A f-fault tolerant distributed hash-table implementation using OpenReplica

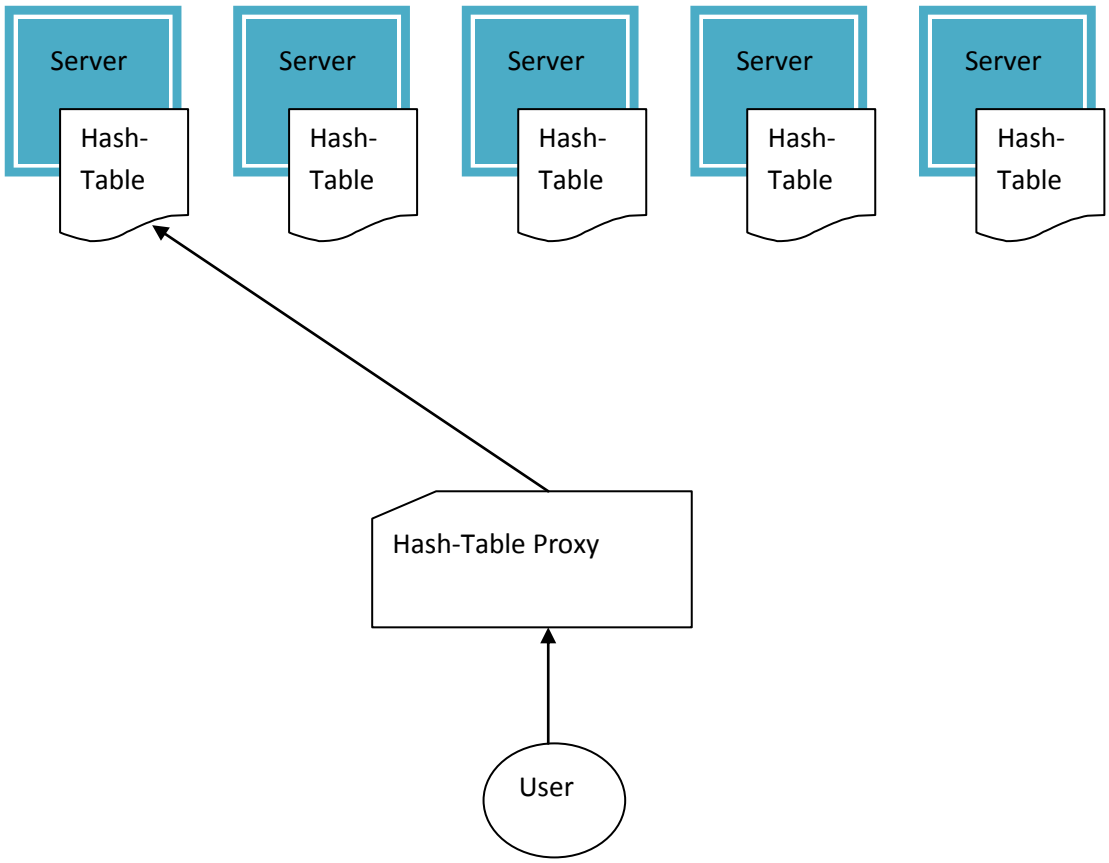
Anastasios Souris (AM 344)

In this short report I describe how I designed and implemented a f-fault tolerant distributed hash-table using OpenReplica. The f-fault tolerant aspect means that the hash-table can tolerate up to f-failures, and the distributed aspect relates to the fact that I have made the design to take advantage of all the servers in a data center, thus increasing the hash-table's performance (i.e parallel request handling) and capacity (i.e storage is distributed among the servers).

Disclaimer: To keep this report short I omit any background related to the methodology that allows an object to be made fault-tolerant (i.e replicated-state-machines and the PAXOS protocol) and the internals of the OpenReplica framework.

