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

Design Document – ELEC 477– Lab 1

Main Problem

For this assignment we were tasked with creating a simple Remote Procedure Call (RPC) which would implement a Key Value Service between Servers and Clients using the User Datagram Protocol (UDP). To run on our own machines, we installed Ubuntu on WSL along with the C++ compiler (VS Code), make, GDBM (Linux) / NDBM (Mac), and protobuf-compiler version 3.6.1 or later.
Despite the seemingly straightforward nature of the task – the primary challenge we encountered was the complexity in establishing a remote procedure call and implementing a key-value service. Figure 1 displays an outline providing an overview of the procedure.

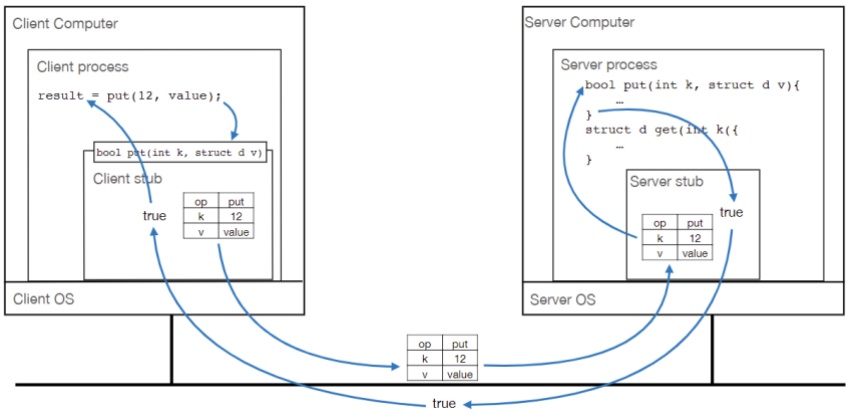

Figure 1: Remote Procedure Call Diagram

Design Solution

Protobuf File

To help marshal the data between client and server, a Protocol Buffer was used to provide a language-neutral, platform-neutral, and flexible mechanism for serializing structured data to communicate between systems. For this part, the most important component was the message header. In simple terms, this confirmed the message was coming from within the targeted system (magic number), the current version of the communication system, and the option of one out of four different types of messages that can be sent within this protocol.

To compile the protobuf file we used the following command:

```
protoc --cpp_out=. rpc.proto
```

This command then created the following files: rpc.pb.cc for the compilation instructions and rpc.pb.hh for the domain instructions. The data types for each of the message parameters were then decided depending on the message function. As an example, kvPutResponse containing a Boolean to classify whether the operation was successful or not. Also, we utilized the “one of” command to distinguish between the 4 different message types.

```
syntax = "proto3";

package RPC;

message rpcHeader {
    int32 magic_number = 1;             // magic number
    int32 version = 2;                  // protocol version
    oneof message {                     // message type
        kvPutRequest put_request = 3;
        kvPutResponse put_response = 4;
        kvGetRequest get_request = 5;
        kvGetResponse get_response = 6;
    }
}
```

Figure 2: Protobuf File Header

This step setup the communication protocol for our system, allowing us to move onto the server code.

The Server

This section introduces the server-side architecture of the RPC system through server.hpp and server.cpp.

Server.hpp

This is the server.hpp code and is shown below in Figure 3. Within this code, the RPCServer class is defined, and inherits functionality from the Node class. This class is the foundation of the server side of the RPC, and is designed to manage the interactions with RPCService, a separate class that will be discussed later. The setDatabaseName method specifies the database that RPCService uses to manage key-value storage. This highlights the difference between the network operations performed by RPCServer while the data handling is performed by the RPCService. This design decision allows the system to be more adaptable in its implementation.

```
#ifndef __SERVER_HPP__
#define __SERVER_HPP__

#include "network.hpp"
#include "service.hpp"
#include <string>
#include <gdbm.h>

using namespace std;

class RPCServer : public Node {
    private:
        shared_ptr<RPCService> theService;

    public:
        RPCServer(string nodeName);
        ~RPCServer() {};

        void setDatabaseName(const string& name);
```

Figure 3: Server.hpp code

Server.cpp

This section will describe the server.cpp code, which can be seen below in Figure 4. This code complements the previously described server.hpp header, by defining the behavior of the RPCServer

class. The server.cpp file details how the RPCServer class utilizes nodeName as a unique identifier and as a part of the database filename, which is an important part of its data management strategy.

