440 Project 2: Tribbler

1 Overview

In this project, you will build an information dissemination service, called *Tribbler*. Clients of Tribbler can post short messages, and subscribe to receive other users messages. It's entirely possible that Tribbler resembles a popular online service whose mascot is a small tweeting bird.

There are three sub-phases to this project:

1. **Phase 1: Basic pub-sub functionality.** Runs on a single node, users can connect, post messages, and see lists of other users messages. Assigned 10/6, due 10/17.
2. **Phase 2: Scaling.** With the success of your initial service, you now have too many users to support on a single machine! In phase 2, you will partition your service across a cluster to enable you to continue to grow. Due 10/25.
3. **Phase 3: Consistent caching to tolerate popularity skew.** In phase 3, Lady Gaga joins your service, and suddenly, everybody wants to subscribe to read her tweets, er, tribbles. Having a few mega-popular Tribblers causes hot spots. In this phase, you'll enable caching to allieviate the hot-spots created by these mega-popular users. Due 11/3.

2 Starter code

`/afs/cs.cmu.edu/academic/class/15440-f11/P2/P2-handout.tar`

We have provided a sample server binary for part 1 in
`/afs/cs.cmu.edu/academic/class/15440-f11/P2/official-server`
and for part 2, in
`/afs/cs.cmu.edu/academic/class/15440-f11/P2/P2-handout.2.tar`
and
`/afs/cs.cmu.edu/academic/class/15440-f11/P2/official-server.2`
if you desire a reference for how the server should behave.

3 Service architecture

Your system will use a fairly classic three-tier architecture. You will implement the application logic and storage layers.

Client: The sample client application (or anything you choose to write) is the first tier (in a real application, it might be running as part of a web service). All it knows how to do is parse commands, send them to the service, and print out results.

Application logic: The application logic implements the things that make "tribbler" different from other applications. Commands like "subscribe to", and "get a list of all tribbles from users I subscribe to" are implemented here.

Storage system: The storage system you will implement provides a key-value storage service (much like a hash table). The storage system will handle JSON-encoded objects. Queries and insertions to the storage system will be in JSON format. Key-value storage systems in general support two types of queries: GETs and PUTs. Yours will have a few advances:

- In part 1, you will add support for add-to-list and remove-from-list to handle subscribing and unsubscribing.
- In part 2, you will distribute this service over your cluster using consistent hashing.
- In part 3, you will add consistent caching using leases.

In part 2, when you distribute the service, it will partition based upon the key. The key can be of the format "X:Y", e.g., "dga:post-23ac9138d7", and the partitioning will be based *only* upon the part before the first ":" (in this case, "dga"). This will allow you to group all information for a single user on the same server.

4 Part 1: Basic Tribbling

For part 1, you will implement an RPC-based Tribbler server that supports the full set of functionality: Subscribing to users, unsubscribing, posting, listing posts, etc. How you choose to structure this internally is up to you: We will test only the external behavior in response to RPC messages.

A note to the wary: READ AHEAD to the part 2 and part 3 requirements, because these should affect how you build your tribble server in part 1. Most importantly, in part 2, you will be separating the functionality of your system into an app logic component and a back-end storage system component. The back-end will have its own set of RPCs. This storage component will ONLY support GET(key), PUT(key, value), AddToList(key, value), and RemoveFromList(key, value). You will be much happier if you build your Part 1 server so that it internally uses only a single hash table, and you define internally Put, Get, AddToList, and RemoveFromList that are the only functions used to access that hash table.

Doing so will also help you avoid race conditions: The Go map class (hash table) is not thread safe. Only one goroutine can be accessing it at a time. Because your server can have multiple RPC clients connected at a time, and the Go RPC framework creates one Goroutine for each, you will want to serialize access to the hash table by communicating with it using channels.

5 Client Interaction with the Tribbler Service

Clients will interact with the Tribbler service using Go RPC. All RPC calls return a status integer, which is defined in `tribproto/tribproto.go`. "Get" RPC calls, for retrieving lists of tribbles, etc., also return either an array of strings (for listing followed userIDs) or an array of Tribble structures.

Tribbles are stored as this struct:

```
type Tribble struct {
  Userid string  // user who posted the Tribble
  Posted int64   // Posting time, from Time.Nanoseconds()
  Contents string
}
```

The Post operation takes only a username and a string. Your server should create the Tribble struct and timestamp it. How you choose to store the list of tribbles associated with a user is up to you, but we suggest taking advantage of the add-to-list backend storage command.

