

ELEC 477

Kevin Yu: 20203451

Raatik Sharma: 20120770

Ainsley Taylor: 20210012

2024-02-05

For example: "File test1.txt

contains the output of our first test. It shows that the value 'Smith Engineering is great' was stored with the key 1, and successfully retrieved using the key 1". You will need to have appropriate print statement in the client so that the output shows that the test is successful.

Test 1

1. *The first test should have a simple client that puts a value and retrieves it and checks that it got the same value back.*

Test 1 asks us to perform a PUT and GET RPC. In our client.cpp file we called a PUT request with a value of "kevin/Oyu" with a key of 7. Test1.txt shows the output of our results when we do a GET request as shown below. This illustrates that the RPC methods for PUT and GET work correctly since we fetch the correct value that was associated with the key 7.

```
mainTest3.cpp x e1client3.cpp e1main.cpp x
e1main.cpp > main(int, char * D)
26
27 // Setup network configuration using command line args
28 int res = network_init(argc, argv);
29 std::stringstream ss;
30
31 // Start up the server
32 std::cout << "Main: *****" << std::endl;
33 std::cout << "Main: starting server" << std::endl;
34
35 // Initializing the server with its own address
36 shared_ptr<E1Server> e1Server = make_shared<E1Server>("e1server");
37 e1Server->setAddress("10.0.0.2");
38 e1Server->setDbmFile("dean.db");
39 e1Server->startServices();
40
41 // Wait for the servers to setup before initializing the client
42 std::this_thread::sleep_for(std::chrono::milliseconds(50));
43
44 // Initialize the client
45 std::cout << "Main: *****" << std::endl;
46 std::cout << "Main: init client" << std::endl;
47 shared_ptr<E1Client> e1Client = make_shared<E1Client>("e1client");
48 e1Client->setAddress("10.0.0.3");
49 e1Client->setServerAddress("10.0.0.2");
50
51 // Starting client to send messages to the server
52 std::cout << "Main: *****" << std::endl;
53 std::cout << "Main: starting client" << std::endl;
54 shared_ptr<thread> t;
55
56 // Create a guard scope to add client to global variables and call its start method
57 {
58     // need a scope for the lock guard.
59     // if this doesn't work put it in a function
60     std::lock_guard<std::mutex> guard(nodes_mutex);
61
62     // Spin off a new thread and call the start() method on the client
63     t = make_shared<thread>([e1Client]()
64     {
65         try{
66             e1Client -> start();
67         } catch (exitThread & e){ } });
68
69     // Critical section
70     nodes.insert(make_pair(t->get_id(), e1Client));
71     names.insert(make_pair(t->get_id(), "e1client"));
72 }
73
74 // Wait for the threads to finish its execution before proceeding with the main thread
75 std::cout << "Main: *****" << std::endl;
76 std::cout << "Main: waiting for clients to finish" << std::endl;
77 t->join();
78
79 // Stops all services on the servers by setting alive flag to false.
80 std::cout << "Main: *****" << std::endl;
81 std::cout << "Main: calling stop services on server" << std::endl;
82 e1Server->stopServices();
83
84 // Wait for all server threads to complete
85 std::cout << "Main: *****" << std::endl;
86 std::cout << "Main: waiting for threads to complete" << std::endl;
87 e1Server->waitForServices();
88
89 google::protobuf::ShutdownProtobufLibrary();
90 std::cout << "Main: shutting down protobuf library" << std::endl;
91
92 return 0;
93
e1client.cpp x
e1client.cpp >
1 #include "e1client.hpp"
2 #include "data.pb.h"
3 using namespace std;
4 using namespace std::string_literals;
5
6 #define close wClose
7
8 /**
9  * Sets the server address attribute so the client stub can create a socket for the server
10 */
11 void E1Client::setServerAddress(char *serverAddress)
12 {
13     this->serverAddress = serverAddress;
14 }
15
16 /**
17  * Defines the RPC calls that the client will invoke
18 */
19 void E1Client::start()
20 {
21     // Create an instance of the client stub
22     clientStub = make_shared<ClientStub>();
23     clientStub->setServerAddress(serverAddress);
24     // Call the put method from the client stub
25     Data::put_response dataPut = clientStub->put(13, reinterpret_cast<const uint_t *>("kevin@Oyu"), 10);
26     std::cout << "E1CLIENT's PUT status: " << dataPut.success() << std::endl;
27
28     // Call the get method from the client stub
29     Data::iget_response dataGet = clientStub->iget(13);
30
31     // Print out the results from the get request
32     std::cout << "Client's GET status: " << dataGet.status() << std::endl;
33     std::cout << "Client's GET value: " << dataGet.value() << std::endl;
34     std::cout << "Client's GET length: " << dataGet.value_length() << std::endl;
35 }
36
```

```
----- E1SERVICE: Get Request Finished -----  
Client's GET status: 1  
Client's GET value: kevinyu  
Client's GET length: 8  
Main: *****  
Main: calling stop services on server  
Main: *****  
Main: waiting for threads to complete  
Main: shutting down protobuf library
```

