|  |  |
| --- | --- |
| **Assignment** | |
| **Course Code** | CSE301A |
| **Course Name** | Distributed Systems |
| **Programme** | B.Tech |
| **Department** | CSE |
| **Faculty** | FET |

|  |  |
| --- | --- |
| **Name of the Student** | Satyajit Ghana |
| **Reg. No.** | 17ETCS002159 |
| **Semester/Year** | 06/2020 |
| **Course Leader(s)** | Chaitra S. |



|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Declaration Sheet | | | | | | | | |
| Student Name | Satyajit Ghana | | | | | | | |
| Reg. No | 17ETCS002159 | | | | | | | |
| Programme | B.Tech | | | | | Semester/Year | 06/2020 | |
| Course Code | CSE301A | | | | | | | |
| Course Title | Distributed Systems | | | | | | | |
| Course Date |  | | to |  | | | | |
| Course Leader | Chaitra S. | | | | | | | |
| **Declaration**  The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly. | | | | | | | | |
| Signature of the Student | |  | | | | | Date |  |
| Submission date stamp  (by Examination & Assessment Section) | |  | | | | | | |
| Signature of the Course Leader and date | | | | | Signature of the Reviewer and date | | | |
|  | | | | |  | | | |

# Contents

[Declaration Sheet ii](#_Toc38950943)

[Contents iii](#_Toc38950944)

[List Of Figures iv](#_Toc38950945)

[1 Question 1 5](#_Toc38950946)

[1.1 Design of Banking Application 5](#_Toc38950947)

[1.1.1 gRPC 5](#_Toc38950948)

[1.1.2 MongoDB Transactions 6](#_Toc38950949)

[1.2 Implementation of Banking Application 8](#_Toc38950950)

[1.3 Testing of developed Banking Application 15](#_Toc38950951)

[2 Question 2 20](#_Toc38950952)

[2.1 RPC, the need for it and two sample programs 20](#_Toc38950953)

[2.1.1 Example 1 20](#_Toc38950954)

[2.1.2 Example 2 25](#_Toc38950955)

[2.2 RMI, the need for it and two sample programs 29](#_Toc38950956)

[2.2.1 Example 1 30](#_Toc38950957)

[2.2.2 Example 2 33](#_Toc38950958)

[2.3 Similarities and Differences between RPC and RMI 35](#_Toc38950959)

[Bibliography 37](#_Toc38950960)

[Appendix A 38](#_Toc38950961)

[Appendix B 49](#_Toc38950962)

# List Of Figures

[Figure 1 gRPC C Core and Wrapped Language Stack (Google) 5](#_Toc38950963)

[Figure 2 gRPC Deployment Diagram 6](#_Toc38950964)

[Figure 3 ZrB Deployment Diagram 6](#_Toc38950965)

[Figure 4 Wired Tiger Transactions 8](#_Toc38950966)

[Figure 5 server help option 14](#_Toc38950967)

[Figure 6 client help option 14](#_Toc38950968)

[Figure 7 client logs 15](#_Toc38950969)

[Figure 8 server logs 15](file:///D:\University-Work\University-Work-SEM-06\Assignment\DS\DS.docx#_Toc38950970)

[Figure 9 DB before transaction 16](#_Toc38950971)

[Figure 10 client logs 17](#_Toc38950972)

[Figure 11 server logs 18](file:///D:\University-Work\University-Work-SEM-06\Assignment\DS\DS.docx#_Toc38950973)

[Figure 12 DB after transaction 18](#_Toc38950974)

[Figure 13 DB after transaction 19](#_Toc38950975)

[Figure 14 Node gRPC Server 23](#_Toc38950976)

[Figure 15 Node gRPC client 24](#_Toc38950977)

[Figure 16 Node Marks Server 27](#_Toc38950978)

[Figure 17 Node Marks Client 28](#_Toc38950979)

[Figure 18 Simple Calculator RMI 33](#_Toc38950980)

[Figure 19 RMI Client 35](#_Toc38950981)

[Figure 20 RMI Server 35](#_Toc38950982)

# Question 1

Solution to Part A

## Design of Banking Application

### gRPC

Since we need to have multiple clients to be connected to the server and working at real-time data, also being very high performant at the same time, a banking application must be robust, secure and responsive at the same time, we chose gRPC to be the best solution for this.

gRPC is a high performance, open-source universal RPC framework, developed by Google. In gRPC, a client application *can directly call methods on a server application on a different machine as if it was a local object*, making it easier to create distributed applications and services. Google’s RPC framework and the protocol itself is built on http2. *It has many advantages such as the ability to enable different languages to interact with each other via gRPC calls.*

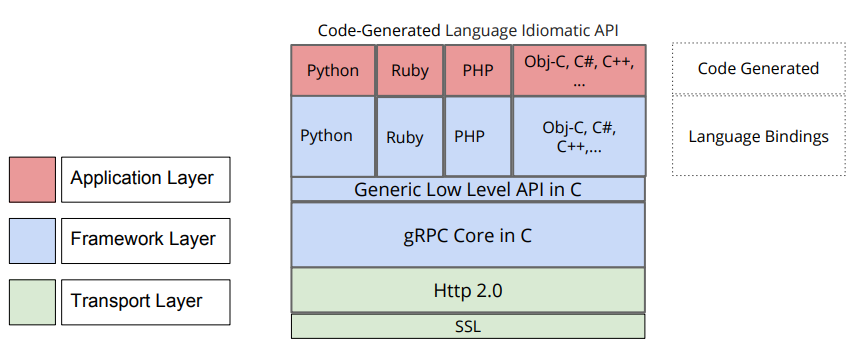


Figure gRPC C Core and Wrapped Language Stack (Google)

gRPC uses Protocol Buffers for serialization/deserialization of objects and data structures sent between clients and servers. Developers need to define their data structures in .proto files and auto generated code provides functions that allow for these structures to be serialized/deserialized.