Step 1: A Simple f-fault Tolerant Hash-Table

OpenReplica, in fact, works in a very simple way. Given a **sequential** hash-table class written in Python, it transforms this class so that it can become a replicated state machine (RSM). This means that it creates a server-side that will run in 2f+1 servers (so that the RSM is f-fault tolerant), and a client-proxy that the user will use to "connect" to the hash-table RSM. This approach is depicted in the following figure:

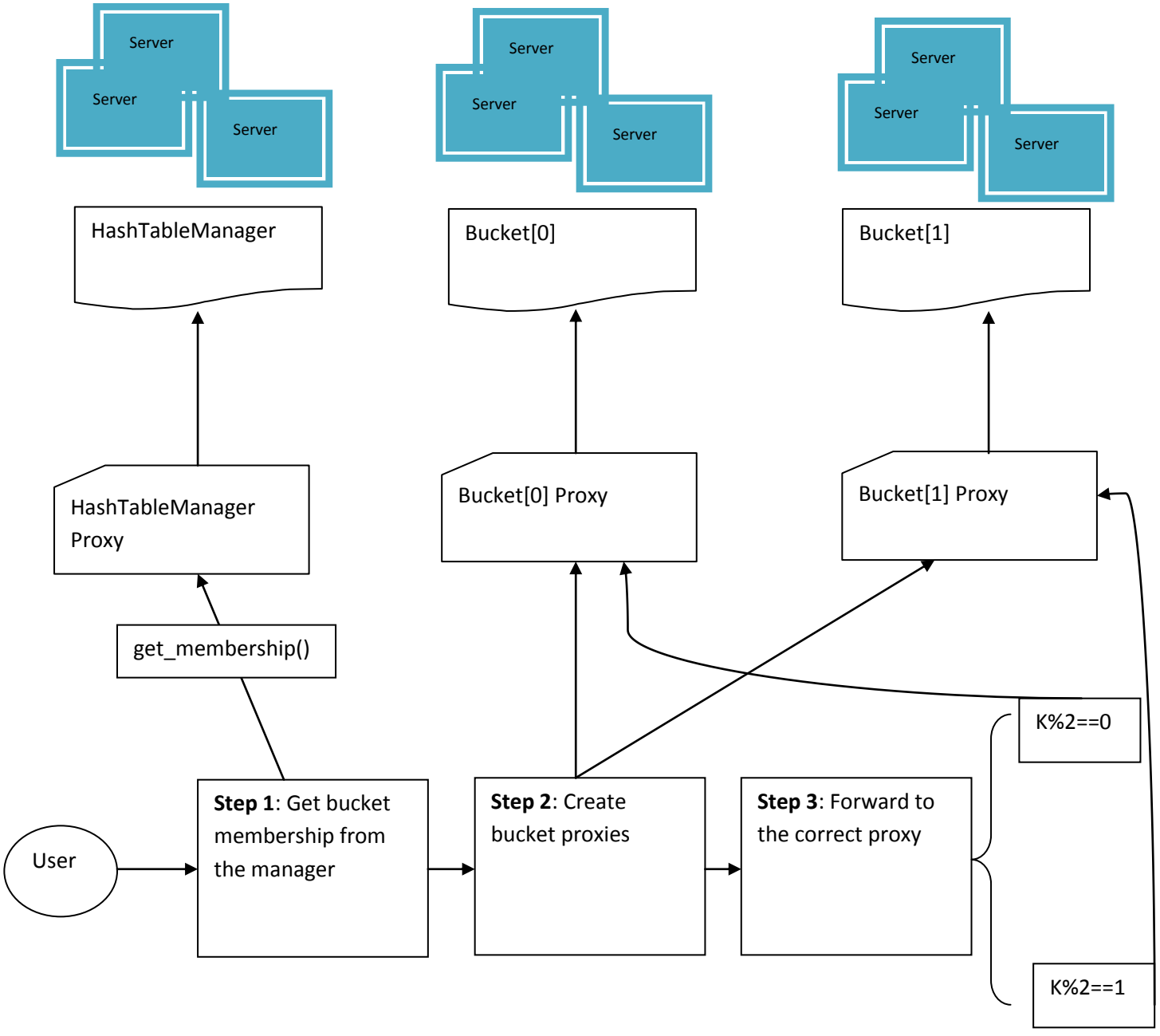


There are two problems with this solution:

1. The capacity of the hash-table is limited to the minimum capacity of the server machines. This is because *each* server stores the entire hash-table.
2. Requests cannot be handled in parallel because there are forwarded to the same RSM which executes the requests in a single total order.

