P2: The Trouble with Tribblers

15440, Fall 2016
Zach Newman zen
Cortney Padua cpadua
November 16, 2016

1 Introduction

P2 served as an exercise in distributed storage systems and multilayered protocols. Major challenges included dividing the work and creating layers of abstraction between the client, the main server, and the storage server; managing concurrent access to caches and other shared data storage; and implementing and handling timeouts across the entire system. In order to tackle this project, Zach (zen) took control of the storage server, while Cortney (cpadua) wrote the Tribserver and its Libstore.

2 Tribserver and Libstore

The Tribserver implementation is where most of the storage schema decisions were made for our Tribbler system. In Tribserver, we decided how keys and values would map to each other, and we passed those key/value pairs back and forth with the storage server using RPC calls. In the end, each user was mapped to a list of users subscribed to them, a list of users they are subscribed to, and a list of keys representing Tribble posts. Each Tribble post key was then mapped to the actual JSON-encoded Tribble, in order to maximize performance for users with many Tribbles.

This schema was communicated to the storage server using the Libstore utility. Libstore made most of the actual RPC calls to the storage server, handled errors, and passed data back to the Tribserver. In addition, Libstore had its own cache of data for frequently accessed users and Tribble posts. To implement this, we first give the Libstore a list of all keys accessed thus far and associate it with a counter and the last time that key was accessed. Before making any RPC Get requests to the storage server, we check the last time that key was accessed and the number of time it was accessed in a set period of time. If the key has been accessed enough times, we request a lease from the storage server so that we can put the information in our cache.

The Libstore has two maps: one that maps keys to string data, and one that maps keys to list data. In addition to this data, the cache also holds the length of time that entry is valid, and a timer that fires after the lease has expired. When this happens, we use Go's built-in `AfterFunc` function, that calls the given function in its own goroutine. In this case, we call a function that finds the proper entry in the cache and deletes it. We also use this expiration function when the storage server calls `RevokeLease` to force the expiration of a cache entry. The two caches are protected by a mutex each, and one must acquire the mutext to do any read or write of the cache. For better performance, we could have used condition variables; however, we felt that it was better to implement a correct and concurrent cache before aiming for performance upgrades.

The Libstore is additionally responsible for routing all requests to get or modify keys and associated values to the correct storage server responsible for that key. To do this, we use a consistent hash ring, where each storage server is responsible for all keys whose hash value is before or equal to the server's hash and after the previous server's hash. To do this, we simply hash the key that Tribserver has requested access to, find the appropriate server using a helper function, and connect to that server to do the operations for this key. Since the list of servers does not change after Libstore connects to the master storage server, we can dial all of the storage servers when Libstore is instantiated.

3 Storage Server

We divided the information we were storing into two data structures, depending on whether the data was in list form, or a single string. We used another map to keep track of our leases associated with each key. When a get was issued with a request for a lease, we would grant it if the lease was not being revoked, and save the time of this transaction and the hostport. If a put, append, remove, or delete operation happened with a key that we had leases for, we would send revoke calls to each of the clients that had requested leases. A problem we encountered is that we would be waiting for a server to respond to us revoking a lease, and we would no longer need it expired because enough time would pass that we could make progress assuming it had been expired. In order to accommodate this, we use asynchronous rpc calls and take the earlier of a response from the libstore, or a timer we set up for this purpose.

Our Synchronization strategy for the storage server is to have a global mutex for each shared data structure. These include two dictionaries that map keys to values and lists of values, and another dictionary that keeps track of leases associated with each key. Synchronizing accesses to these shared resources is not sufficient to conform with the requirements, however. For example, if one put is called before the other, we need to guarantee that the first one will execute first. We achieve this by creating another class of mutexes that prevent appends, puts, removes, or deletes from occurring where the key is the same. However, we should allow concurrent write operations for different keys. We allocate an array of 10 mutexes in the beginning of the program and give them out to each incoming write request based on the hash of the key. This solves this problem for us. A fixed amount of mutexes is useful for avoiding race conditions.