Distributed System Final Report

Fault-Tolerant Key-Value Server Client Using MPI (Redis Clone)

University of Science and Technology of Hanoi

February, 2022

Contents

1 Introduction

1.1 Definition

1.2 Redis and key-value databases

A key-value database, sometimes called a key-value store, is a type of database that stores data in key-value format. The key is unique and is mapped to a specific value. It is familiar to refer to this type of key-value format as a dictionary or a hash table; however, the value can be stored in different data structures, not just a simple string. We can add/update and remove key-value pairs in a key-value store, query key and retrieve its corresponding value.

Redis stands for Remote Dictionary Server, which serves as an in-memory key-value store. It is well-known for its fast read and writes operations since the data are stored in caches. Redis supports different values such as list, set, hash, sorted set, and streams. Redis can be used on top of Node-JS to speed up the query process in web applications.

1.3 Fault-tolerance

Fault-tolerance is the ability of the system to "survive" and continue functioning in the face of failures. The failure varies from network failure, hardware infrastructure failure, and machine crashes. Redis Sentinel is a mechanism to set up and run Redis in a fault-tolerant manner by continuously checking and verifying whether the master server goes down and finds a way to vote for the new master.

1.4 Message Passing Interface

MPI(Message Passing Interface) is a library designed to create programs that can run efficiently on most parallel architectures. In the message-passing model of parallel computation, the processes executing in parallel have separate address spaces. Communication occurs when a portion of one process's address space is copied into another process's address space. This operation is cooperative and occurs only when the first process executes send operations and the second process executes a receive operation.[1]

1.5 Authors

Member	**Student ID**
Nguyen Xuan Tung	BI10-188
Nguyen Quang Anh	BI10-012
Lu Khanh Huyen	BI10-083
Tran Hong Quan	BI10-149
Vu Duc Chinh	BI10-024

2 Analysis and Design

Nowadays, there is a common trade-off pattern in distributed systems; in order to gain high performance, many systems split the databases into several blocks and store them in different servers. However, this will increase the possibility of failures. To solve this problem, a replication strategy was introduced to provide high availability of the data.

2.1 Primary/Secondary Replication

Primary/Secondary Replication is a mechanism where the primary replica forwards the operations sent from the client to the secondary replica to achieve the same state.

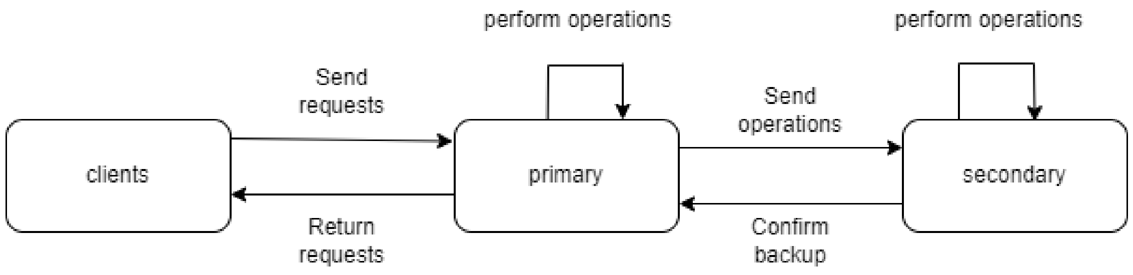


Figure 1: Primary/Backup Replication[2]

We follow the replicated state machine strategy: the system must ensure that all servers have the same states after every client's operation.

2.2 Design

In the beginning, the primary and backup server write their port address to the file. Afterward, the client will read the address from the file and connect to the primary server. The client then starts sending the request to the server, and two scenarios will happen.

2.2.1 Normal Scenario

The client sends a request to the primary server(e.g., GET/PUT/DELETE).

If the client sends a GET request, the primary server will return its corresponding value to the client.

If the client sends a PUT request, the primary server will add the key-value data to its database and forward the data and the operation to the secondary server.

If the client sends a DELETE request, the primary server will remove the key-value data from the database and forward the key and the operation to the secondary server.

The secondary server performs the same operation as the primary server and sends an acknowledgment message to the primary server.