Test 2

2. You must test that data containing null bytes is stored and retrieved correctly.

Test 2 asks us to see if our RPC works with storing and returning null bytes. In this test we do the same PUT and GET request as in Test 1. But we illustrate that storing and retrieving null bytes works correctly since “kevin\0yu” contains a null byte in the middle indicating that the end of the string is after the name “kevin” and “yu” would not be stored theoretically. However, we see from the image below that our GET request does indeed return back “kevinyu” meaning our RPC method can store and retrieve null bytes.

```

mainTest3.cpp | eClient.cpp | main.cpp
main.cpp > main(int, char* D)
26
27 // Setup network configuration using command line args
28 int res = network_init(argc, argv);
29 std::stringstream ss;
30
31 // Start up the server
32 std::cout << "Main: *****" << std::endl;
33 std::cout << "Main: starting server" << std::endl;
34
35 // Initializing the server with its own address
36 shared_ptr<IServer> eIServer = make_shared<IServer>("eIServer");
37 eIServer->setAddress("10.0.0.2");
38 eIServer->setdbFile("dean.db");
39 eIServer->startServices();
40
41 // Wait for the servers to setup before initializing the client
42 std::this_thread::sleep_for(std::chrono::milliseconds(50));
43
44 // Initialize the client
45 std::cout << "Main: *****" << std::endl;
46 std::cout << "Main: init client" << std::endl;
47 shared_ptr<IClient> eIClient = make_shared<IClient>("eIClient");
48 eIClient->setAddress("10.0.0.3");
49 eIClient->setServerAddress("10.0.0.2");
50
51 // Starting Client to send messages to the server
52 std::cout << "Main: *****" << std::endl;
53 std::cout << "Main: starting client" << std::endl;
54 shared_ptr<thread> t;
55
56 // Create a guard scope to add Client to global variables and call its start method
57 {
58     // need a scope for the lock guard.
59     // if this doesn't work put it in a function
60     std::lock_guard<std::mutex> guard(nodes_mutex);
61
62     // Spin off a new thread and call the start() method on the client
63     t = make_shared<thread>([eIClient]{});
64     {
65         try{
66             eIClient -> start();
67         } catch (exitThread & e){ } });
68
69     // Critical section
70     nodes.insert(make_pair(t->get_id(), eIClient));
71     names.insert(make_pair(t->get_id(), "eClient"));
72 }
73
74 // Wait for the threads to finish its execution before proceeding with the main thread
75 std::cout << "Main: *****" << std::endl;
76 std::cout << "Main: waiting for clients to finish" << std::endl;
77 t->join();
78
79 // Stops all services on the servers by setting alive flag to false.
80 std::cout << "Main: *****" << std::endl;
81 std::cout << "Main: calling stop services on server" << std::endl;
82 eIServer->stopServices();
83
84 // Wait for all server threads to complete
85 std::cout << "Main: *****" << std::endl;
86 std::cout << "Main: waiting for threads to complete" << std::endl;
87 eIServer->waitforServices();
88
89 google::protobuf::ShutdownProtobufLibrary();
90 std::cout << "Main: shutting down protobuf library" << std::endl;
91
92 return 0;
93
eClient.cpp
mainTest3.cpp | eClient.cpp | main.cpp | eClient.cpp X
eClient.cpp >
1 #include "eClient.hpp"
2 #include "data.pb.h"
3 using namespace std;
4 using namespace std::string_literals;
5
6 #define close eclose
7
8 /*
9  * Sets the server address attribute so the client stub can create a socket for the server
10 */
11 void IClient::setServerAddress(char "serverAddress")
12 {
13     this->serverAddress = serverAddress;
14 }
15
16 /*
17  * Defines the RPC calls that the client will invoke
18 */
19 void IClient::start()
20 {
21     // Create an instance of the client stub
22     ClientStub = make_shared<ClientStub>();
23     ClientStub->setServerAddress(serverAddress);
24     // Call the put method from the client stub
25     Data::put_response dataPut = ClientStub->put(13, reinterpret_cast<const uint8_t*>("kevin@ju"), 10);
26     std::cout << "eCLIENT's PUT status: " << dataPut.success() << std::endl;
27
28     // Call the get method from the client stub
29     Data::get_response dataGet = ClientStub->get(13);
30
31     // Print out the results from the get request
32     std::cout << "Client's GET status: " << dataGet.status() << std::endl;
33     std::cout << "Client's GET value: " << dataGet.value() << std::endl;
34     std::cout << "Client's GET length: " << dataGet.value_length() << std::endl;
35 }
36

