Fault-tolerant key-value server client using RPC

Group members:

- Nguyen Duc Anh (BI9-041)
- Doan Minh Long (BI9-145)
- Tran Khanh Duong (BI9-078)
- Vu Long Dung (BI9-070)
- Tran Ngoc Diep (BI7-033)

A. Problem Description

Fault tolerance is the property that enables a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components [1].

With the implementation of a fault-tolerant design, a system should be able to continues its intended operation, possibly at a reduced level, rather than failing completely, when some part of the system fails [2].

When outlining plans for the project, we specify that this service needs to satisfy these requirements:

1. The service maintains a database of key/value pairs.
2. The fault-tolerant solution must be software-based, not hardware based *(Solutions such as RAID, redundancy for the PSU and RAM and redundant NICS are out of the question due to limited budget).*
3. During a system failure, the service should use a backup system immediately (preferrably in less than 20 seconds). Therefore, at least two servers are needed.
4. The backup system will sync data from the primary server continuously and asynchronously. Due to the asynchronous nature, backup servers may return data that does not reflect the state of the data on the primary.

   > If the primary dies and contains data that backups do not have, then the primary should revert that write operations to maintain data consistency across servers.

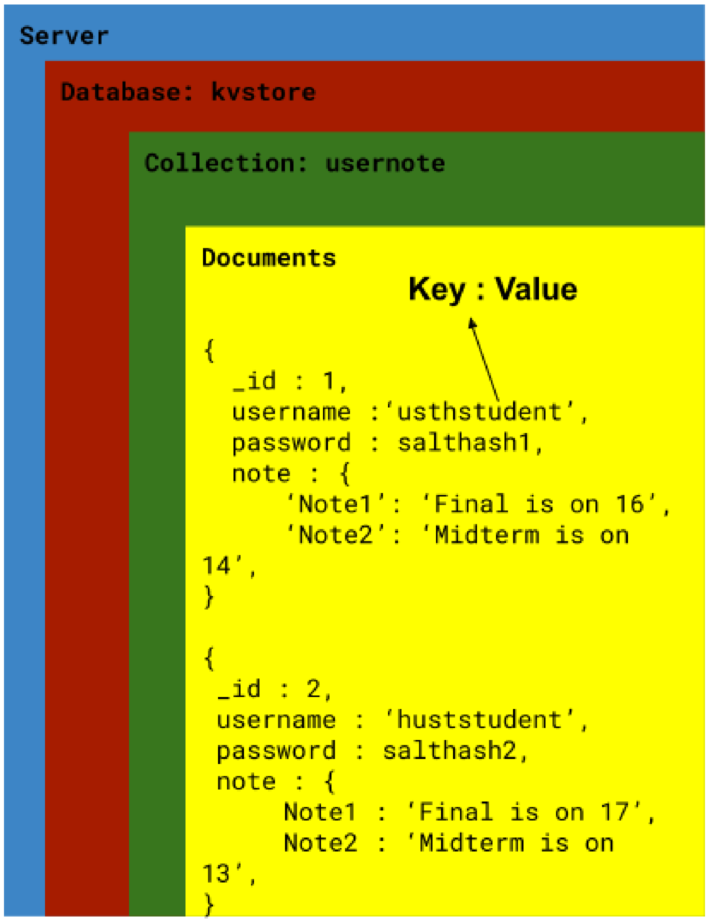
5. The primary server will always maintain a log with a specified max size in case the backup server goes offline so it can resync new data being inserted during downtime.
6. There will always be a small curtain of time between the transition of primary and backup servers. We have to figure out a way to deal with client incoming requests during this state.
7. The servers will implement *Remote Procedure Call* to facilitate clients to perform SET, GET and PUT operations onto the database.

B. Design

Taking into account the above requirements, our team decides to implement **MongoDB** into the database server. It fulfils the aforementioned requirements in the following aspects:

1. A key-value store is a storage system that stores values indexed by a key. You're limited to query by key and the values are opaque, the store doesn't know anything about them. **MongoDB** is a document database which extends the concept of the key-value database. In a document database, the value contains structured or semi-structured data [**Figure 1**].