The constructor instantiates the RPC service with a shared pointer. It also passes nodeName and a weak pointer to itself. This implementation establishes the connection between the server and the service and manages network connections and data storage from the server's perspective.

```
1   #include "server.hpp"
2
3   RPCServer::RPCServer(string nodeName) : Node(nodeName) {
4       cout << "Main: Server " << nodeName << " adding rpc service" << endl;
5       theService = make_shared<RPCService>(nodeName, weak_from_this());
6
7       setDatabaseName(nodeName+".gdbm");
8       addService(theService);
9   }
10
11  void RPCServer::setDatabaseName(const string& name) {
12      theService->setDatabaseName(name);
13  }
14
```

Figure 4: Server.cpp code

The Service

The Service code is implemented by service.hpp and service.cpp. Together, these files form the foundation of the communication service between client and server, as well as the server-side RPC system. This design decision allowed more control from the service to format servers uniformly and simplify complexity.

Service.hpp

The service.hpp file defines the RPCService class and inherits from Service to implement the core functionality such as establishing the UDP socket connection, handling incoming RPC Put and Get requests, and interacting with the GDBM database for storage.

The key features of this code include initializing and closing the database connection, dispatching incoming messages based on their type, and conducting the actual Put and Get operations with data validation and storage commands.

```
20  class RPCService : public Service {
21      int sockfd;
22      in_port_t PORT = 8080;
23      static const int MAXMSG = 1400;
24      uint8_t udpMessage[MAXMSG];
25      uint8_t udpReply[MAXMSG];
26      struct sockaddr_in cliaddr;
27
28      private:
29          string databaseName;
30          GDBM_FILE database;
31          //DBM* ndbmDatabase;
32
33      public:
34          RPCService();
35          RPCService(string nodeName, weak_ptr<Node> p):Service(nodeName + ".a1_service",p) { }
36          ~RPCService() {
37              stop();
38          }
39
40          void start();
41          void stop();
42
43          void setDatabaseName(const string& name);
44
45          bool initDatabase();
46          void closeDatabase();
47
48          void dispatch(const uint8_t* udpMessage, size_t messageLength, const sockaddr_in& clientAddress, socklen_t clientAddressLength);
49          bool Put(int32_t key, const uint8_t * value, uint16_t value_len);
50          RPC::kvGetResponse Get(int32_t key);
51  };
52
53  #endif
54
```

Figure 5: Service.hpp code

Service.cpp

The service.cpp code is shown below in Figure 6 and Figure 7, and implements the methods declared in service.hpp, bringing functionality to the RPC service. It includes the setup of a UDP socket for communication, database initialization with GDBM, and the dispatch method. The dispatch method parses incoming messages and directs them to the appropriate Put or Get methods.

There were multiple design decisions made within the dispatch method. Firstly, the team decided to serialize and un-serialize the RPC request and response data as byte arrays. This decision was made because arrays can be sent and received directly over the network without the need for complicated transformations. It also ensured that the client received compact and parse-able responses. The team also decided to include error handling and validation for incoming messages, including magic number and version checks. This was a design decision made by the team to ensure that only properly formatted requests are processed, making the service more reliable. Additionally, by checking magic numbers and versions, the service can identify and reject incompatible messages, preventing errors.

There are several other critical operations in this class, such as validation, and response generation. This showcases the server's ability to handle RPC requests efficiently. Overall, this file demonstrates how network messages are processed, data is stored or retrieved from the database, and appropriate responses are sent back to clients.

