Project 2: Tribbler

1 Overview

This project is due on **Monday, April 7 at 11:59pm**.

The starter code for this project is hosted as a read-only repository on GitHub. For instructions on how to build, run, test, and submit your server implementation, see the `README.md` file in the project's root directory. To clone a copy, execute the following Git command:

```
git clone https://github.com/cmu440/p2.git
```

In this project, you will implement an information dissemination service called *Tribbler*. Clients of Tribbler can post short messages, read messages, and subscribe to receive other users' messages. We will begin by discussing the high-level architecture of the system. In your implementation, you will be using Go's `rpc` package to implement the application layer and backend storage system as well as incorporating caching features to improve the service's scalability.

You may work on this project **with a partner who is also in the class**. However, before you make any decisions, please refer to the late day policy to avoid any last-minute surprises regarding partner-projects and late days.

2 System Architecture

Tribbler will use a fairly classic three-tier architecture consisting of a client, application, and backend storage system:

Client: A Tribbler client application communicates with the application and is the layer seen by the user. Its function is to interpret user commands, send them to the application, and present the results back. For this project, a sample client has been provided as part of the starter code and is located in the `tribclient` package.

Application: The application layer implements the central commands of Tribbler, such as subscribing, posting messages, etc. The application layer will be implemented as a server and will run as a persistent service. Each server will have a `Libstore` library, which will handle all of the communication with the backend storage servers and will implement a lease-based cache consistency mechanism.

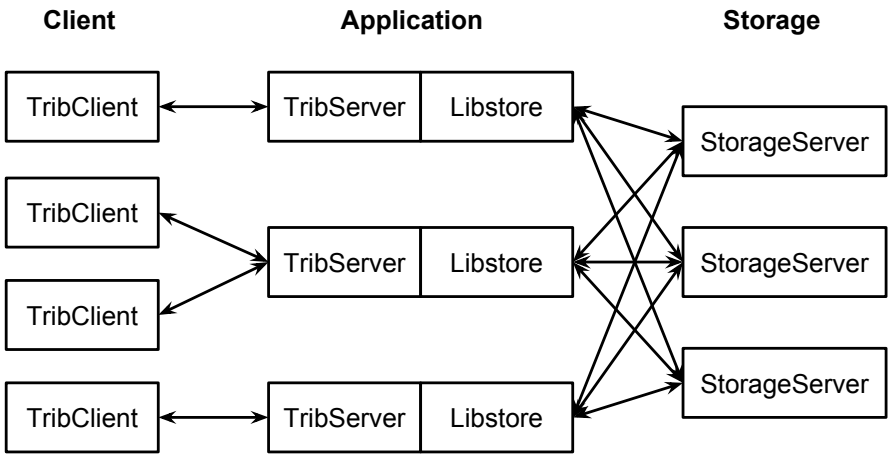

Figure 1: System architecture. One or more clients connect to a single Tribbler server, each of which communicates with one or more storage servers through its `Libstore`.

Storage: The storage system provides a key/value storage service (much like a hash table) and will store its `Tribble`s as JSON-marshalled objects. Each storage server will support `Get`, `Put`, `GetList`, `AppendToList`, and `RemoveFromList` queries. In addition, the storage system will support the following features:

- It is distributed over your cluster using *consistent hashing*. The consistent hashing ring will consist of a single *master* server, and all other servers will be *slaves*.
- It supports the aforementioned lease-based client caching model by issuing leases and registering callbacks.

We begin by discussing the application layer components in detail.

3 Application Layer

For the application layer, you will implement an RPC-based Tribbler server that supports the full set of functionality: subscribing/unsubscribing to users, posting/retrieving Tribbles, etc. The Tribbler server should be stateless—it won't persistently store any data about users. Instead, the Tribbler server will indirectly communicate with a backend key/value storage system, as shown in Figure 1. The only storage on the Tribbler server will be its small, short-lived cache.

3.1 Tribbler Server

Tribbler clients will interact with the Tribbler servers using Go RPC. All RPC calls reply with a integer status, which is defined in the `rpc/tribrpc` package. The `Get*` RPC calls (such as `GetTribbles`, `GetSubscriptions`, etc.) also reply with either a slice of strings (for listing followed user IDs) or a slice of Tribbles, each defined by the following Go `struct`:

```
type Tribble struct {
        UserID   string
        Posted   time.Time
        Contents string
}
```

The `PostTribble` operation takes only a user ID and the string content. Your Tribbler server is responsible for creating the `Tribble` and timestamping it. How you choose to store the list of Tribbles associated with a user is up to you.