   Originally, only a key-value database is required. But we want to implement a database which has a username and password access authentication method. Each users will have three keys: *username, password and note.* Users will be able to perform operations such as setNote, getNote, etc… if they provide credentials. MongoDB is suitable for this.



```
{
  _id : 1,
  username :'usthstudent',
  password : salthash1,
  note : {
      'Note1': 'Final is on 16',
      'Note2': 'Midterm is on
14',
}

{
 _id : 2,
 username : 'huststudent',
 password : salthash2,
 note : {
      Note1 : 'Final is on 17',
      Note2 : 'Midterm is on
13',
}
```

Figure 1: Database Model

2. It is a software solution for the fault tolerance problem.
3. MongoDB has a concept of replication which provides redundancy and increases data

availability [**Figure 2**]. With multiple copies of data on different database servers, replication provides a level of fault tolerance against the loss of a single database server [3]. *If the primary is unavailable, the replica set elects a secondary to be primary and continues normal operation. The old primary rejoins the set when available.*

> We choose to have one primary server along with two secondaries servers. As a result, so if primary server crashes, we essentially have two backup servers.

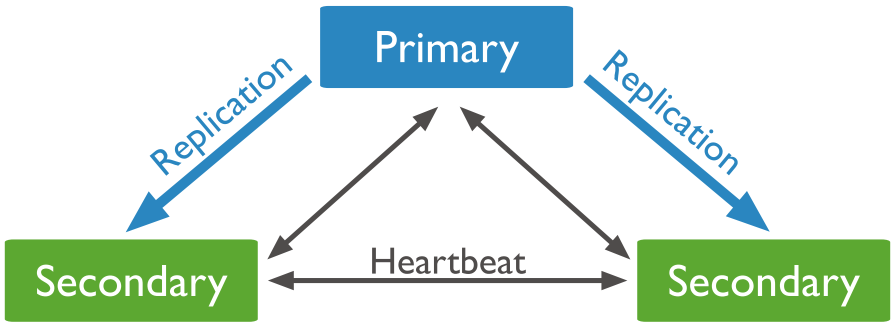


Figure 2: Replication

4. MongoDB replication allows secondary (or backup) servers to sync continuously and asynchronously. Furthermore, it has a 'rollback' functionality to revert write operations for keeping consistency.

   > In MongoDB, rollback is rare. Because it is able to specify for primary to only respond to client requests only after a specified amount of secondary has replicated new data successfully [**Figure 3**].

5. Each members in MongoDB replica set has an oplog which keeps a rolling record of all operations that modify the data stored in the database. Backups copy oplog from primary.

   > Considering the scope of our project, there is not much traffic between client and the servers. For demonstration purpose, let's say there are 100 write requests per hour. That means 2400 requests a day. Each write operations should not exceed 1KB of storage (total characters of username, password and notes are limitted to be below 600). So the minimum size of oplog will be 2.4 MB. **10 MB** of oplog should be plenty to work with.

6. During graceful degradation, MongoDB will block all WRITE requests from clients. READ requests however can still be routed to secondary servers during this time by setting read preferences to **primaryPreferred** [**Figure 4**].