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

```cpp
#include "service.hpp"
#include "rpc.pb.h"
#include <gdbm.h>
#include <netinet/in.h>  // For the AF_INET and sockaddr_in
#include <cstring>       // For memset
#include <iostream>      // For cerr
#include <sys/socket.h>  // For socket functions
#include <arpa/inet.h>   // For inet_ntop
#include <iomanip>       // For setw, setfill

#ifdef __APPLE__
#define MSG_CONFIRM 0
#endif

using namespace std;
using namespace string_literals;

#define close mclose
void mclose(int fd);

void RPCService::start() {
    cerr << "in RPCService::start" << endl;

    struct sockaddr_in servaddr, cliaddr;

    // Create a socket to receive messages
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Clear structure memory and set server address information
    memset(&servaddr, 0, sizeof(servaddr));
    memset(&cliaddr, 0, sizeof(cliaddr));

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(PORT);

    // Bind the socket with the server address
    if (bind(sockfd, (const struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    if (!initDatabase()) {
        cerr << "Database initialization failed." << endl;
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    while (alive) {
        cerr << "waiting for call from client" << endl;

        socklen_t len = sizeof(cliaddr);
        int n = recvfrom(sockfd, udpMessage, MAXMSG, MSG_WAITALL, (struct sockaddr *)&cliaddr, &len);

        if (n >= 0) {
            cerr << "I am the server. I received a message from the client at address " << inet_ntoa(cliaddr.sin_addr) << " and port " << ntohs(cliaddr.sin_port) << ".\n";
        } else {
            perror("recvfrom error");
        }

        cerr << "server received " << n << " bytes." << endl;

        cerr << "Received bytes: ";
        for (int i = 0; i < min(n, 4); ++i) {
            cerr << hex << (0xFF & static_cast<int>(udpMessage[i])) << " ";
        }
        cerr << dec << "\n";

        char clientStrBuffer[INET_ADDRSTRLEN];
        const char* clientstr = inet_ntop(AF_INET, &(cliaddr.sin_addr), clientStrBuffer, INET_ADDRSTRLEN);
        if (clientstr != nullptr) {
            cerr << "from address " << clientstr << endl;
        } else {
            perror("inet_ntop error");
```

Figure 6: Service.cpp code part 1.

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

```
            perror("inet_ntop error");
        }

        dispatch(udpMessage, n, cliaddr, len);
    }

    closeDatabase();
    close(sockfd);
}



void RPCService::stop() {
    cerr << "in RPCService::stop" << endl;
    alive = false;
};


void RPCService::setDatabaseName(const string& name) {
    this->databaseName = name;
};


bool RPCService::initDatabase() {
    cerr << "in RPCService::initDatabase" << endl;
    database = gdbm_open(databaseName.c_str(), 0, GDBM_WRCREAT, 0666, nullptr);
    if (!database) {
        cerr << "Failed to open database: " << databaseName << endl;
        return false;
    }
    return true;
}


void RPCService::closeDatabase() {
    cerr << "in RPCService::closeDatabase" << endl;
    if (database) {
        gdbm_close(database);
        database = nullptr;
    }
}


void RPCService::dispatch(const uint8_t* udpMessage, size_t messageLength, const sockaddr_in& clientAddress, socklen_t clientAddressLength) {
    cerr << "in RPCService::dispatch" << endl;

    RPC::rpcHeader message;
    if (!message.ParseFromArray(udpMessage, messageLength)) {
        cerr << "Failed to parse incoming message." << endl;
        return;
    }

    cerr << "Extracted magic number: " << message.magic_number() << ", Expected: 1" << endl;

    if(message.magic_number() != 1) {
        cerr << "magic number incorrect" << endl;
        return;
    }

    cerr << "Extracted version: " << message.version() << ", Expected: 2" << endl;

    if(message.version() != 2) {
        cerr << "wrong version" << endl;
        return;
    }

    switch (message.message_case()) {
            // cerr << "i am here in the switch" << endl;
        case RPC::rpcHeader::kPutRequest: {
                cerr << "i am here in kPutRequest" << endl;
            const RPC::kvPutRequest& putRequest = message.put_request();
            const string& value = putRequest.value();
            bool status = Put(putRequest.key(), reinterpret_cast<const uint8_t*>(value.data()), value.length());

            RPC::kvPutResponse putResponse;
```

Figure 7: Service.cpp code part 2.