The primary server writes the address of the port name to the file, so the client will know which port they will connect in the next operation.

The primary server will send a successful PUT/DELETE message or the value from the key to the client.

2.2.2 Failure Scenario

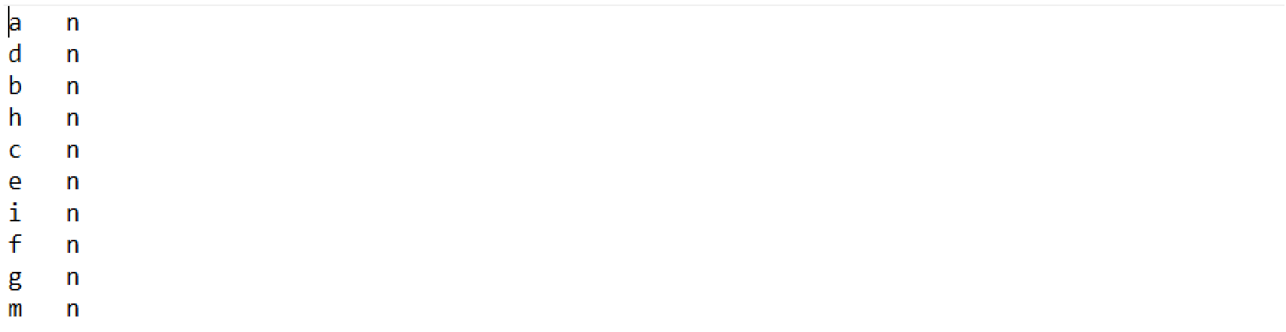
Since the failure scenario is hard to implement, we implement a random function to simulate the case of failure. We randomize the number from 0 to 10, and if the number is greater or equal to 8, then the failure occurs.

These are the steps we design after the failure happens:

- The primary server sends the random number to the secondary server.
- After receiving the number from the primary server, the secondary server will know that it will become the primary server.
- The two servers' states are still synchronized; however, the previously sent operation from the client would not be executed. The client might have to send back the operation.
- The new primary server will write its address port to the file called "primaryserveraddress.txt."
- The client will now read the file and know which server it will connect to in the next operation.

3 Implementation

We have a database file called "db.txt." This is where the server loads and dumps the data at the beginning and the end of the operations. It has a structure like this:



```
a	n
d	n
b	n
h	n
c	n
e	n
i	n
f	n
g	n
m	n
```

Figure 2: Database structure in file

At the beginning of the project, the backup server and the primary server open their port, write the port name to the file, and load the database to an unordered map.

```
//writes port address to the file
MPI_Open_port(MPI_INFO_NULL, portname);
fp = fopen("portname.txt", "w");
fprintf(fp, "%s\n", portname);

//load the database from file
std::ifstream input(filename);
for (std::string eachLine; getline(input, eachLine); ){
  std::istringstream strm(eachLine);
  std::string a, b;
  while (strm >> a >> b) {
    db[a] = b;
  }
}

MPI_Open_port(MPI_INFO_NULL, portname);
fp = fopen("a.txt", "w");
fprintf(fp, "%s\n", portname);
fp = fopen("b.txt", "r");
fread(portnameb, 1, 53, fp);
fclose(fp);
fp = fopen("c.txt", "r");
fread(portnamec, 1, 53, fp);
fclose(fp);
```

```
MPI_Comm_connect(portnameb, MPI_INFO_NULL, 0,
 ↪  MPI_COMM_WORLD, &a);
```

