Design Document

Table of Contents

1 Requirements 1

1.1 Distributed Hash Table 1

1.1.1 Server thread 1

1.1.2 Client thread 1

1.2 Hashing Functions 2

1.2.1 Client’s hashing Function 3

1.2.2 Server’s Hashing Function 3

1.3 Replication 3

1.3.1 server\_config 3

1.3.2 replica\_server\_config 4

1.3.3 Data Structures 4

1.3.4 API’s 4

1.4 Code flow 6

1.4.1 PUT 6

1.4.2 GET 6

1.4.3 DELETE 8

# Requirements

## Distributed Hash Table

In this project, we are designing a distributed hash table over 8 peers. The peers will serve as clients as well as servers. When the peer starts, it will read a global configuration file to read the IP: PORT combination where it should listen as a server.

There would be following two threads:

### Server thread

This thread will maintain a hash table of 1M entries. The hash table for this assignment is a simple array of 1M.

* **Server will run a select() linux call to listen to the current open client fd’s.**
* **Clients will establish socket connection to a peer server will be established only once i.e. during the first request. After that, the socket will be kept alive till the time peer/server shuts down.**

### Client thread

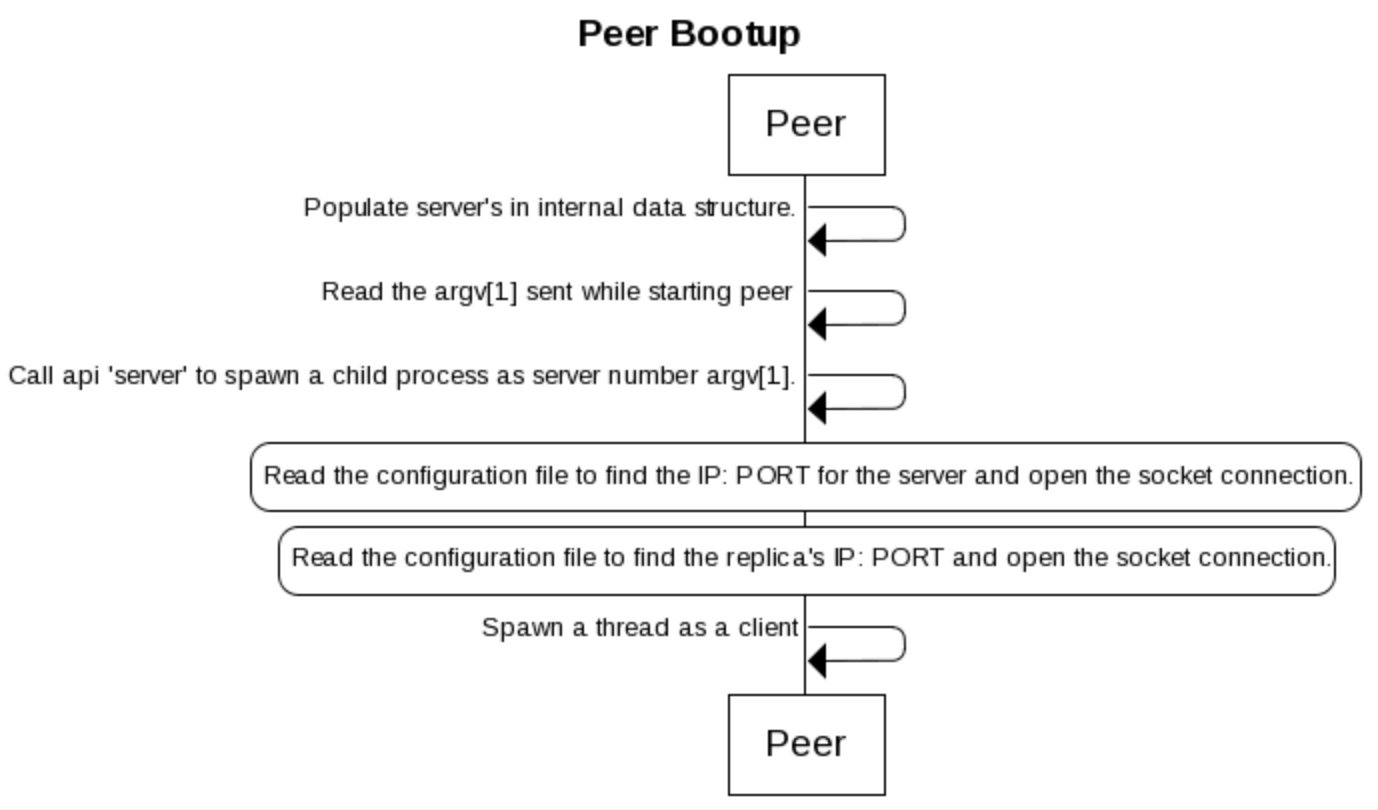
This thread will present the users with following options:

1. \*\*\* Enter 1 for inserting an entry \*\*\* 🡪 PUT
2. \*\*\* Enter 2 for getting an entry \*\*\* 🡪 GET
3. \*\*\* Enter 3 for deleting an entry \*\*\* 🡪 DEL
4. \*\*\* Enter 4 to exit \*\*\* 🡪 EXIT

Along with sending the request to the primary server, these API’s would send a request to the replica as well to insert, delete or get the entry.

In case of insertion or deletion, the same entry would be inserted or deleted respectively in the replica as well.

While retrieving, primary server is down, secondary server would be contacted. An error would be returned if the secondary server could not be reached as well.



## Hashing Functions

### Client’s hashing Function

Client will call put/get/delete API’s. These API’s will internally calculate the server to whom which the request should be sent. This is done via the following API :

int get\_hashing\_server (char \*key) {

int hash = 0;

        char \*temp = strdup(key);

        while (\*temp != '\0') {

                hash = hash + \*temp;

                temp++;

        }

        temp = NULL;

        return hash % NUM\_SERVERS;

}

This function will calculate the sum of the ASCII equivalent of all the characters and then mod it with the number of servers. The final result shall then be the server to which the request should be sent.

### Server’s Hashing Function

At server side, a simple hashing function is implemented to calculate the hash for the incoming key and save it in the hash map:

int server\_compute\_hash(char \*key) {

int hash = 0;

char \*temp = strdup(key);

while (\*temp != '\0') {

hash = hash + \*temp;

temp++;

}

temp = NULL;

return hash % MAX\_HASH\_ENTRIES;

}

Again, this function will calculate the sum of the ASCII equivalent of all the characters and then mod it with the number of entries the server can store to generate the right place to insert the key in the hash table.

## Replications

We have defined two configuration files to bootstrap the network:

### server\_config

Configuration file containing IP: PORT for the servers.

Example in the following file, IP: PORT of first server would be 127.0.0.1 1231:

nikatari$ cat server\_config

127.0.0.1 1231

127.0.0.1 1232

127.0.0.1 1233

127.0.0.1 1234

127.0.0.1 1235

127.0.0.1 1236

127.0.0.1 1237

127.0.0.1 1238

### replica\_server\_config

Configuration file containing IP: PORT for the servers. The way we are maintaining server’s IP and port statically in a configuration file, we would be maintaining replica’s configuration in a file as well. This configuration file contains IP: PORT of the replica servers.