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

```
    switch (message.message_case()) {
            // cerr << "i am here in the switch" << endl;
        case RPC::rpcHeader::kPutRequest: {
                cerr << "i am here in kPutRequest" << endl;
            const RPC::kvPutRequest& putRequest = message.put_request();
            const string& value = putRequest.value();
            bool status = Put(putRequest.key(), reinterpret_cast<const uint8_t*>(value.data()), value.length());

            RPC::kvPutResponse putResponse;
            putResponse.set_status(status);
            size_t respSize = putResponse.ByteSizeLong();
            if (!putResponse.SerializeToArray(udpReply, MAXMSG)) {
                cerr << "Serialization failed for putResponse" << endl;
                return;
            }

            cerr << "I am the server. I am sending a response to the client at address " << inet_ntoa(clientAddress.sin_addr) << " and port " << ntohs(clientAddress.sin_port) << ".\n";

            // Send response
            int servern = sendto(sockfd, udpReply, respSize, MSG_CONFIRM, reinterpret_cast<const struct sockaddr*>(&clientAddress), clientAddressLength);
            if (servern < 0) {
                perror("sendto failed");
            } else {
                cerr << "Server sent " << servern << " bytes in response to kvPutRequest" << endl;
                    cerr << "Server sent " << servern << " bytes in response to kvPutRequest. Data: ";
                for (int i = 0; i < servern; ++i) {
                    cerr << hex << setw(2) << setfill('0') << (0xff & static_cast<int>(udpReply[i])) << " ";
                }
                cerr << dec << endl;
            }

            break;
        }

        case RPC::rpcHeader::kGetRequest: {
            const RPC::kvGetRequest& getRequest = message.get_request();
            RPC::kvGetResponse getResponse = Get(getRequest.key());

            RPC::rpcHeader responseHeader;
            *responseHeader.mutable_get_response() = getResponse;

            size_t respSize = responseHeader.ByteSizeLong();
            if (respSize > MAXMSG) {
                cerr << "Serialized message size exceeds MAXMSG limit." << endl;
                return;
            }
            uint8_t serializedResponse[respSize];

            if (!responseHeader.SerializeToArray(serializedResponse, respSize)) {
                cerr << "Serialization failed for getResponse" << endl;
                return;
            }

            int servern = sendto(sockfd, serializedResponse, respSize, 0,
                                 reinterpret_cast<const struct sockaddr*>(&clientAddress), clientAddressLength);
            if (servern < 0) {
                perror("sendto failed");
            } else {
                cerr << "Server sent " << servern << " bytes in response to kvGetRequest" << endl;
            }
            break;
        }


        default: {
            cerr << "Unknown message type" << endl;
            break;
        }
    }
}
```

Figure 8: Service.cpp code part 3.

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

```cpp
bool RPCService::Put(int32_t key, const uint8_t *value, uint16_t value_len) {
    std::cerr << "in RPCService::Put" << endl;
    if (!database) {
        cerr << "Database not open, cannot run put. Returning false." << endl;
        return false;
    }

    std::cerr << "i am here" << endl;

    datum databaseKey, databaseValue;
    databaseKey.dptr = reinterpret_cast<char*>(&key);
    databaseKey.dsize = sizeof(int32_t);
    databaseValue.dptr = reinterpret_cast<char*>(const_cast<uint8_t*>(value));
    databaseValue.dsize = value_len;

    std::cerr << "i am here 2" << endl;

    int storeResult = gdbm_store(database, databaseKey, databaseValue, GDBM_REPLACE);
    std::cerr << "i am here 3" << endl;
    if (storeResult != 0) {
        std::cerr << "Failed to store value in the database. GDBM store result: " << storeResult << ". Returning false." << endl;
        return false;
    }
    std::cerr << "Value successfully stored in the database." << endl;
    return true;
}


RPC::kvGetResponse RPCService::Get(int32_t key) {
    cerr << "in RPCService::Get" << endl;
    RPC::kvGetResponse response;
    response.set_status(false);

    if (!database) {
        cerr << "Database not open, cannot run get. Returning response with status false." << endl;
        return response;
    }

    datum databaseKey;
    databaseKey.dptr = reinterpret_cast<char*>(&key);
    databaseKey.dsize = sizeof(key);

    datum result = gdbm_fetch(database, databaseKey);
    if (result.dptr != nullptr) {

        response.set_status(true);
        response.set_value(std::string(result.dptr, result.dsize));
        free(result.dptr);
    }

    return response;
}
```

Figure 9: Service.cpp code part 4.

