Aidan Kealey - 20151256
Kailey Fejer - 20182434
Paulina Flores - 20152096

Design Document – ELEC 477– Lab 3

1.0 - Main Problem

The task for this assignment was to integrate the Primary Backup Protocol to the Key Value Service used in previous assignments. In this assignment, we used the terms backup and replica interchangeably. This provided a setup where you could have a single fixed server for a given data item and have all write/update operations forwarded to a Primary Server. From there the Primary Server sends writes to other servers and waits for acknowledgements, providing security replicas have the most up to date information. Additionally, this also provides the system the ability to block write/update operations until all copies are updated, further preventing any inconsistencies.

2.0 - Design Solution

2.1 – Adding KVServerStub.cpp

For our design solution, we implemented KVServerStub.cpp, which is a stub class used to differentiate between a backup and a Primary Server. This class allows the Primary Server to communicate with the backup server, and handles operations such as initialization, key-value put operations, and shutdown functionalities.

The constructor, shown in Figure 1 below, initializes the KVServerStub object with the provided server name, backup IP, and port. It sets the initial state as not ready and initializes the serial number.

```
KVServerStub::KVServerStub(string name, string backupIP, in_port_t backupPort){
    //cout << "Entering KVServerStub Constructor" << endl;
    cout << "KVServerStub: Creating KVServerStub for backup: " << backupIP << " on port " << backupPort << endl;
    this->name = name;
    this->backupIP = backupIP;
    this->backupPort = backupPort;
    this->ready = false;
    this->serial = 1;
    cout << "KVServerStub: **BACKUP copy complete** - exiting KVServerStub Constructor" << endl;
}
```

Figure 1: KVServerStub Constructor

Then, the init() function configures the server address, creates a UDP socket, sets a timeout for receiving data, and marks the server as ready upon successful setup.

For kvPut(), if the server is not ready, it initializes it. Then, it constructs a request message containing the key-value pair and sends it to the backup server. It waits for a response and ensures correct message parsing and matching of magic numbers, versions, and serial numbers. Upon receiving a response, it extracts the put result and returns success or failure accordingly.

Overall, this design solution shows communication with the backup server, while ensuring proper initialization, handling of put requests, and shutdown.

Aidan Kealey - 20151256
Kailey Fejer - 20182434
Paulina Flores - 20152096

2.2 – Changes to KVServer.cpp

To integrate the Primary Backup Protocol to the Key Value Service, changes had to be made to the KVServer code. These changes included adding the *setIsPrimary*, *addBackupName*, *addBackupPort*, and *setPrimaryServer* methods, which are responsible for setting the primary status, backup names, ports, and setting the Primary Server's name. These can be seen in Figure 2 below. In the *addBackupName* and *addBackupPort* methods, the actions are only executed if the server is identified as the Primary Server. Similarly, in *setPrimaryServer*, the actions are only executed if the server is not primary. This logic ensures the servers are doing the correct operations based on their role (either primary or backup).

```
void KVServer::addBackupName(string address) {
    // These methods only make sense for the primary server
    if (kvService->isPrimary) {
        kvService->addBackupName(address);
    }
}

void KVServer::addBackupPort(in_port_t port) {
    // These methods only make sense for the primary server
    if (kvService->isPrimary) {
        kvService->addBackupPort(port);
    }
}


void KVServer::setPrimaryServer(string name) {
    // This method is for backup servers to set their primary server's name
    if (!kvService->isPrimary) {
        kvService->setPrimaryServer(name);
    }
}
```

Figure 2: addBackupName, addBackupPort and setPrimaryServer methods in KVServer.

2.3 – Changes to KVService.cpp

Similarly to KVService additional methods were used, to integrate the Primary Backup Protocol to the Key Value Service. The methods can be seen in Figure 3 below, and are as follows:

- *addBackupName*: This method adds a backup server's IP address to the list of backup IPs and is only allowed for the Primary Server.
- *addBackupPort*: This method adds a backup server's port to the list of backup ports and is only allowed for the Primary Server.
- *setPrimaryServer*: This method sets the name of the Primary Server for backup servers. It is used to ensure that the backup servers can identify the Primary Server.

