added, and may appear in arbitrary order. However, after all concurrent `list_append()`s return, `list_get()` should always return the list with a consistent order.

Here is an example of a valid call and return sequence: How does the mechanism work?

- Initially, the list "k" is empty.
- A invokes `list_append("k", "a")`
- B invokes `list_append("k", "b")`
- C calls `list_get("k")` and gets `["b"]`. Note that `"b"` appears first in the list here.
- D calls `list_get("k")` and gets `["a", "b"]`, note that although `"b"` appeared first last time, it appears at the second position in the list now.
- A's `list_append()` call returns
- B's `list_append()` call returns The order is fixed after the functions return.
- C calls `list_get("k")` again and gets `["a", "b"]`
- D calls `list_get("k")` again and gets `["a", "b"]`

`list_remove()` removes all matched values that are appended into the list in the past, and returns number properly. When (and only when) concurrent `list_remove()`s on the same key and value is called, it is okay to 'double count' elements being removed. The return number may be incorrect.

`list_keys()` keeps the same semantics.

Entry Functions

The entry functions will remain exactly the same as they are in Lab 2. The only thing that will change is that there may be multiple keepers listed in the KeeperConfig.

Additional Assumptions

- No network errors; when a TCP connection is lost you can assume that the RPC server crashed.
- When a bin-client, backend, or keeper is killed, all data in that process will be lost; nothing will be carried over a respawn. should detect in 10 seconds
- It will take less than 20 seconds to read all data stored on a backend and write it to another backend.

The bin client should have get a different storage after hashing.
=> We can simply scan through the backend list.

Why is the sequential consistency assumption released?

use a background thread to keep scanning all keepers
and maintain an alive list.

Requirements

- Although you might change how data is stored in the backends, your implementation should pass all past test cases, which means your system should be functional with a single backend.
- If there are at least three backends online, there should never be any data loss. Note that the set of three backends might change over time, so long as there are at least three at any given moment.
- Assuming there are backends online, storage function calls always return without error, even when a node and/or a keeper just joined or left.

Building Hints

- You can use the logging techniques described in class to store everything (in lists on the backends, even for values). replace the key string functions with key list functions
- Let the keeper(s) keep track on the status of all the nodes, and do the data migration when a backend joins or leaves.
- Keepers should also keep track of the status of each other.

Keeper
1) sync the time
2) keep track of node statuses
3) migrate data
4) keep track of keeper statuses

For the ease of debugging, you can maintain some log messages (by using the log crate, or by writing to a TCP socket or a log file). However, for the convenience of grading, please turn them off by default when you turn in your code.

RPCs Between Keepers

While not necessary to complete the lab, you may wish to implement an RPC service so that keepers can communicate with one another. Without the RPC, we can still store information in the backend for the keepers to communicate.

You are allowed to define an RPC interface that is used between the keepers. Use protobuf syntax to define the messages and RPC calls that your keepers will use. Refer to the protobuf language guide for information on how to write protobuf specifications. You can also reference the `tribbler/proto/rpc.proto` file to see how the storage RPC interface is specified.

A blank `proto` file is provided to you in `lab/proto/keeper.proto`. When building your project with `cargo build` this proto file will get compiled into the `lab/src/keeper.rs` module, which you can then use to implement your RPCs

`Report.md`

Similar to in Lab 2, please include a report file. See the description in Lab 2 for more details.

Turning In

Each student must submit their own version of the group repository to Gradescope by the deadline (+their individual late hours). Every member of the team can submit the exact same repository commit. Individual submissions are being used to allow individual teammates to make use of their late hours if they feel the need.

Happy Lab 3. :-)