The `tribserver` package contains the API that you will use to implement your `TribServer` (RPC helper functions and constants are provided in the `rpc/tribrpc` package as well). The `tribserver` package consists of the following operations, each of which are documented in the `tribserver` package:

Creating users: Before a user can either subscribe, add Tribbles, or be subscribed to, it must first be created using the `CreateUser` method. On success, it replies with `tribrpc.OK`. If the user already is in the system, it replies with `tribrpc.Exists`. There is no interface to delete users (once a user is created, it can never be deleted).

Adding/removing subscriptions: Users can subscribe or unsubscribe to another user using the `AddSubscription/RemoveSubscription` methods. Your server should not allow a user to subscribe to a nonexistent user ID, nor allow a nonexistent user ID to subscribe to anyone. Remember, that user IDs can never be deleted (that is, if you test that a user ID is valid, it will be valid forever). So don't worry about race conditions such as validating the user and then adding the subscription—the user will still be valid.

Getting subscriptions: The `GetSubscriptions` method retrieves a list of users to whom the target user subscribes.

Posting Tribbles: As discussed briefly above, the `PostTribble` operation takes a user ID and a string identifying the Tribble's content. The server is responsible for timestampping the entry, creating the `Tribble`, and passing the `Tribble` to its local `Libstore` to be stored/cached.

Getting Tribbles: The `GetTribbles` method retrieves a list of a maximum of 100 most recent Tribbles posted by a particular user, in reverse chronological order (most recent first). The `GetTribblesBySubscription` method retrieves a maximum of 100 total Tribbles, posted by all users to which a particular user is subscribed (also in reverse chronological order, most recent first).

At a high level, there are two technical requirements for your implementation:

Storage efficiency: A well-written implementation will *not* store a gigantic list of all Tribbles for a user in a single key/value entry. Your system should be able to handle users with thousands of Tribbles without excessive bloat or slowdown. We suggest storing a list of Tribble IDs (which you will have to define) in some way, and then storing each Tribble as a separate key/value item stored on the same partition as the user ID.[1]

Race-condition free: It is possible that multiple Tribbler servers will be handling data for the same user at the same time. If two or more servers receive `PostTribble` calls for the same user at the same time, your system *must* handle the two operations appropriately. To help you with this, we recommend implementing the storage server's `AppendToList` operation (and possibly others) so that they operate atomically.

3.2 The Libstore Library

As mentioned previously, each Tribble server will create and use an instance of the `Libstore` library to provide efficient and transparent access to the storage servers. The Libstore API is specified in the `libstore/libstore_api.go` file. RPC helper functions and constants are provided in the `rpc/librpc` package as well. Under the hood, it is responsible for two major tasks:

Request routing: Given a key, the Libstore must route the request to the appropriate storage server.

Consistent caching using leases: For frequently-accessed keys, the Libstore must keep a local copy (e.g., in a small hash table) respond to lease expiration/revocation events from the storage servers.

[1] Assuming you implement your storage server in this way, we recommend using the `Get`/`Put` methods to store `Tribble`s and the `AppendToList`/`RemoveFromList` methods to store lists of IDs that reference the actual `Tribble` objects.

Upon creation, an instance of the Libstore will first contact the master storage node using the `GetServers` RPC, which will retrieve a list of available storage servers in the consistent hashing ring. To simplify your implementation, you may assume that this list will not change over the course of execution. If the `GetServers` replies with status `NotReady`, then this means that not all of the storage servers have joined the ring yet. If this occurs, your client should sleep for one second and retry up to 5 times.

When (and if) `GetServers` replies with status `OK`, the Libstore will begin to communicate with the storage servers via RPC. Your Libstore should cache any connections made to the storage servers to ensure efficient communication (in other words, after opening a connection to a storage server, your Libstore should reuse the connection for subsequent requests).

When implementing the `Libstore`, we recommend taking the following approach:

1. First, support only a single master storage server, which will eliminate the need to worry about request routing (you'll be able to test that you have the basic RPCs working).
2. Next, add request routing and support multiple backend storage servers.
3. Finally, support caching and leases (you can avoid the need to do this earlier by always setting the `WantLease` parameter to `false` when executing `Get*` RPCs to the storage server.