Like KVServer, this logic ensures the servers are doing the correct operations based on their role (either primary or backup).

Aidan Kealey - 20151256
Kailey Fejer - 20182434
Paulina Flores - 20152096

```
bool KVServiceServer::addBackupName(string address){
    if(isPrimary){
        backupIPs.push_back(address);
        return true;
    }
    return false;
}

bool KVServiceServer::addBackupPort(in_port_t port){
    if(isPrimary){
        backupPorts.push_back(port);
        return true;
    }
    return false;
}
void KVServiceServer::setPrimaryServer(const string& name) {
    this->primaryName = name;
}
```

Figure 3: addBackupName, addBackupPort and setPrimaryServer methods in KVService.

Additionally, changes were made within the start method. Firstly, the *isPrimary* variable is introduced, to distinguish between the primary and backup servers. Now, the registration logic related to the service directory server is dependent on the *isPrimary* flag, meaning that only the Primary Server will register with the service directory server.

The changes to start also introduce logic to manage backup servers by storing their IP addresses and ports. This information is used to create instances of KVServerStub to allow for communication with backups. When a kvPut request is executed on the primary, it forwards this request to the backup. The relevant sections of code are shown in Figure 4 and Figure 5 below.

```
if(isPrimary){
    //cout << "isPrimary value is: " << isPrimary << endl;
    bool success = registerService();

if (!backupIPs.empty() && !backupPorts.empty()) {
cout << "KVServiceServer: **BACKUP Request**" << "- Backup Name: " << backupIPs[0] << " @Port: " << backupPorts[0] << " for primary server: " << name << endl;
KVServerStub kvserverstub = KVServerStub(name, backupIPs[0], backupPorts[0]);
//cout << "KVServerStub created for backup: " << backupIPs[0] << endl;
backupStubs.push_back(kvserverstub);
//cout << "the value in backupStubs for backupIPs is" << backupIPs[0] << endl;
}

}
```

Figure 4: Logic for managing Backup Servers in KVService (1).

Aidan Kealey - 20151256
Kailey Fejer - 20182434
Paulina Flores - 20152096

```
if(isPrimary && replyMsg.has_putres() && replyMsg.putres().status()){
        //Forward the message to  backup
        cout << "[Primary Server] Forwarding PUT request to " << backupStubs.size() << " servers." << endl;
        //cout << "KV Primary Server: Forwarding put request to all backup servers" << endl;
        //cout << "Forwarding put request to all backup servers. Total backups: " << backupStubs.size() << endl;


        const E477KV::putRequest &preq = receivedMsg.putargs();
        for(int i = 0; i < backupStubs.size(); i++){
            cout << "[Primary Server] Forwarding to Backup Server"
                    << " (Name: " << backupIPs[i]
                    << ", Port: " << backupPorts[i] << ")." << endl;

            bool result = backupStubs[i].kvPut((int32_t)preq.key(), (const uint8_t *)preq.value().c_str(), (uint16_t)preq.value().length());
            cout << "[Primary Server] Forwarding result for Backup Server"
                << ": " << (result ? "Success" : "Failure") << endl;

            cout << "----------------------------------------" << endl;
            cout << "{KV Primary Server}" << endl;
            cout << "{KV Primary Server} Result from forwarding put request to " + backupIPs[i] << + " is " << result << endl;
        }
        cout << "{KV Primary Server} Finished forwarding put requests" << endl;
        cout << "----------------------------------------" << endl;
        cout << "[Primary Server] Completed forwarding PUT requests to all backups." << endl;
    }
```

Figure 5: Logic for managing backup servers in KVService (2).

2.0 – Testing and Main

This code was tested using 1 test case, implemented in the main code, and is detailed in the accompanying testing document.