Note: A good 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 1000s of tribbles without excessive bloat or slowdown. We suggest storing a list of tribble IDs in some way, and then storing each tribble as a separate key-value item stored on the same partition as the user ID.

The file `tribbleclient/tribbleclient.go` contains a list of the functions used to access the Tribbler service. The skeleton `server/server.go` file we have provided has stubs for each of these functions.

Creating users Before a `userid` can either subscribe, add tribbles, or be subscribed to, it must first be created.

```
CreateUser(Userid string) status TribbleStatus
```

On success, returns `tribproto.OK`. If the user already is in the system, returns `tribproto.EEXISTS`.

There is no interface to delete users. A userid can never be re-used.

Manipulating the social graph Users can subscribe or unsubscribe to another user's tribbles using these functions:

```
AddSubscription(userid string, subscribeto string) status TribbleStatus
RemoveSubscription(userid string, subscribeto string) status TribbleStatus
```

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, though, that user IDs can never be deleted – so 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.

Observing the social graph This function lists the users to whom the target user subscribes:

```
GetSubscriptions(Userid string)  (subscriptions []string, status Tribblestatus)
```

Posting Tribbles The client interface to posting a tribble provides only the contents. The server is responsible for timestampping the entry and creating a Tribble struct.

```
PostTribble(Userid string, TribbleContents string) status TribbleStatus
```

Reading Tribbles The most basic function retrieves a list of the most recent tribbles by a particular user, in reverse chronological order, up to a maximum of 100 entries, in reverse chronological order (most recent first):

```
GetTribbles(Userid string)  (tribbles []Tribble, status TribbleStatus)
```

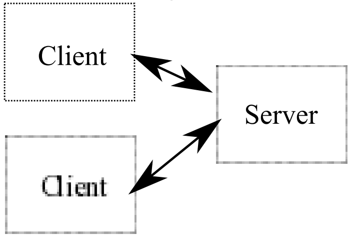
The other function retrieves all tribbles from all users to which a particular user is subscribed, up to a maximum of 100, in reverse chronological order (most recent first):

```
GetTribblesBySubscription(Userid string)  (tribbles []Tribble, status TribbleStatus)
```

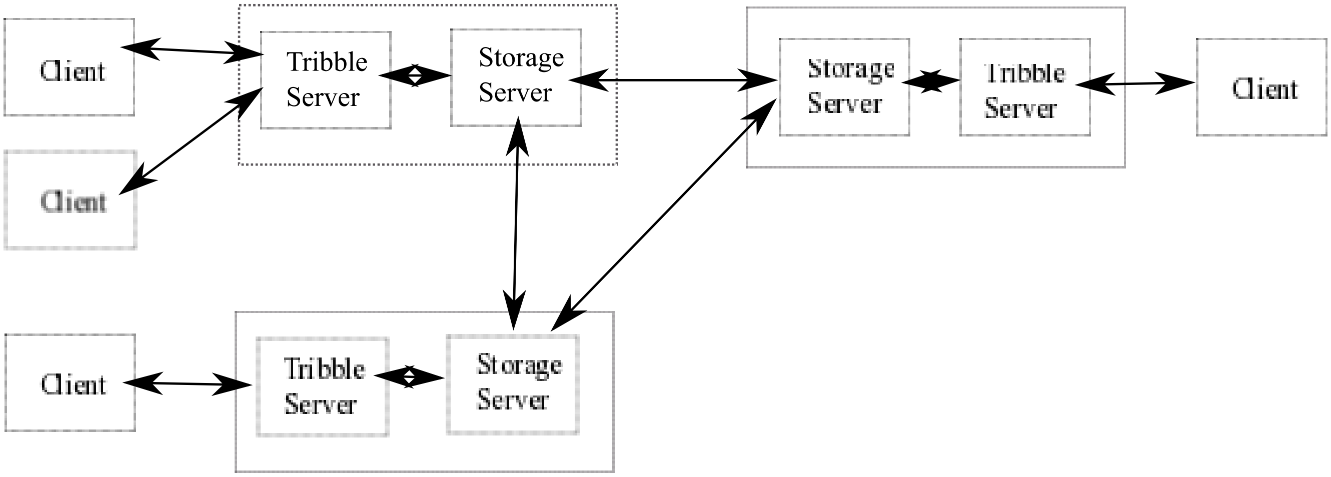
6 Phase 2: Cluster Scaling

In part 2 of this project, you will scale your tribbler server to be able to run on multiple machines in a cluster. To facilitate grading, we'll just run them as multiple processes on the same node, but they will communicate via RPCs over TCP in the same way they would if they were remote.[1]