The client then chooses between 3 options: 1 to get value from key, 2 to add/update value, 3 to delete key and its corresponding value.

```
std::cin >> option;
switch (option) {
    case 1: {
      MPI_Send(&option, 1, MPI_INT, 0, 2001, a);
      MPI_Send(&key_db, 100, MPI_CHAR, 0, 2001, a);
      MPI_Comm_accept(portnamenow, MPI_INFO_NULL, 0,
       ↪  MPI_COMM_WORLD, &a);
      MPI_Recv(&value_db, 100, MPI_CHAR, MPI_ANY_SOURCE,
       ↪  2004, a, &status);
      std::cout << "Value of " << key << " is " <<
       ↪  value_db << std::endl;
    }
    case 2: {
      MPI_Send(&option, 1, MPI_INT, 0, 2001, a);
      MPI_Send(&key_db, 100, MPI_CHAR, 0, 2001, a);
      MPI_Send(&value_db, 100, MPI_CHAR, 0, 2001, a);
      sleep(1.0);
      while (fopen("server_primarynow.txt", "r") != NULL)
       ↪  {
        fread(portnamenow, 1, 53, fp);
        fclose(fp);
        break;
        }
        MPI_Comm_accept(portnamenow, MPI_INFO_NULL, 0,
         ↪  MPI_COMM_WORLD, &a);
    }
      case 3: {
        MPI_Send(&option, 1, MPI_INT, 0, 2001, a);
        MPI_Send(&key_db, 100, MPI_CHAR, 0, 2001, a);
        sleep(1.0);
        while (fopen("server_primarynow.txt", "r") != NULL)
         ↪  {
          fread(portnamenow, 1, 53, fp);
          fclose(fp);
          break;
        }
        MPI_Comm_accept(portnamenow, MPI_INFO_NULL, 0,
         ↪  MPI_COMM_WORLD, &a);
        break;
      }
```

The primary server receives the key or value from the client and does the corresponding operation requests by the client.

The first operation is get value by key.

```
value = db[key_db];
strcpy(value_db, value.c_str());
MPI_Comm_connect(portnamec, MPI_INFO_NULL, 0,
 ↪  MPI_COMM_WORLD, &b);
MPI_Send(&option, 1, MPI_INT, 0, 2002, b);
```

```
MPI_Comm_connect(portnamea, MPI_INFO_NULL, 0,
 ↪  MPI_COMM_WORLD, &b);
MPI_Send(&value_db, 100, MPI_CHAR, 0, 2004, b);
```

The second operation is add/update with the corresponding key and value.

```
fp = fopen("primarynow.txt", "w");
fprintf(fp, "%s\n", portname);
db[key_db] = value_db;
MPI_Send(&option, 1, MPI_INT, 0, 2002, b);
MPI_Send(&random_num, 1, MPI_INT, 0, 2002, b);
MPI_Send(&key_db, 100, MPI_CHAR, 0, 2002, b);
MPI_Send(&value_db, 100, MPI_CHAR, 0, 2002, b);
MPI_Recv(&buf, 1, MPI_CHAR, MPI_ANY_SOURCE, 2003, b,
 ↪  &status);

MPI_Comm_connect(portnamea, MPI_INFO_NULL, 0,
 ↪  MPI_COMM_WORLD, &b);
MPI_Send(&buf, 1, MPI_CHAR, 0, 2004, b);
```

The third operation is delete the corresponding key-value pair.

```
fp = fopen("primarynow.txt", "w");
fprintf(fp, "%s\n", portname);
db.erase(key_db);
MPI_Send(&option, 1, MPI_INT, 0, 2002, b);
MPI_Send(&random_num, 1, MPI_INT, 0, 2002, b);
MPI_Send(&key_db, 100, MPI_CHAR, 0, 2002, b);
MPI_Send(&value_db, 100, MPI_CHAR, 0, 2002, b);
MPI_Recv(&buf, 1, MPI_CHAR, MPI_ANY_SOURCE, 2003, b,
 ↪  &status);
MPI_Comm_connect(portnamea, MPI_INFO_NULL, 0,
 ↪  MPI_COMM_WORLD, &b);
MPI_Send(&buf, 1, MPI_CHAR, 0, 2004, b);
```

The random function to simulate the probability of failure tolerance is:

```
int randomRange(int min, int max) //range : [min, max]
{
  static bool first = true;
  if (first)
  {
    srand(time(NULL)); //seeding for the first time only!
    first = false;
  }
  return min + rand() % ((max + 1) - min);
}
```

We have a block of code to decide whether the server will be the primary or backup server.