```

```

----- E1SERVICE: Get Request Finished -----
Client's GET status: 1
Client's GET value: kevinju
Client's GET length: 8
Main: *****
Main: calling stop services on server
Main: *****
Main: waiting for threads to complete
Main: shutting down protobuf library

```

Test 3

3. You should have tests with two servers and two clients to show independence.

In Test 3, we setup two independent clients and two independent servers each with their unique addresses and each server has its own unique database. We call a PUT request for each client with the exact same key “69” but client 1 has a value of “111” and client 2 has a value of “420”. When we call the

GET request on each client with the same key of “69” we see that each client returns their respective value that they stored with their PUT request. This illustrates that each client and server pair are independent of one another.

```

mainTest3.cpp
// Setup network configuration using command line args
int res = network_init(args, &np);
std::stringstream ss;

// Initializing the server with its own address
std::cout << "Main: *****" << std::endl;
std::cout << "Main: starting server" << std::endl;

// INITIALIZING SERVER 1
shared_ptr<IServer> e1Server = make_shared<IServer>("e1server");
e1Server->setAddress("10.0.0.7");
e1Server->setSocketFile("dcm1.db");
e1Server->startServices();

// INITIALIZING SERVER 2
shared_ptr<IServer> e2Server = make_shared<IServer>("e2server");
e2Server->setAddress("10.0.0.4");
e2Server->setSocketFile("dcm2.db");
e2Server->startServices();

// Wait for the servers to setup before initializing the client
std::this_thread::sleep_for(std::chrono::milliseconds(50));

// Initialize the client 1 with unique address and unique server address
std::cout << "Main: *****" << std::endl;
std::cout << "Main: init client" << std::endl;
shared_ptr<IClient> e1Client = make_shared<IClient>("e1client");
e1Client->setAddress("10.0.0.3");
e1Client->setServerAddress("10.0.0.7");

// Initialize the client 2 with unique address and unique server address
std::cout << "Main: *****" << std::endl;
std::cout << "Main: init client" << std::endl;
shared_ptr<IClient> e2Client = make_shared<IClient>("e2client");
e2Client->setAddress("10.0.0.5");
e2Client->setServerAddress("10.0.0.4");

// Starting Client to send messages to the server
std::cout << "Main: *****" << std::endl;
std::cout << "Main: starting client" << std::endl;
shared_ptr<Thread> t1;
shared_ptr<Thread> t2;

// Create a guard scope to add client1 to global variables and call its start method
{
    Data:put_response dataPut = clientStub->put(69, reinterpret_cast<const uint8_t*>("111"), strlen("111"));
    std::cout << "Client1's PUT status: " << dataPut.success() << std::endl;
    std::cout << "----- CLIENT 1 RESULTS -----" << std::endl;

    // Call the get method from the client stub
    Data:get_response dataGet = clientStub->get(69);

    // Print out the results from the get request
    std::cout << "Client1's GET status: " << dataGet.status() << std::endl;
    std::cout << "Client1's GET value: " << dataGet.value() << std::endl;
    std::cout << "Client1's GET length: " << dataGet.value_length() << std::endl;
}
else if (addr_str == "10.0.0.5")
{
    shared_ptr<ClientStub> clientStub2 = make_shared<ClientStub>();
    clientStub2->setServerAddress(serverAddress);

    std::cout << "INSIDE CLIENT 2: " << std::endl;

    // Call the put method from the client stub
    Data:put_response dataPut = clientStub2->put(69, reinterpret_cast<const uint8_t*>("420"), strlen("420"));
    std::cout << "Client2's PUT status: " << dataPut.success() << std::endl;
    std::cout << "----- CLIENT 2 RESULTS -----" << std::endl;

    // Call the get method from the client stub
    Data:get_response dataGet = clientStub2->get(69);

    // Print out the results from the get request
    std::cout << "Client2's GET status: " << dataGet.status() << std::endl;
    std::cout << "Client2's GET value: " << dataGet.value() << std::endl;
    std::cout << "Client2's GET length: " << dataGet.value_length() << std::endl;
}
// Increase the conditional doesn't work for finding what RPC to call
else
{
    std::cout << "No clients with address" << addr_str << " found " << std::endl;
}
}