7. Python has the package **XMLRPC** to handle *Remote Procedure Call.*

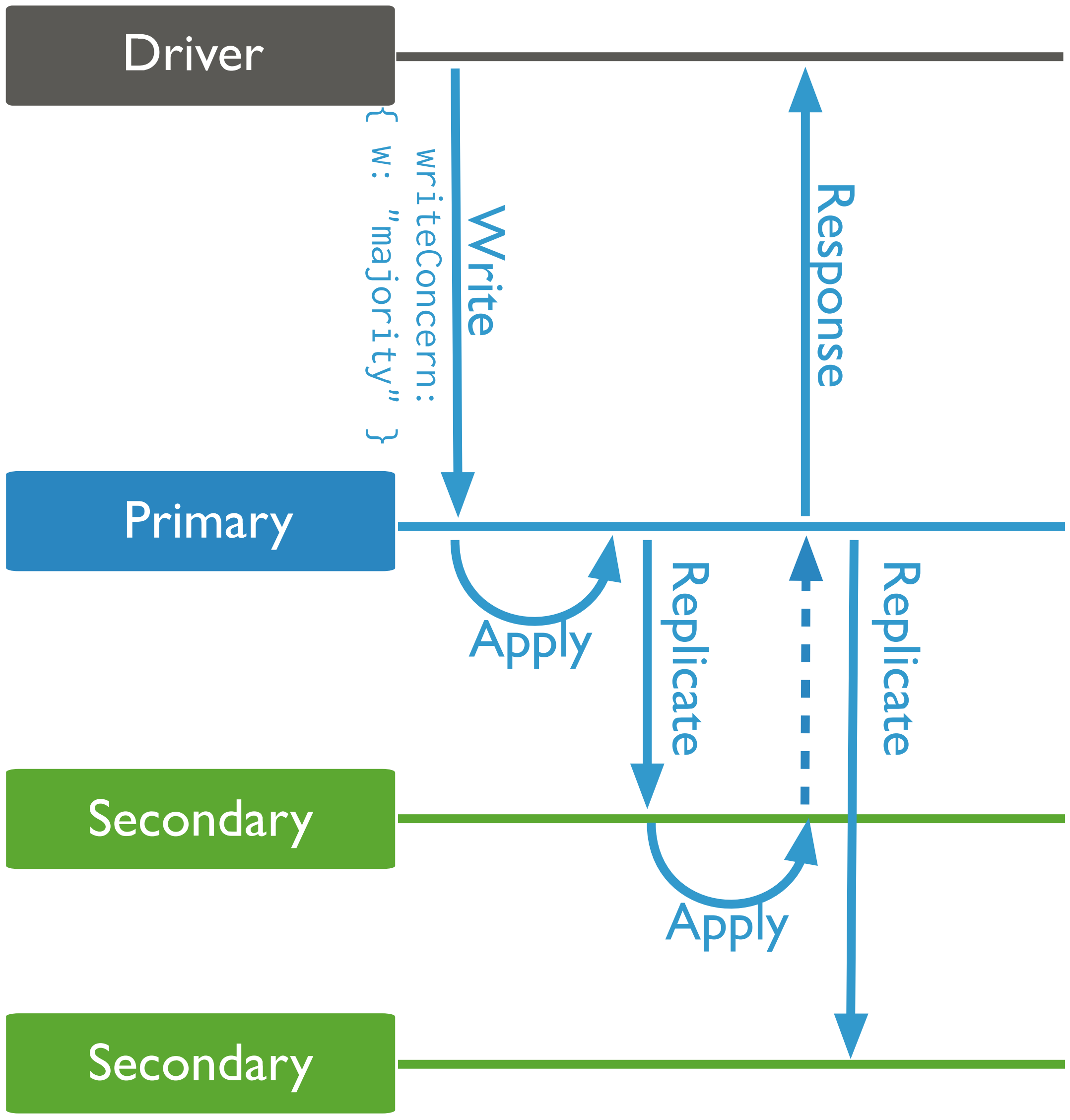


Figure 3: Write Majority

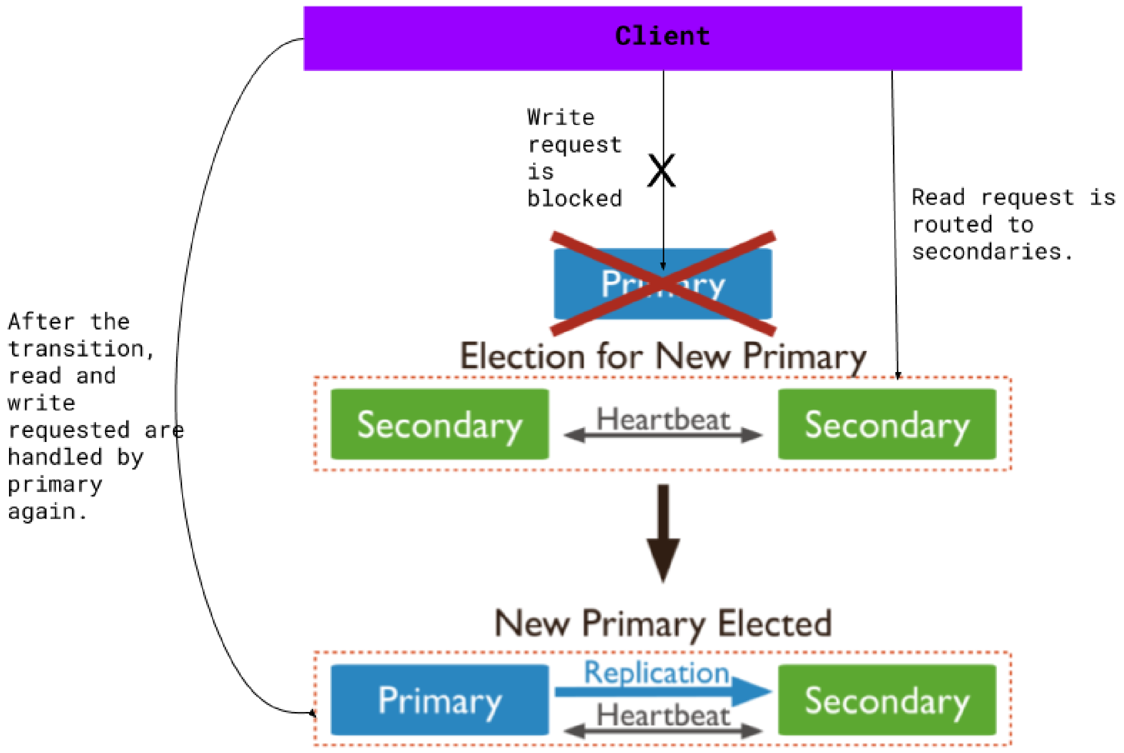


Figure 4: Write and Read Requests during graceful degradation

C. System Organization

Ideally, three of the database servers should have different static IPs, but since most of our home IPs are behind ISP NAT, we do not have our public IP address. Some of our ISPs even restrict port forwarding in their NAT router. The final decision is to have two IP addresses for demonstrating the system. One serves as a client and one serve as a XMLRPC server at port 12345 and database servers at 3 different ports: 27017, 27018, 27019 [**Figure 5**].

We wrote a script to populate the primary server with data for testing purpose. The data will then be replicated to secondary servers:

> First, use this guide I've written to set up three mongod instances with replication enabled: [Link]. Then populate data with the script *kvstore.py*.

D. Implementation

In the XMLRPC server, there are three functions that will serve similarly as controller between client calls and the database servers:

- ***register:***

  When clients call register, they will provide two parameters: username and password. The XMLRPC server has a connecting client to the MongoDB replication set will call

