15-440 Fall 2012 Project 2: Tribbler

Assigned: 10/9
Part A due: 10/23, 10pm
Part B due: 11/1, 10pm

David Andersen (`dga@cs.cmu.edu`) is the lead instructor for this assignment.

1 Logistics

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 actual implementation must be entirely your group's own work.

All code for this project can be obtained from the following AFS directory:
`/afs/cs.cmu.edu/academic/class/15440-f12/P2`
In this directory, you will find the following:

`go`: The root of a code tree for the project. Refer to the file `go/README.txt` for an explanation of the tree structure.

`P2-f12.tgz`: A compressed tar file containing the entire code tree.

`writeup.pdf`: This document.

We may create modified versions of some of the files in the tree as the project progresses, so keep checking the announcements posted on the class web page.

Details on how to hand in your code will be provided closer to the due dates.

This project is designed so that you can do your work on either a Linux machine or an x86-based Mac running the OS X operating system. If you are using one of the Andrew Linux machines (including those in the GHC third floor cluster), you will find `go` files installed in `/usr/local/lib/go` and the `go` program in `/usr/local/bin`

You will need to set the `GOPATH` environment variable to the path of the `go` directory for your project. So, for example, if you untar the distribution into the directory `/usr/me/15-440` and you use `csh`, you should include the following line in your `.cshrc` file:

```
setenv GOPATH /usr/me/15-440/P2/go
```

You also need to set your GOROOT environment variable appropriately for your machine. On Andrew Linux, use:

```
setenv GOROOT /usr/local/lib/go
```

2 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: Implement Tribbler using the reference storage system**
2. **Phase 2: Implement the back-end storage system**

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.

For efficiency, your application logic server (the "tribbler" server) will run as a persistent service. It will be able to cache frequently-used items so that it does not need to go to the backend storage system to retrieve them. To do so, it will use a callback and lease-based cache consistency mechanism similar to that which you learned about for AFS.

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:

- It supports add-to-list and remove-from-list to handle subscribing and unsubscribing;
- It is distributed over your cluster using consistent hashing; and
- It supports the aforementioned lease-based client caching model.

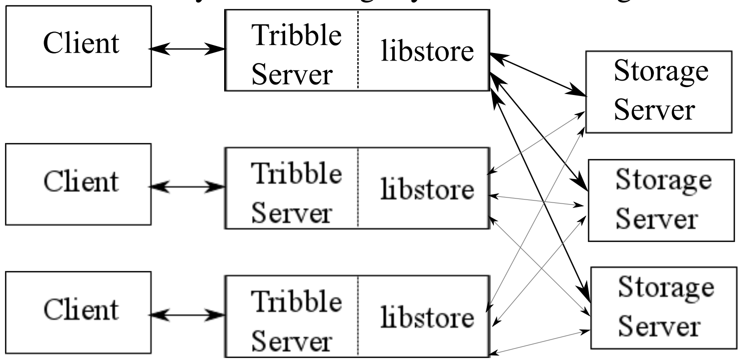
To distribute the service, it will partition based upon the key. The key can be of the format "X:Y", e.g., "dga:post-23ac9138d7", and the partitioning will be based *only* upon the part before the first ":" (in this case, "dga"). This will allow you to group all information for a single user on the same 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. It MUST store all of its data in the back-end key-value storage nodes, which we will provide. How you choose to structure it internally and represent the data is up to you: We will test only the external behavior in response to RPC messages. While we may look for *excessive* amounts of RPCs to and from the storage servers, or excessively large RPCs, we won't look inside of them.

A suggestion: READ AHEAD to the part 2 requirements so that you understand the big picture of what your system will be doing, and how the back-end storage system works. The back-end has its own RPCs. This storage supports GET(key), PUT(key, value), AddToList(key, value), and RemoveFrom-List(key, value). You will be much happier if you build your Part 1 server so that it uses these interfaces appropriately.

Your system will be structured as an application logic component, the Tribbler server, which will be stateless: It won't persistently store any data about users at all. The tribbler server will communicate with a back-end key-value storage system, the Storageserver:



The only storage on the Tribbler server will be its (small, short-lived) cache.

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.

There are two technical requirements for your implementation:

1. **Storage efficiency**: 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.

2. **Race-condition free**: Multiple Tribbler application servers may be handling data for the same user at the same time. If two or more servers receive "post" updates for the same user at the same time, your system *must* handle them appropriately. To help you with this, the AppendToList operation on the storage server operates atomically.

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 Tribblestat
```

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 Tribbl
```

5.1 Tribble server interaction with the storage service

Your tribble server will link against the `libstore` library, which provides transparent access to the back-end storage service. We will provide you with a sample `libstore` library to start working with, and you will write your own to replace it.