To accomplish this task, you will first separate your system into an application logic component, the Tribbler server, which will be stateless: It won't store any persistent data about users at all. The tribbler server will communicate with a back-end key-value storage system, the Storageserver:



Part 1 organization



Part 2 organization

[1] It's straightforward to make your servers actually run on multiple machines. The only way in which we're cheating is by always sending "localhost" as the hostname.

The Tribble server will *only* implement the application logic needed to translate, e.g., “Subscribe to X” into storage requests, e.g., “AddToList ...” and provide functions such as checking to make sure that the user one is subscribing to actually exists. It won’t store anything itself.

In the implementation (see sample code), each process will have both a tribble server and a storage server. Each is provided by a different object which registers its methods with the RPC server. The tribble server will talk only to its “local” storage server – the one in the same process. The storage server component will provide the abstraction of a large, fast key-value store, and is responsible for coordinating between storage server instances. For example:

1. TribbleServer on node 1 asks Storage server on node 1: Get(“testkey”)
2. Storage server 1 examines “testkey”, decides that this key is actually stored on node 2.
3. Storage1 → Storage2: Get(“testkey”)
4. Storage2 → Storage1: Value=...
5. Storage1 → TribbleServer1: Value=...

In this way, your Tribble server code doesn’t need to know whether it’s running on one node, or 1000. It behaves the same way regardless.

6.1 Partitioning

Your storage servers will divide the responsibility amongst each other using *consistent hashing*. We’ll cover this technique more in class, but it’s quite simple: Take a numeric range from, e.g., 1 to $2^{32}-1$. Like most computer math, when you get to the end of this range, you wrap around to the beginning: $0,1,2,...,2^{32}-2,2^{32}-1,0,...$

Every node picks an ID somewhere in this range. To identify which node handles a particular key, hash the key into a number in this range. The key will be handled by the “successor” to this number. For example, if there is a node n_9 at 10,555 and another node n_{20} at 19,200, and hash(key_1) = 13,232, then key_1 will be handled by n_{20}, because that’s the first node in the range *after* the key.

We’ll use a simplified version of consistent hashing where we tell each node what its ID is, mostly to facilitate testing. Servers take a command line parameter that specifies where in the numeric ID space they sit.[2] If no ID is provided, your storageserver should pick one randomly using rand.Uint32(). Don’t forget to seed the random number generator first, or you’ll always get the same node ID on all nodes!

Your implementation should use the `hash/fnv` package New32 hash function to go from string keys to 32 bit numbers.[3] The key ID will be an unsigned 32 bit integer (uint32).

Remember that partitioning only happens based upon the characters before the first “:” in the key. The keys “dga:bad_taste_in_music” and “dga:likes_go” should both be handled by the same node![4]

6.2 Knowing which other Storageservers exist

Your implementation does not need to handle node arrivals or departures. You can assume that the list of servers is static throughout the system’s existence.[5] But your nodes still need to find out what the list of nodes is!

To accomplish this, the server takes a command line flag informing it either:

- This server process is the master, and there will be *N* nodes total

OR

- The hostname:port of the master.

The non-master servers will use the `Register()` RPC to the master in order to register and find out the list of other nodes. Because nodes may not join at the same time,

- The master may not have started yet, and the RPC Dial may fail
- Not all nodes may have joined yet, so register may return “not ready”.

[2] Our implementation is simplified in two ways. First, most consistent hashing uses a larger numeric space – 64, 128, or 160 bits, instead of a 32 bit number. Second, our servers will only join the space once. In real implementations, each server picks multiple IDs in the range, to provide better load balancing.

[3] 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.

[4] You might find the function `strings.Split` handy.

[5] Obviously, this is not an assumption you would make outside of a class assignment!

In either of these cases, your node must keep retrying, with a 1 second delay between retries. The master will not return "Ready" until all nodes have registered, so that it can give out a list of all of the hostname:port and node IDs of the storage servers.