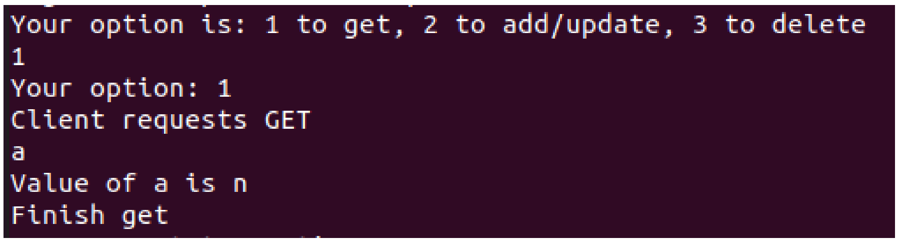
```
if (randomnum > 8 && isprimary == 0) {
  isprimary = 1;
}
else if (randomnum > 8 && isprimary == 1) {
  isprimary = 0;
}
```

4 Results

4.1 Normal scenarios

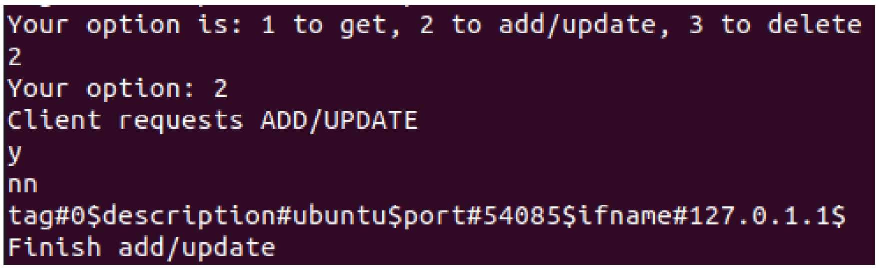
When no failure happens, we got the following results.

In the client side, when the clients request operation, the server will return the value of the key or the message depends on whether the failure occurs or not.



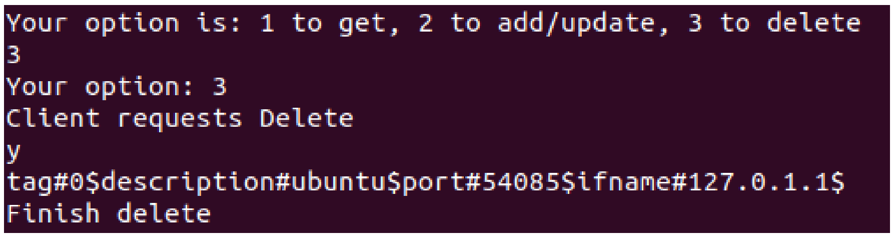
```
Your option is: 1 to get, 2 to add/update, 3 to delete
1
Your option: 1
Client requests GET
a
Value of a is n
Finish get
```

Figure 3: Client's GET request



```
Your option is: 1 to get, 2 to add/update, 3 to delete
2
Your option: 2
Client requests ADD/UPDATE
y
nn
tag#0$description#ubuntu$port#54085$ifname#127.0.1.1$
Finish add/update
```

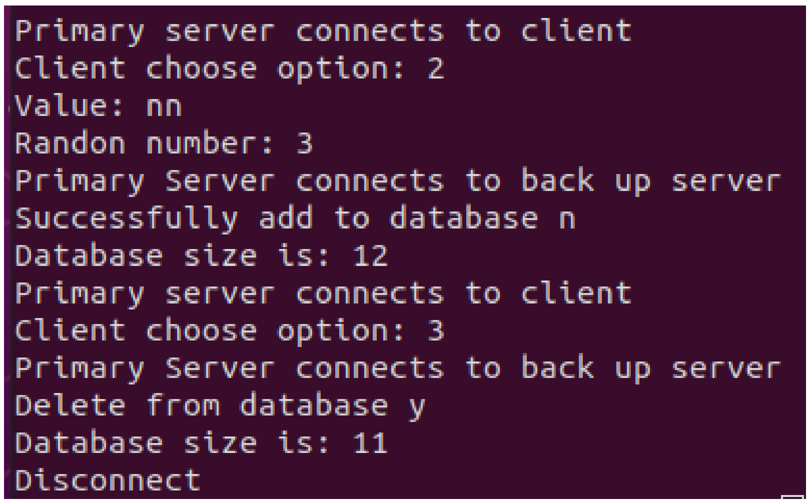
Figure 4: Client's ADD/UPDATE request



```
Your option is: 1 to get, 2 to add/update, 3 to delete
3
Your option: 3
Client requests Delete
y
tag#0$description#ubuntu$port#54085$ifname#127.0.1.1$
Finish delete
```