```

```

----- CLIENT 2 RESULTS -----
----- E1SERVICE: Put Request Finished -----
CLIENT STUB: Success status of the put request:0
Client1's PUT status: 0
----- CLIENT 1 RESULTS -----
E1SERVICE: server received 12 bytes
E1SERVICE: Client Address 10.0.0.5
E1SERVICE: server received 12 bytes
E1SERVICE: Client Address 10.0.0.3
----- E1SERVICE: Get Request Finished -----
Client2's GET status: 1
Client2's GET value: 420
Client2's GET length: 8
----- E1SERVICE: Get Request Finished -----
Client1's GET status: 1
Client1's GET value: 111
Client1's GET length: 8
Main: *****
Main: calling stop services on server
Main: *****
Main: waiting for threads to complete
Main: shutting down protobuf library

```

Test 4

4. You should have tests that write different values to the same server with the same key to overwrite, this test should be timed so that if one of the packets is delayed, the value returned at the end by one of the clients is different.

In Test 4, we have two clients that write two PUT requests and then call two GET requests. However, the first client’s GET request is delayed meaning that the PUT request from the second client will overwrite the value in the gdbm database. When both clients return their values, we see that the GET responses

print out the second client's stored value. Therefore the second client has overwritten the first client's initial value.

```
mainTest4.cpp > main(int, char * D)
35 // INITIALIZING SERVER 1
36 shared_ptr<IServer> e1Server = make_shared<IServer>("e1server");
37 e1Server->setAddress("10.0.0.2");
38 e1Server->setDbmFile("dean.db");
39 e1Server->startServices();
40
41
42
43 // wait for servers to get up and running...
44 std::this_thread::sleep_for(std::chrono::milliseconds(50));
45
46 // INITIALIZING CLIENT 1
47 std::cout << "Main: *****" << std::endl;
48 std::cout << "Main: init client" << std::endl;
49 shared_ptr<IClient> e1Client = make_shared<IClient>("e1client");
50 e1Client->setAddress("10.0.0.3");
51 e1Client->setServerAddress("10.0.0.2");
52
53 // INITIALIZING CLIENT 2
54 std::cout << "Main: *****" << std::endl;
55 std::cout << "Main: init client" << std::endl;
56 shared_ptr<IClient> e2Client = make_shared<IClient>("e2client");
57 e2Client->setAddress("10.0.0.3");
58 e2Client->setServerAddress("10.0.0.2");
59
60 // Starting Client to send messages to the server
61 std::cout << "Main: *****" << std::endl;
62 std::cout << "Main: starting client" << std::endl;
63 shared_ptr<thread> t1;
64 shared_ptr<thread> t2;
65
66 // Client 1
67 {
68     // need a scope for the lock guard.
69     // If this doesn't work put it in a function
70     std::lock_guard<std::mutex> guard(nodes_mutex);
71
72     t = make_shared<thread>([e1Client]()
73     {
74         try{
75             e1Client -> start();
76         } catch (exitThread & e){ } });
77
78 // Critical section
79 nodes.insert(make_pair(t->get_id(), e1Client));
80 names.insert(make_pair(t->get_id(), "e1client"));
81
82 // Client 2
83 {
84     // need a scope for the lock guard.
85     // If this doesn't work put it in a function
86     std::lock_guard<std::mutex> guard(nodes_mutex);
87
88     t2 = make_shared<thread>([e2Client]()
89     {
90         try{
91             e2Client -> start();
92         } catch (exitThread & e){ } });