Example in the following file, IP: PORT of first server’s replica would be 127.0.0.1 1232:

nikatari$ cat replica\_server\_config

127.0.0.1 1232

127.0.0.1 1233

127.0.0.1 1234

127.0.0.1 1235

127.0.0.1 1236

127.0.0.1 1237

127.0.0.1 1238

127.0.0.1 1231

### Data Structures

|  |  |
| --- | --- |
| /\* Data structure to send data to server from client \*/  struct client\_data {  char \*ip;  char \*port;  char \*(\*hash\_table)[1];  };  /\* FD's for primary and secondary replica's  \* First column will have primary's fd and  \* second column will have secondary's fd  \*/  int server\_fds[NUM\_SERVERS][1][2]; | /\* Data structure used internally by server to spawn a thread to Put/Get/Delete \*/  struct server\_data {  int client\_fd;  char \*msg;  char \*(\*hash\_table)[1];  };  /\* Lock for all hash operations \*/  pthread\_mutex\_t lock = PTHREAD\_MUTEX\_INITIALIZER;  /\* Global structure to save the replica's config's \*/  char \*replica\_servers[NUM\_SERVERS][2]; |

### 

### API’s

/\* Populate the servers after reading from server\_config \*/

int populate\_servers (char \*servers[NUM\_SERVERS][2]);

/\* Populate the servers after reading from replica\_server\_config \*/

int populate\_replica\_servers (void);

/\* Function used by servers to compute the hash \*/

int server\_compute\_hash (char \*key);

/\* Function used by put function to determine

\* the server where entry should be

\* inserted/retrieved/deleted \*/

int get\_hashing\_server (char \*key);

/\* Function to start a peer as a server \*/

void \*server (char \*ip, char \*port, char \*hash\_table[12500][1]);

/\* Function to insert an entry to the hash table \*/

char \*put (char \*key ,char \*value, char \*servers[NUM\_SERVERS][2]);

/\* Function to get an entry from the hash table \*/

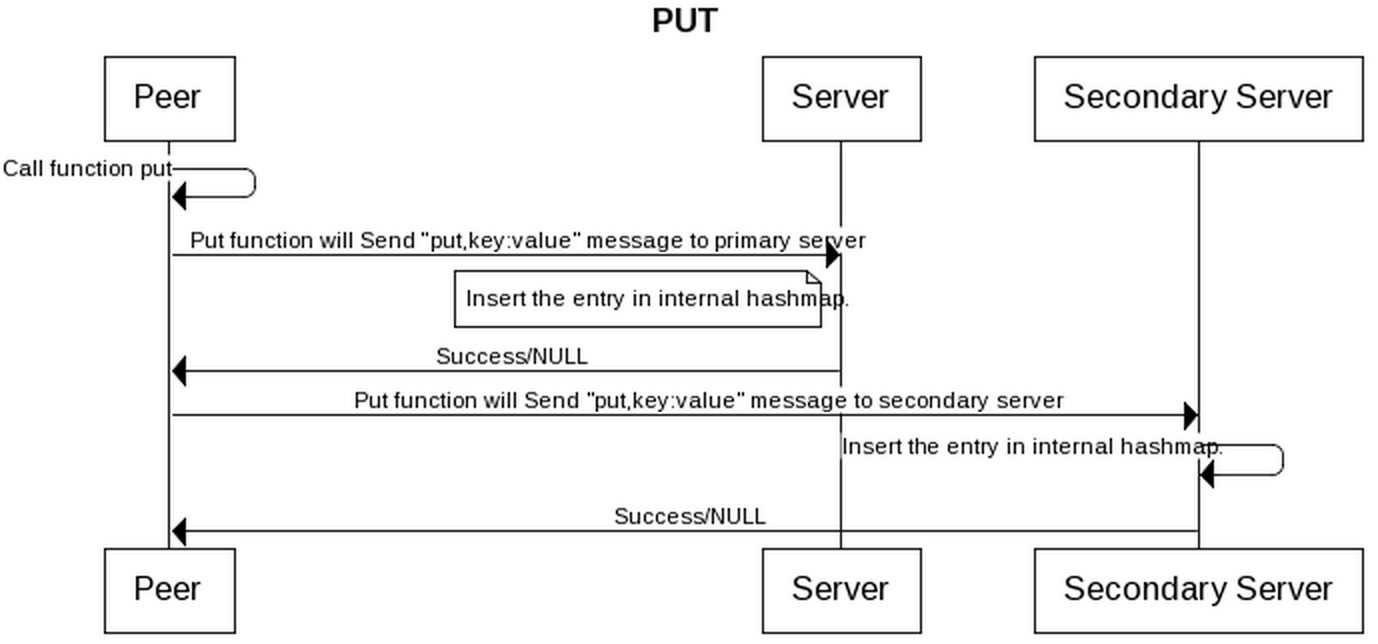
char \*get (char \*key, char \*servers[NUM\_SERVERS][2]);