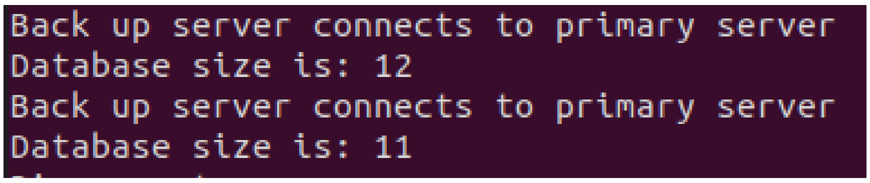
Figure 5: Client's DELETE request

These are the logs displayed on the primary server and the secondary server.



```
Primary server connects to client
Client choose option: 2
Value: nn
Randon number: 3
Primary Server connects to back up server
Successfully add to database n
Database size is: 12
Primary server connects to client
Client choose option: 3
Primary Server connects to back up server
Delete from database y
Database size is: 11
Disconnect
```

Figure 6: Primary server's log

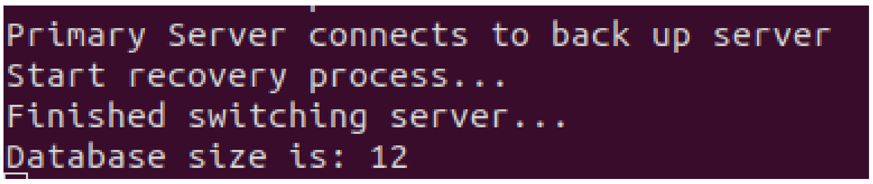


```
Back up server connects to primary server
Database size is: 12
Back up server connects to primary server
Database size is: 11
```

Figure 7: Secondary server's log

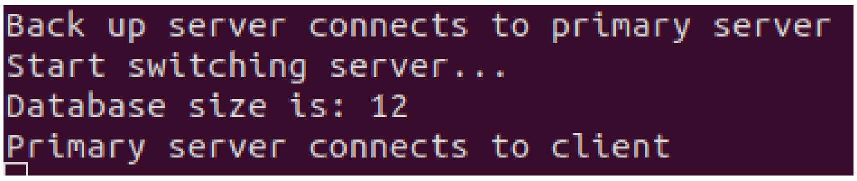
4.2 Failure scenarios

These are the logs displayed on the primary server and the secondary server when failure happens.



```
Primary Server connects to back up server
Start recovery process...
Finished switching server...
Database size is: 12
```

Figure 8: Primary server's log



```
Back up server connects to primary server
Start switching server...
Database size is: 12
Primary server connects to client
```

Figure 9: Secondary server's log

5 Conclusion and Future work

5.1 Conclusion

In this project, we have applied Message Passing Interface to create one client and two servers. We also successfully implemented primary/secondary replication, the simplest way of replication.

5.2 Future work

Many adaptations extensions have been left for the future due to lack of time. We only implement one backup server in this project instead of several backup servers as in a real-life application. We also do not implement failure scenarios such as network failure or machine crash. We may try to implement a metadata server that can keep track of the system servers, detect failures, and coordinate the recovery process. In short, implementing the metadata server and forcing the server to send the heartbeat message to it to detect failures instead of forcing the primary server to send the random number to the secondary server would remove all of our drawbacks.

References

[1] William Gropp, Ewing Lusk, and Anthony Skjellum. 2014. Using MPI: Portable Parallel Programming with the Message-Passing Interface. The MIT Press.

[2] https://levelup.gitconnected.com/deep-dive-into-primary-secondary-replication-for-fault-tolerance-6ba203b06901