4 Backend Storage System

The backend storage system consists of one or more `StorageServer`s, which together partition all data in the Tribbler system. Each `StorageServer` provides a key/value storage service and stores `Tribble`s as JSON-marshalled objects in an in-memory, thread-safe hash table. Each `StorageServer` will support `Get`, `Put`, `GetList`, `AppendToList`, and `RemoveFromList` queries.

At a high level, the Tribbler servers implement the logic that translates client requests into storage server operations. When a client sends a query to a Tribbler server, the server must determine which storage server is responsible for handling the query before carrying out the request. As Figure 1 suggests, the Tribbler servers should not interact with the storage system directly, but through its Libstore library instead.

The `storageserver/storageserver_api.go` file contains the API which you will need to implement against—consult it frequently! RPC helper functions and constants are provided in the `rpc/storagerpc` package as well.

4.1 Initialization and Startup

Before any Tribbler operations can be made, the storage system must properly setup its consistent hashing ring. The startup phase will differ depending on whether or not the storage server is a master or a slave:

Master server: When the master server starts up, it should begin listening for incoming connections and should register itself to receive RPCs from the other servers in the system. If one or more slave servers are expected to join, the master server should wait until all slaves have joined the consistent hashing ring before considering the startup phase to be complete.

Slave server: When a slave server starts up, it must register itself as part of the consistent hashing ring by sending an `RegisterServer` RPC to the master. If the master has received `RegisterServer` requests from all other slave servers, then it should reply with an `OK` status and a slice consisting of all servers (including itself) in the ring. Otherwise, the master should reply with a `NotReady` status indicating that the startup phase is still in progress. The slave server should sleep for one second before sending another `RegisterServer` request, and this process should repeat until the master finally replies with an `OK` status indicating that the startup phase is complete.

To simplify your implementation, you may assume that the list of servers is static throughout the system's existence. Once the initialization phase is complete and all storage servers have joined the ring, the storage servers should be ready to respond to RPCs from all other servers in the system.

4.2 Partitioning

Your storage servers will each store a subset of the key/value pairs by partitioning using *consistent hashing*. First, every node is assigned an unsigned, 32-bit integer in the range $[0, 2^{32} - 1]$. Then, for a given key, the node that handles it is selected by generating a hash in this range of the key and finding the "successor" of this value. For example, if a node n_9 is at `10,555` and another node n_{20} is at `19,200`, and `hash(key) = 13,232`, then `key` will be handled by n_{20} because that's the first node in the range *after or equal to* the key's hash value.

In this project, we'll use a simplified version of consistent hashing where the storage server is assigned its node ID beforehand (mostly to facilitate testing). Your implementation will use the `hash/fnv` package `New32` hash function to go from string keys to unsigned 32-bit

numbers.[2] We have provided a convenience function `StoreHash` for you in the `libstore` package that you should use to hash `string` keys to `uint32` hash values.

Your implementation should format keys using a separating colon (i.e. `thom:post-23ac9138d7`). Partitioning must be based *only* on the substring before the colon (i.e. `thom`). This will allow you to group all information for a single user on the same server (i.e. the keys `thom:radio` and `thom:head` should be handled by the same node).

4.3 Consistent Caching with Leases

Finally, you will improve the scalability of your system for users who are followed by many other users. The challenge is that extremely popular users may generate a large number of queries, all of which will go to the single server that handles their data. This requires a lot of RPCs and puts a heavy load on one or a few machines.

To fix this problem, you will add a caching mechanism to your Libstore and storage servers. The logic for caching is described below from the perspective of a querying Libstore L (the Tribbler server linked together with L is the one that is actually generating the query). Consider a key stored on storage server S. If a request for the key is made on L, then:

1. If the key is found in L's local cache and its lease is still valid, return the key's value from the cache.
2. If the key is not cached and L has sent `QueryCacheThresh` or more queries for the key in the last `QueryCacheSeconds` seconds, then send a request to S with the `WantLease` flag set to `true`. When the reply comes back, L should insert the result into its local cache before returning it to the caller.
3. Otherwise, send the query to S and return the key to the caller without caching.

This design will ensure that popular keys are cached on the `Libstore` machines while keeping the total number of granted leases manageable (since providing a lease on every query would add overhead without improving performance for unpopular keys).

Further, if a `Put`, `AppendToList`, or `RemoveFromList` request is made on storage server S, and S has previously granted valid leases for the key, then before applying the modification S must act as follows:

1. It should stop granting new leases for the key.

[2] This hash is very fast, but isn't cryptographically strong. It's good enough for our class, but there are probably better choices for industrial-strength key-value stores.