Libstore provides the functions listed in `libstore/libstore-api.go`.

5.2 Writing Libstore

Libstore provides easy-to-use storage access functions to the client code (in this case, the Tribbler server).

Under the hood, it is responsible for two major tasks:

1. *Request routing*: Given a key, libstore must route the request to the appropriate storage server; and

2. *Consistent caching using leases*: For frequently-accessed keys, the library must keep a local copy (e.g., in a small hash table), and respond to lease expiration events and listen for revocation requests from the servers.

Upon creation, an instance of Libstore will first contact the master storage node, supplied to it as a parameter. It will use the `GetServers` RPC to obtain a list of available storage servers and their IDs in the consistent hashing ring. You can assume that this list will not change over the course of execution.[1]

The response to GetServers includes a `bool` indicator, `Ready`. If the master is not ready, it means that not all of the storage servers have joined the system yet. If this occurs, your client should sleep for a second or two and retry up to 5 times.

It will then open, as necessary, and cache, RPC connections to the back-end storage servers. By "cache", we mean that after opening a connection to a back-end, your libstore should reuse the connection for subsequent requests.

Suggested implementation path:

1. *First, support only a single storage server*. Don't do any request routing. You'll be able to test that you have the basic RPCs working.

2. *Next, add request routing* and support multiple back-end 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."

[1] This is an obvious difference from reality.

6 Part 2: Implementing the Backend Storage System

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.[2]

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 begin an HTTP server and register a Storageserver object with and RPC handler on the HTTP server. This is accomplished for you by the storageserver. Your storageimpl will provide the actual methods called via RPC.

Each storage server is responsible for only its sub-range of hash values (see next section, Partitioning). It should return an error for attempts to ask it for items outside of its range. The client storage library should have found the correct server to talk to.

The library will, in turn, hide from your Tribble application logic whether it is talking to one or one thousand back-end servers.

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: $0, 1, 2, \dots, 2^{32}-2, 2^{32}-1, 0, \dots$

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.[3] 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.[4] The key ID will be an unsigned 32 bit integer (uint32).

We have provided a convenience function for you in the libstore-api.go file that will hash a string using FNV. It does not separate the string by colons - your code must do that.

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![5]

[2]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.

[3]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.

[4]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.

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

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.[6] 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".

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.

The storage library will use a similar `GetServers` call to find out the identities of the servers. The call to GetServers should retry in the same manner.

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.

We have provided an RPC adapter file, storagerpc, which glues the RPC calls into functions that must be defined in your storage implementation. (We did so to ensure that, first, only the specific functions are exported; and second, to be able to detect missing functions at compile time.) Please don't modify storagerpc.go – as you can see, it's just there to make sure that the RPC package exports exactly 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.

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

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 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 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 libstore and storage servers. The logic for caching will look like this, from the perspective of querying client libstore *Q* (the tribbler server linked together with *Q* is generating the query):

For a key stored on remote node *N*...

1. If the key is stored in Q’s local *cache*, and its lease is still valid, return it from cache.
2. 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.
3. 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 clients (in the library) while keeping the total number of leases granted manageable. (Providing a lease on every query adds overhead without improving performance for unpopular keys).

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 could still reply to GET requests, but will not give leases. Whether or not you implement this is up to you.)
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

The timeouts and rules covering leases are defined in `storageproto.go`.

The server should grant leases for `LEASE_SECONDS`. But it should not consider the lease expired until an *additional* `LEASE_GUARD_SECONDS` have elapsed. This helps guard against clock drift between the server and clients.

`QUERY_CACHE_SECONDS` and `QUERY_CACHE_THRESH` control whether or not the storage library will request a lease. It will do so if there have been THRESH queries within the last SECONDS seconds, and not otherwise.

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 `QUERY_CACHE_THRESH + 1` queries in half of the `QUERY_CACHE_SECONDS` time period.

Please do not hardcode those values into your code: We may decrease or increase the values from storageproto.go to facilitate testing. Refer to them via storageproto.

7.2 Revocation

The revocation RPC takes a key to revoke the lease for. Your client library must have registered a handler of type StorageRPC (see cacherpc/cacherpc.go). When requesting a lease, it must properly supply its HTTP hostname/port (remember that we're using localhost for everything to simplify things) in the lease request. To revoke leases, the server should contact the client on this callback. Upon receiving this request, a client should immediately invalidate its cached value of the specified key.

It should return success if the key was invalidated or if the key was not cached.

The server must not allow writes (or further leases) until *all* clients that were caching the object have confirmed revocation.

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.