The Clients

The clients inherit from the Node class (like the server) and contains the instance of the ClientStub to make the RPC call. This includes its own start function to start client activity and methods to set the server address. The start function within our implementation also provides a sample activity to make sure everything is working as intended.

The Client Stub then contains the bulk of the network communication. This includes initializing and shutting down the network connection as well as the kvPut and kvGet instantiations. Key things to note for these methods are the different debugging steps used to make sure all components were working properly.

As an example, for initializing the network connection, timeouts were used to filter out messages that were left hanging and erase any orphan calls. Then for both the kvPut and kvGet instantiations, several terminal messages were sent out to confirm where a message was being sent to and from who it was from. Examples of these are provided below.

```
129     cout << "ClientStub::kvGet - Sending to " << serverAddress << ":" << PORT << endl;
130
131     struct sockaddr_in servaddr;
132     memset(&servaddr, 0, sizeof(servaddr));
133     servaddr.sin_family = AF_INET;
134     servaddr.sin_port = htons(PORT);
135     inet_pton(AF_INET, serverAddress.c_str(), &servaddr.sin_addr);
136
137     RPC::rpcHeader header;
138     header.set_magic_number(1);
139     header.set_version(2);
140     auto* getRequest = header.mutable_get_request();
141     getRequest->set_key(key);
142
143     size_t messageSize = header.ByteSizeLong();
144     uint8_t sendmessageBuffer[messageSize];
145
146     if (!header.SerializeToArray(sendmessageBuffer, messageSize)) {
147         cerr << "Failed to serialize kvGet request." << endl;
148         return {false, ""};
149     }
150
151     cout << "ClientStub::kvGet - Serialized message bytes: ";
152     for (size_t i = 0; i < messageSize; ++i) {
153         cout << hex << (0xFF & static_cast<int>(sendmessageBuffer[i])) << " ";
154     }
155     cout << dec << endl;
156
157     cout << "ClientStub::kvGet - Serialized message size: " << messageSize << " bytes" << endl;
```

Figure 10: kvPut and kvGet code

To check and debug serialization and marshalling, serialized message bytes and size were displayed, and upon deserialization, they were retrieved from the response header with the get_reponse() function. In this manner, the Client class system served to easily create different client stubs for our test cases, and correctly send and receive messages, reporting its activity throughout.

Main

The overall purpose of the main file, regardless of the test case, was to control the parameters of the simulation and start the thread for each simulated computer in the program. Several iterations of the main code were used to test different components for this assignment. Key functions of this code include setting the mutual exclusion, database name and server address.

To set mutual exclusion, lock guards and wait timers allow the clients enough time to setup while using the shared pointer for the Node. Additionally, we use maps to further manage the client and server's access to the shared pointer.

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

```cpp
// Start the server
auto server = std::make_shared<RPCServer>("server");
dynamic_cast<RPCServer*>(server.get())->setDatabaseName("server_db.gdbm");
server->setAddress("10.0.0.2"); // Set server address to 10.0.0.2
server->startServices();

// Wait for servers to get up and running...
std::this_thread::sleep_for(std::chrono::milliseconds(50));

std::cout << "Main: **********" << std::endl;
std::cout << "Main: initializing client" << std::endl;

auto client = std::make_shared<RPCClient>("client");
dynamic_cast<RPCClient*>(client.get())->setServerAddress("10.0.0.3");

std::shared_ptr<std::thread> clientThread;
{
    std::lock_guard<std::mutex> guard(nodes_mutex);
    clientThread = std::make_shared<std::thread>([client]() {
        try {
            client->start();
        } catch (const std::exception& e) {
            std::cerr << "Client encountered an exception: " << e.what() << std::endl;
        }
    });
    nodes.insert(std::make_pair(clientThread->get_id(), client));
    names.insert(std::make_pair(clientThread->get_id(), "client"));
}

std::cout << "Main: **********" << std::endl;
std::cout << "Main: waiting for client to finish" << std::endl;
clientThread->join();
```

Figure 11: main.cpp.

The last component is waiting for all the threads to finish. The ShutdownProtobufLibrary() was used along with stopServices and waitForServices to make sure all threads were completed before the program wrapped up.