Step 2: A Simple f-fault Tolerant *Distributed* Hash-Table Without Support for Increasing its Capacity: The Static Case

In the next step of my design, I decided to make each *bucket* of the hash-table a RSM. Because a new client must somehow learn the size of the hash-table (that is, how many buckets it contains), and the *name* of each bucket (that is, how it will discover the RSM for that bucket), I have added to my design a *HashTableManager* that returns that information. So now the design looks like that:



In terms of RSMs and the OpenReplica framework, the *only* difference is that now I make the bucket class of the hash-table implementation a RSM (i.e "concoordify" it in terms of OpenReplica) and not the hash-table class itself. I also add a HashTableManager class that keeps a dictionary from bucket indices to *names* (i.e either dns name for that bucket or ip-port pairs for OpenReplica to work). The use case that I am considering here is that the service provider that sets the hash-table, first initiates the bucket replicas and then starts the hash-table manager which it initializes with the names of the bucket replicas it has started. Then the client can use the hash-table (as depicted in the figure above) with these straightforward steps:

1. Contact the HashTableManager and get the membership of the buckets. That is, get how many buckets are there, and for each bucket its RSM name.
2. Instantiate bucket client proxies to the buckets.
3. Forward client requests to the appropriate bucket proxy (i.e using key%size)

The advantage of this approach from Step 1 of my design, is that now i increase both the performance and the capacity of the hash-table because one server will hold the memory required only for its bucket and requests that are forwarded to different proxies will, in general, be answered by different servers and thus can be handled in parallel.

Step 3: Increasing the Capacity

The use-case that I am considering here is that the provider of the hash-table got access to more servers (i.e more servers are added to the data center) and, hence, wants to add those servers to the hash-table. This suggests that the capacity of the hash-table be changed, i.e add one more bucket to the hash-table. I managed to support this functionality with the following restriction: only the service provider can add more buckets to the hash-table and does so in sequential fashion (i.e one bucket at a time – with an exception that I am explaining later in this report). Also, I haven't considered how to shrink the table's capacity, even though in the worst-case if, for some reason, the service provider hash access to less servers that before, it can *migrate* the buckets from the servers he no longer has access to, to the other servers (in this way the size of the hash-table remains the same and no modification is required in the design).

Assume that there are 2 buckets in the hash-table, and the service provider wants to add one more (i.e the new size of the hash-table becomes 3). Also, assume that at the time the new bucket is added the hash-table contains the keys from 1 to 15. So the contents of the buckets are:

- Bucket[0] = 2, 4, 6, 8, 10, 12, 14
- Bucket[1] = 1,3,5,7,9,11,13,15

Now with the addition of the 3rd bucket the contents of the buckets should be (key%3):

- Bucket[0] = 3, 6, 9, 12, 15
- Bucket[1] = 1, 4, 7, 10, 13
- Bucket[2] = 2, 5, 8, 11, 14

The problem with this is that the addition of a single bucket resulted in most of the keys being moved to a new bucket. Some keys moved from 0→1, some from 1→0, and some from (0,1)→2.

This results in major traffic in the network which I want to avoid. There is a simple solution to this problem . What I would like to do is, when I add one more bucket, that bucket to choose a *victim* bucket from which it splits its (key,value) pairs and gets one portion of them. No other buckets should be split. Intuitively, this is done by requiring the size of the hash-table to be doubled; i.e at all times the size of the hash-table be some power of 2 ($2^i, i \geq 1$).

If I add the 4th bucket in the previous example I get the following contents

- Bucket[0] = 4,8,12
- Bucket[1] = 1,5,9,13
- Bucket [2] = 2,6,10,14
- Bucket[3] =3,7,11,15

One can observe that: (a) Bucket[2] got its contents from bucket[0] and (b) Bucket[3] got its contents from bucket[1]. As a result, when I add Bucket[2] I only retrieve data from Bucket[0], and when I add bucket[3] I only retrieve data from Bucket[2].A benefit here is that when I insert bucket[2] only the requests hitting bucket[0] are affected.