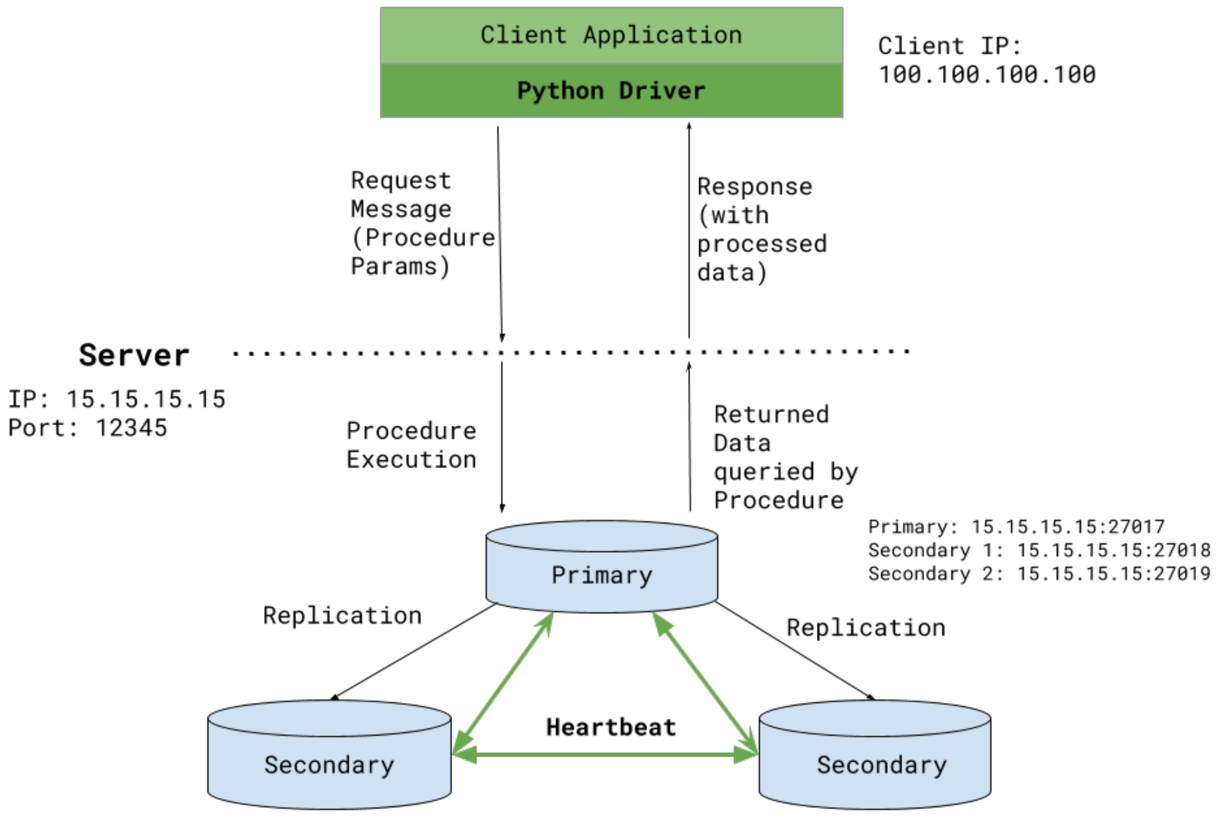

Client Application
Python Driver
Client IP: 100.100.100.100
Request Message (Procedure Params)
Response (with processed data)
Server
IP: 15.15.15.15
Port: 12345
Procedure Execution
Returned Data queried by Procedure
Primary: 15.15.15.15:27017
Secondary 1: 15.15.15.15:27018
Secondary 2: 15.15.15.15:27019
Primary
Replication
Replication
Heartbeat
Secondary
Secondary

Figure 5: System Organization

a function to insert these information to the database.

```
if (len(username) <= 0 or len(username) > 50):
    return "The minimum length for username is 1 andthe   maximum length is 0"
if not re.match('^[a-zA-Z0-9_]+$', username):
    return "Only letters, digits, and underscores (_)are  allowed in the username"
user_doc = userpass.find_one({"username": username})
if (user_doc != None):
    return ("The username already exists")
else:
    userpass.insert_one(
        {"username": username, "password": hashPas(password), "note": []})
    return("Registered successfully")
return("Function run")
```

Before actually inserting user input into the database, the username is checked if it's correctly adhered to the required standard (*must contain only specified characters*). The password is hashed using the function *hashPass*, which leverages the **bcrypt** library.

- ***setNote:***

  Four parameters are needed from clients:

  - *username*
  - *password*
  - *title*
  - *content*

```
user_doc = verifyPassword(username, password)
if (user_doc == False):
    return "Username or password is incorrect."
elif (len(newnote) <= 1 or len(newnote) > 500):
    return "Note length must be between 1 and 500."
else:
    # Filter $ to prevent nosql injection
    newnote = json.loads(sanitize(newnote))
    key = list(newnote)[0]
    value = newnote[key]
    update_key = "note." + str(key)
    userpass.update_one({"username": username}, {
                        "$set": {update_key: value}})
    return "You have added new content to the note."
```

First we verify username and password. If the username exists, the Note is checked if its length is between 1 and 500 characters. If this is true, the note will be filtered to prevent NoSQL injection, broken down to note title and note content then inserted to MongoDB.

PS: Title and content parameters are combined into a Json format in client side and sent to the server as one argument.

- ***getNote:***

  Two parameters are needed: username and password.

```
user_doc = verifyPassword(username, password)
if (user_doc == False):
    return "Username or password is incorrect"
else:
    if (user_doc["note"]):
        data = json.dumps(user_doc["note"])
        return('Your note is: \n[%s]' % (data))
    else:
        return("Your note is empty")
```

  Verify username and password. If it's corrected, return note.

E. Testing

We perform manual testing to verify expected behaviours:

1. During graceful degradation, the replication set must block all WRITE traffic.

   Passed. We killed primary server with Ctrl + C and then immediately performed setNote. No response indicated that the request failed.

2. During graceful degradation, the replication set should allow READ traffic through secondary servers.

   Passed. We killed primary server with Ctrl + C and then immediately performed getNote. The query result is still returned from secondaries.

3. A rollback reverts write operations on a former primary when the member rejoins its replica set after a failover. A rollback is necessary only if the primary had accepted write operations that the secondaries had not successfully replicated before the primary stepped down. Rollbacks are rare. Still we have to test if it's performing correctly.

   Not reproducible. Replication happens really fast that write data from primary are replicated faster than we can shutdown the primary server manually.

4. Secondary server should become primary server in less than 10 seconds after primary server dies.

   Passed. Took around 3 seconds for secondary server to promote itself.

5. Note must be between 1 and 500 characters.

Passed. Tested payload is 1000 chars. Server rejected.

6. Username must be between 1 and 15 characters, contains specified chars.

   Passed. Payloads are input that are above 15 chars and contain characters other than alphanumeric chars and underscore. Server rejected all.

7. NoSQL injection prevention.

   *Passed. All **$** input is escaped.*

8. Hashing.

   Passed. All passwords generated from scripts and inserted are stored as hash with unique salt in the database.