/\* Function to delete an entry from the hash table \*/

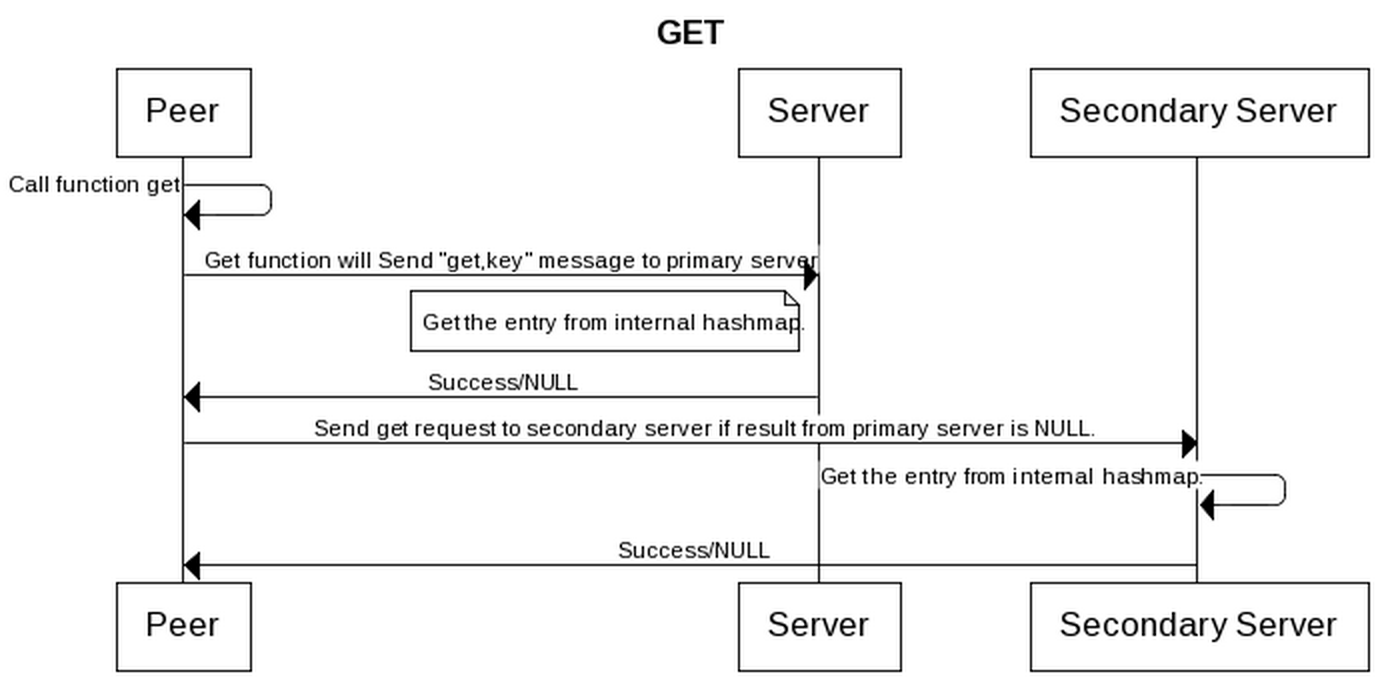
char \*del (char \*key, char \*servers[NUM\_SERVERS][2]);

## Code flow

### PUT



### GET



### DELETE

### 

End of Document