6.3 Storage server API

The storage server API is documented in the handout code in `storageproto.go`. Note, however, that this only defines the RPC interface to the storage server. You are free to pick any way you want for your local Tribbleserver application server to communicate with its local storage server. For convenience, we have provided a new main function in server.go that instantiates the local storage server and passes a pointer to it to the constructor for your Tribble server.

To work around some weirdness with the go RPC, we have provided an RPC adapter file, storagerpc, which glues the RPC calls into functions defined in storageserver.go. Please don't modify storagerpc.go – as you can see, it's just there to make sure that the RPC package exports only the right set of functions.

Lists should store only one instance of a particular item. If the item already exists in the list, the storage server should return `storageproto.EITEMEXISTS` and not add a duplicate item to the list. This should make it easier to handle subscriptions and unsubscriptions.

6.4 Notes on consistency/atomicity

Your server must not lose data. e.g., it must not do an unlocked read-modify-write to add to a list across the network – doing so could overwrite another write that happened in between!

You do not need to worry about cross-key consistency issues for GetTribblesBySubscription. For example, in this scenario:

1. Client2: Post("a", "first post!"). Returns successfully.
2. Client1: calls GetTribblesBySubscription() (subscribed to "a", "b")
3. Client2: Post("a", "a was here"). Returns successfully.
4. Client3: Post("b", "b is sleeping"). Returns successfully.
5. Client1: returns from GetTribblesBySubscription

Your server could return any of:

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

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.

6.5 Testing

We will test both your entire server application using the Tribbler API, as well as your storage server system using the Storage API. Tests are forthcoming.

7 Phase 3: Consistent Caching with Leases

In this phase, you will further improve the scalability of your system for users who are followed by many other users. The challenge is that these highly-popular users generate a large number of queries, which all go to the one machine that handles their data. This requires a lot of RPCs and puts heavy load on just one or a few machines.

To fix this problem, you will add caching to the Storageserver layer. The logic for caching will look like this, from the perspective of querying storage node Q (the tribbler server linked together with Q is generating the query):

1. If the key is stored locally, return it.

2. If the key is stored on remote node N...
   (a) If the key is stored in Q's local *cache*, and its lease is still valid, return it from cache.
   (b) If this query is the 3rd or later query that *Q* has sent for this key in the last 5 seconds, then request it from node *N* with the WantLease flag set to true. When the reply comes back, insert it into the cache and return it.
   (c) Otherwise, send the query to *N* and return it without caching.

The lease-granting node *N* must track what leases it has granted.

This design is intended to ensure that popular keys are cached at storagenodes while keeping the total number of leases granted manageable. (Providing a lease on every query adds overhead without improving performance for unpopular keys).

For this portion of the assignment, we have changed the interface in `storageproto.go` to include the `WantLease` flag in Get requests, and to return a `LeaseStruct` in the responses.

If a modification request (Put, AppendToList, RemoveFromList) arrives at the owning node *N*, and *N* has granted valid leases to the key, then before applying the modification, *N* must:

1. Stop granting new leases on the key (the server may still reply to GET requests, but will not give leases)
2. Send a RevokeLease RPC to the holders of valid leases
3. Not allow modification until *all* lease holders have replied with a RevokeLeaseReply containing an OK status, *or* all of the leases have expired, including their guard times.
4. Apply the modification
5. Can now resume granting leases for the key

7.1 Lease rules

Servers should grant leases for 5 seconds.

Clients should expire those leases after the number of seconds specified in the lease.

The server should consider a lease no longer outstanding after 5 + 1 seconds, the lease seconds plus a small amount of guard time in case of clock drift. The formal definitions for these constants are:

```
const (
        QUERY_COUNT_SECONDS = 5  // Request lease if this is the 3rd+
        QUERY_COUNT_THRESH = 3   // query in the last 5 seconds
        LEASE_SECONDS = 5       // Leases are valid for 5 seconds
        LEASE_GUARD_SECONDS = 1  // Servers add a short guard time
)
```

It's OK to be a bit inaccurate about the `QUERY_COUNT_THRESH`. If you occasionally slightly undercount the number of queries you've sent for a key, it's OK. The important thing is to make sure that you *don't* request a lease on the first one or two queries per count seconds, 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 3 queries in *2 seconds*, not 5 seconds.

7.2 Revocation

The revocation RPC takes a key to revoke the lease for.