2. Then, it should send a `RevokeLease` RPC to all holders of valid leases for the key.
3. During this time, it should not allow modification until either all lease holders have replied with a `RevokeLeaseReply` containing an `OK` status, *or* all of the leases have expired (including their guard times).
4. Finally, it should apply the modification, after which S can now resume granting leases for the key again.

The `structs` and constants covering leases are provided in the `librpc` and `storagerpc` packages. The important points are summarized below:

- The server should keep track of what leases it has granted and grant leases for `LeaseSeconds`. However, the server should not consider the lease expired until an *additional* `LeaseGuardSeconds` have elapsed. This will help guard against clock drift between the server and clients.
- The server must keep track of which clients have cached which objects and send revocations *only* to clients that have (or had in the very recent past) a lease on the object.
- The `QueryCacheSeconds` and `QueryCacheThresh` constants control whether or not the `Libstore` should request a lease. The `Libstore` will do so if and only if there have been `QueryCacheThresh` queries within the last `QueryCacheSeconds` seconds.

  Note that it's fine if you occasionally slightly undercount the number of queries you've sent for a key. The important thing is to make sure that you *don't* request a lease on the first one or two queries per `QueryCacheSeconds`, and to make sure that you *do* request a lease if there are a lot of queries. Where we want to see caching, our tests will always send at least `QueryCacheThresh + 1` queries in half of the `QueryCacheSeconds` time period.
- The revocation RPC requires that your `Libstore` implements the `LeaseCallbacks` interface, defined in the `rpc/librpc` package. Remember in Go interfaces are determined based on whether or not a type implements all methods defined by the interface, so all your `Libstore` needs to do is implement the `RevokeLease` method accordingly and register itself to receive RPCs by calling Go's `rpc.RegisterName` function.
- The `RevokeLease` method should reply with status `OK` if the key was successfully invalidated, and should return status `KeyNotFound` if the key was not found in the cache. When the server revokes a lease, remember that it must not allow writes (or any further leases) for that key until *all* clients that were caching the key have confirmed revocation.

4.4 Atomicity and consistency

Your storage servers must properly handle race conditions and must not lose any data (i.e. it must not do an unlocked read-modify-write operation to add to a list across the network, as doing so could overwrite a different write operation that happened in between).

To simplify your implementation, you do *not* need to worry about cross-key consistency issues for `GetTribblesBySubscription`. For example, consider user `a` and user `b` in the following scenario:

1. `TribClient2`: `PostTribble("a", "first post!")`. Returns successfully.
2. `TribClient1`: Calls `GetTribblesBySubscription` (subscribed to `"a"`, `"b"`).
3. `TribClient2`: `PostTribble("a", "a was here")`. Returns successfully.
4. `TribClient3`: `PostTribble("b", "b is sleeping")`. Returns successfully.
5. `TribClient1`: Returns from `GetTribblesBySubscription`.

The return value for `GetTribblesBySubscription` in step 5 could be any of:

- `["a":"first post!"]`
- `["a":"first post!"], ["a":"a was here"]`
- `["a":"first post!"], ["b":"b is sleeping"]`
- `["a":"first post!"], ["a":"a was here"], ["b":"b is sleeping"]`

These are all possible because because there is no enforced ordering. Note, however, that the return must include the "first post" that completed successfully *before* the call to `GetTribblesBySubscription`. This might seem complex, but this is the behavior you'll observe if you just use `Get` and `GetList` to read the data without doing anything fancy.

5 Starter Code

The starter code for this project can be found in the `p2/src/github.com/cmu440/tribbler` directory. You will need to implement the following packages:

- The `tribserver` package contains the API and a skeletal implementation of the `TribServer` you will implement.

- The `libstore` package contains the API and a skeletal implementation of the `Libstore` you will implement.
- The `storageserver` package contains the API and a skeletal implementation of the `StorageServer` you will implement.

Note that the `tribclient` package contains a sample implementation of one possible client of the Tribbler system. It has been provided as an example and does not need to be modified. See the `README.md` file on GitHub for a more detailed discussion about the provided starter code and tests.

6 Project Requirements

As you write code for this project, also keep in mind the following requirements:

- You must work on this project with only your partner. You are free to discuss high-level design issues with other people in the class, but every aspect of your implementation must be entirely you and your partner's own work.
- You must format your code using `go fmt` and must follow Go's standard naming conventions. See the Formatting and Names sections of Effective Go for details.
- You may use any of the synchronization primitives in Go's `sync` package for this project.