To complete the solution I must solve the following problems:

1. Most likely, I want to insert only one more bucket and not to double the buckets. For example, if I already have 256 buckets maybe I would like to add 10-20 more buckets and not 256 more (which would require 256*f additional servers). The scheme presented above works only when the size of the hash-table is a power of two, and, hence, I must deal with this situation.
2. How to inform the HashTableManager of the new bucket membership.
3. How to deal with concurrent requests that are made to a bucket when it is being split.

Solution to Problem 1

Assume that I have a correct way to inform the HashTableManager of the new bucket membership. Then, when I add Bucket[2] to the hash-table manager, what it does is that it assigns Bucket[2] to the new replica names for that bucket, and assigns Bucket[3] to the replica names of Bucket[1]. In that way, until I add Bucket[3], those clients that create a Bucket[3] client proxy will be directed to the replicas of Bucket[1] which as i showed in the previous example contains the contents of Bucket[3]. When the service provider adds Bucket[3] it will split the contents of Bucket[1] into a new bucket. Then, any requests that were using the “old” Bucket[3] that were in fact forwarded to Bucket[1] replicas will be informed that a split is in progress and thus wait until the split is over. Then, they will be forwarded to the new replicas for Bucket[3] that contain contents from Bucket[1] split over to Bucket[3].

Solution to Problem 2

The service provider should provide the replica name of the new bucket. Using the reasoning explained above (*solution to problem 1)*¸the HashTableManager knows how to assign the new replica name to the next bucket and assign the rest of them (to fill in the next power of two) to the appropriate old bucket replicas. Definitely, if I add one bucket which splits from some *victim* bucket when I finish transferring the necessary data from the victim bucket to the new bucket, then I must inform the victim bucket that it is no longer responsible for that portion of the key-space. Now, with a simple argument, I state that I must first inform the victim bucket of this fact and *then* the HashTableManager, because otherwise I would have the following problem:

1. I have a bucket of size 2. Some clients receive from the HashTableManager this setup and create proxies to buckets Bucket[0] and Bucket[1].
2. Now I add Bucket[2] and I split data from Bucket[0].
3. I inform the HashTableManager of Bucket[2]
4. A new client arrives and sees Bucket[2] from the HashTableManager. This new client will forward its requests to Bucket[2].
5. *Before the service provider tells Bucket[0] that it is no longer responsible for the key-space that it moved to Bucket[2]:*
    a. Some old client forwards a request to Bucket[0], which it replies to because it doesn't know yet that for that key-space some other bucket is responsible. Hence I have a incorrect situation.

To avoid the above problem, I have the service provider first inform the victim bucket that it is no longer responsible for some portion of its key-space.

The problem now is that the *old clients* that will try to use Bucket[0] for the key-space that Bucket[2] is now responsible for, will get some kind of exception from Bucket[0] telling them that Bucket[0] is no longer responsible for that key. A simple thing that the client can do in this situation is that it can repeatedly query the HashTableManager until it gets a new view of the hash-table, which it means that the service provider informed the HashTableManager of the new bucket. I also support one more functionality which is based on the following fact:

- When more buckets are added, the buckets form a linked list with the property that the contents of some bucket are moved to buckets following it in that list (i.e following right pointers).

Refer to the appendix for an example of why this is true. This means that when the service provider adds Bucket[2], it tells Bucket[0] that its next bucket is Bucket[2] and tells Bucket[2] that its next bucket is Bucket[1]. This modification is trivial to implement (see appendix) and only requires a bucket to additional store the name of the next bucket (i.e the dns name in case of OpenReplica). This simple modification, though, allows me to support this functionality:

1. Since the service provider will first inform the bucket it means that the bucket will be the first to know which is the next link. So some client can query the victim bucket and hope that it will learn sooner the next bucket and create a proxy to it.
2. The “repeatedly query” solution above makes sense if the client wants to be non-blocking. That is, if it happens to get an exception from some Bucket because a split is in progress it can perform some other things (i.e logging) and then try again. However, if the client wants to be blocking, because it doesn’t have something else to do, I use the *blocking* functionality provided by the OpenReplica framework in this way:
    a. Suppose some client sends a request R to a victim bucket. The victim bucket sees that that requests makes a conflict with a split that is in progress so it blocks the client. This means, it records that a client request is in progress.
    b. When the service provider informs the victim bucket of the next bucket, the victim bucket responds to each client with the link to the next bucket. In that way, the clients avoid having to make an extra roundtrip to get the next link from the HashTableManager.

Let me point that at some point the client must re-load the bucket membership from the HashTableManager to avoid having to follow links each time.

Solution to Problem 3

First, I describe the problem: Assume that I add Bucket[2] and I choose Bucket[0] as the victim bucket. The service provider requests Bucket[0] for a *snapshot* of the appropriate key-space to move to Bucket[2] (and then as I explained earlier will inform Bucket[0] that that portion of the key-space belongs to another bucket). The problem is that while the service provider moves the snapshot to Bucket[2], some client requests may go to Bucket[0] affecting that snapshot. This will make the snapshot that the service provider moved to Bucket[2] invalid. A possible solution is to have Bucket[0] monitor whether there is a request that conflicts with some snapshot of some split that is in progress. Then, when the service provider tells Bucket[0] that the split is over it responds with an exception and some new snapshot. However, because the snapshots are generally larger I do not want the service provider to re-transfer a large amount of data. So, I do the opposite:

- When a victim bucket detects that an *update request* conflicts with a snapshot of some split that is in progress, then:
    1. For a non-blocking client it responds with an exception informing the client that some split is in progress
    2. For a blocking client, it blocks the client which it will respond to afterwards as I explained in “*solution to problem 2*”.

An important optimization is that the victim bucket can always respond to some *get()* operation if a split is in progress because that operation doesn’t change the snapshot.

Appendix

My purpose here is to convince you that the buckets are linked into a list so that i.e all the initial contents of Bucket[1] are split over buckets that follow it in that list. This allows me to provide correctness in the sense that even if a client gets from a bucket an exception it will also get the next link and it can follow that link. Eventually, it will find the correct bucket.

In this *short* report I give the intuition with a simple example. For the long story consult the paper "*Split-Ordered Lists – Lock-Free Resizable Hash Tables"* which introduces the recursive split-order on integer keys that has the properties I am using. However, because, it is based solely on the size of the hash-table being a power of 2, a simple example suffices to illustrate the linked list property:

Assume that I have the size-2 hash-table as in the beginning of this report:

- Bucket[0] = 2, 4, 6, 8, 10, 12, 14
- Bucket[1] = 1,3,5,7,9,11,13,15

Then I get the size-8 hash-table:

- Bucket[0] = 8
- Bucket[1] = 1, 9
- Bucket [2] = 2,10
- Bucket[3] =3,11
- Bucket[4] = 4,12
- Bucket[5] = 5,13
- Bucket[6] = 6,14
- Bucket[7] = 7,15

The implementation makes sure that the buckets form this list:

Bucket[0] → *Bucket[4]* → *Bucket[2]* → *Bucket[6]* → *Bucket[1]* → *Bucket[5]* → *Bucket[3]* → *Bucket[7]*

One can easily verify that the initial contents of Bucket[1] are split to buckets that follow it in the linked list, that is in buckets Bucket[5], Bucket[3] and Bucket[7]. NOT in previous buckets in that list.

Limitations

In this last section, I want to point out some more limitations of my design.

First, I haven't made the service provider fault-tolerant in my implementation. However, I am certain that it can be done in a straightforward way by having the service provider log the last operation it performed and then continue from there.

Lastly, I want to mention that if the hash-table size is M, then most probably the service provider can add M more buckets in parallel because these affect separate portions of the list I presented above – that is, each of the new buckets splits from a different bucket from the initial M. This is a simple argument which I haven't checked and I haven't implemented in my prototype.