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

Testing Document – ELEC 477 – Lab 1

Test Case 1: Putting a Value

For the first test, we had a simple client put a value, retrieve it and check it got the same value back. This helped test the put and get functions were working correctly and is shown in our main.cpp file.

```
int32_t key = 1;
string value = "Hello, RPC World!";

try {
    bool putSuccess = stub.kvPut(key, value);
    if (putSuccess) {
        cout << "Successfully put value: " << value << " at key: " << key << endl;
    } else {
        cerr << "Failed to put value at key: " << key << endl;
    }

    auto getResult = stub.kvGet(key);
    if (getResult.first) {
        cout << "Successfully got value: " << getResult.second << " for key: " << key << endl;
    } else {
        cerr << "Failed to get value for key: " << key << endl;
    }
} catch (const exception& e) {
    cerr << "Exception encountered during RPC operations: " << e.what() << std::endl;
}
```

File testcase1.txt.

Test Case 2: Data with Null Bytes

The next test saw whether null bytes were retrieved correctly, this was to check the case where the Boolean would return false for the get function. There were errors in implementation, but the test case code is as follows:

```
173  void testCase2() {
174      cout << "Running case 2: data with Null bytes" << endl;
175
176      int32_t key = 456;
177      uint8_t value[] = {0x00, 0x00, 0x00};
178      uint16_t value_len = sizeof(value);
179      rpcService.Put(key, value, value_len);
180
181      RPC::kvGetResponse response = rpcService.Get(key);
182
183      if (response.status() && response.value() == 456 && response.value_len() == "3") {
184          cout << "case 2 - Passed" << endl;
185      } else {
186          cout << "case 2 - Failed" << endl;
187      }
188
189      cout << "done case 2" << endl;
190  }
```

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

Test Case 3: Independent 2 Servers and 2 Clients

Then, to show independence and the ability to run parallel programs within the same distributed system, we tested a set of severs and clients. There were errors in implementation, but the test case code is as follows:

```
192  void testCase3() {
193      cout << "Running cse 3: 2 servers and 2 clients" << endl;
194
195      // Set up two servers
196      shared_ptr<Node> parentNode1 = make_shared<Node>("parent_node1");
197      string nodeName1 = "service1";
198      RPCService rpcService1(nodeName1, parentNode1);
199
200      shared_ptr<Node> parentNode2 = make_shared<Node>("parent_node2");
201      string nodeName2 = "service2";
202      RPCService rpcService2(nodeName2, parentNode2);
203
204      rpcService1.start();
205      rpcService2.start();
206
207      // Set up two clients
208      Client client1(8080, "10.0.0.2");
209      Client client2(8080, "10.0.0.3");
210
211      client1.start();
212      client2.start();
213
214      rpcService1.stop();
215      rpcService2.stop();
216      client1.stop();
217      client2.stop();
218
219      cout << "done case 3" << endl;
220  }
```

Test Case 4: Timed Key Overwrite

We also wanted to test what would happen if we wrote different values to the same server under the same key. This also tested what would happen when packages are delayed. There were errors in implementation, but the test case code is as follows:

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

```
void testCase4() {
    cout << "Running case 4: key overwrite" << endl;

    shared_ptr<Node> parentNode = make_shared<Node>("parent_node");
    string nodeName = "service";
    RPCService rpcService(nodeName, parentNode);

    rpcService.start();

    int32_t key = 789;
    uint8_t value[] = {0x01, 0x02, 0x03};
    uint16_t value_len = sizeof(value);

    rpcService.Put(key, value, value_len);

    RPC::kvGetResponse response = rpcService.Get(key);

    if (response.status() && response.value() == 123 && response.value_len() == "3") {
        cout << "test - Passed" << endl;
    } else {
        cout << "test - Failed" << endl;
    }

    std::this_thread::sleep_for(std::chrono::seconds(5));

    int32_t key = 789;
    uint8_t value[] = {0xAA, 0xBB, 0xCC};
    uint16_t value_len = sizeof(value);
    rpcService.Put(key, value, value_len);

    RPC::kvGetResponse response = rpcService.Get(key);

    if (response.status() && response.value() == 789 && response.value_len() == "3") {
        cout << "test - Passed" << endl;
    } else {
        cout << "test - Failed" << endl;
    }

    cout << "done case " << endl;
}
```