93
94 // Critical section?
95 nodes.insert(make_pair(t2->get_id(), e2Client));
96 names.insert(make_pair(t2->get_id(), "e2client"));
97
98 }
99
100 // Terminates the threads?
101 std::cout << "Main: *****" << std::endl;
102 }
```

```
e1client.cpp > main()
1 #include "e1client4.h"
2 #include "data.pb.h"
3 using namespace std;
4 using namespace std::string_literals;
5
6 #define close mclose
7
8 /**
9  * Sets the server address attribute so the client stub can create a socket for the server
10 */
11 void IClient::setServerAddress(char *serverAddress)
12 {
13     this->serverAddress = serverAddress;
14 }
15
16 /**
17  * Defines the RPC calls that the client will invoke
18 */
19 void IClient::start()
20 {
21     // Create an instance of the client stub
22     clientStub = make_shared<ClientStub>();
23     clientStub->setServerAddress(serverAddress);
24
25     // Conditionally call specific RPC methods based on what server the client is trying to call
26     if (addr_str == "10.0.0.3")
27     {
28         // Call the put method from the client stub
29         Data:put_response dataPut = clientStub->put(7, reinterpret_cast<const uint8_t *>("123"), strlen("123"));
30         std::this_thread::sleep_for(std::chrono::milliseconds(3000));
31
32         std::cout << "----- CLIENT 1 RESULTS -----" << std::endl;
33
34         // Call the get method from the client stub
35         Data:get_response dataGet = clientStub->get(7);
36
37         // Print out the results from the get request
38         std::cout << "Client1's GET status: " << dataGet.status() << std::endl;
39         std::cout << "Client1's GET value: " << dataGet.value() << std::endl;
40         std::cout << "Client1's GET length: " << dataGet.value_length() << std::endl;
41     }
42     else if (addr_str == "10.0.0.5")
43     {
44         // Call the put method from the client stub
45         Data:put_response dataPut = clientStub->put(7, reinterpret_cast<const uint8_t *>("586"), strlen("586"));
46         std::this_thread::sleep_for(std::chrono::milliseconds(50));
47
48         std::cout << "----- CLIENT 2 RESULTS -----" << std::endl;
49
50         // Call the get method from the client stub
51         Data:get_response dataGet = clientStub->get(7);
52
53         // Print out the results from the get request
54         std::cout << "Client2's GET status: " << dataGet.status() << std::endl;
55         std::cout << "Client2's GET value: " << dataGet.value() << std::endl;
56         std::cout << "Client2's GET length: " << dataGet.value_length() << std::endl;
57     }
58 }
```

```
----- E1SERVICE: Put Request Finished -----
CLIENT STUB: Success status of the put request:0
----- CLIENT 2 RESULTS -----
E1SERVICE: server received 12 bytes
E1SERVICE: Client Address 10.0.0.5
----- E1SERVICE: Get Request Finished -----
Client2's GET status: 1
Client2's GET value: 586
Client2's GET length: 8
----- CLIENT 1 RESULTS -----
E1SERVICE: server received 12 bytes
E1SERVICE: Client Address 10.0.0.3
----- E1SERVICE: Get Request Finished -----
Client1's GET status: 1
Client1's GET value: 586
Client1's GET length: 8
Main: *****
Main: calling stop services on server
Main: *****
Main: waiting for threads to complete
Main: shutting down protobuf library
```