The gRPC diagram below describes how different clients running of different operating systems and using a different programming language, can connect to the gRPC server and call a remote procedure. This allows for ***heterogeneity*** in the system.

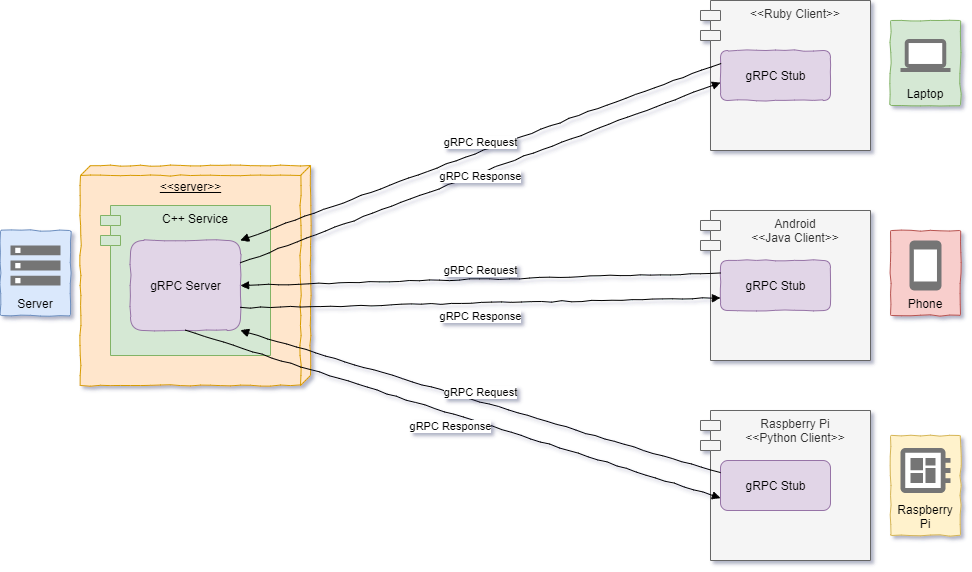


Figure gRPC Deployment Diagram

The Deployment Diagram for our Bank Server and Bank Client is as below,

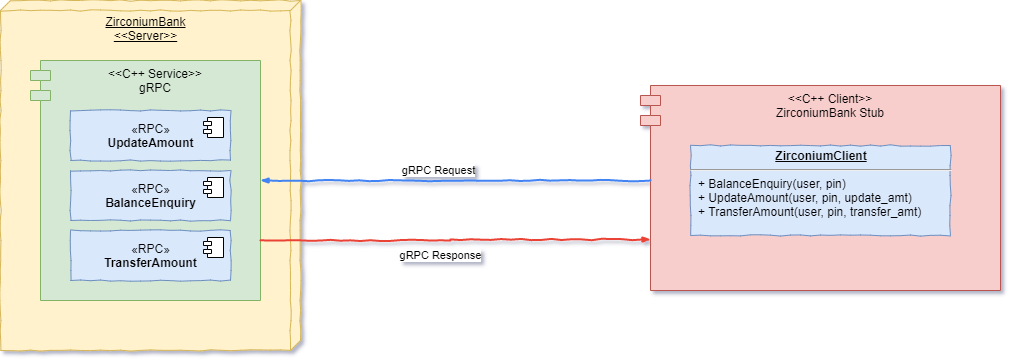


Figure ZrB Deployment Diagram

### MongoDB Transactions

The Database chosen for the application was MongoDB, since it supports *distributed transactions*, and handles then very efficiently, also it follows *ACID properties* which is essential for our application.

MongoDB uses multi-granularity locking that allows operations to lock at the global, database or collection level, and allows for individual storage engines to implement their own concurrency control below the collection level (e.g., at the document-level in WiredTiger).

MongoDB uses reader-writer locks that allow concurrent readers shared access to a resource, such as a database or collection. In addition to a shared (S) locking mode for reads and an exclusive (X) locking mode for write operations, intent shared (IS) and intent exclusive (IX) modes indicate an intent to read or write a resource using a finer granularity lock. When locking at a certain granularity, all higher levels are locked using an intent lock.

For example, when locking a collection for writing (using mode X), both the corresponding database lock and the global lock must be locked in intent exclusive (IX) mode. A single database can simultaneously be locked in IS and IX mode, but an exclusive (X) lock cannot coexist with any other modes, and a shared (S) lock can only coexists with intent shared (IS) locks.

Locks are fair, with reads and writes being queued in order. However, to optimize throughput, when one request is granted, all other compatible requests will be granted at the same time, potentially releasing them before a conflicting request. For example, consider a case in which an X lock was just released, and in which the conflict queue contains the following items:

IS → IS → X → X → S → IS

**(MongoDB Documentation)**

Because a single document can contain related data that would otherwise be modeled across separate parent-child tables in a relational schema, MongoDB’s atomic single-document operations already provide transaction semantics that meet the data integrity needs of the majority of applications. One or more fields may be written in a single operation, including updates to multiple sub-documents and elements of an array. The guarantees provided by MongoDB ensure complete isolation as a document is updated; any errors cause the operation to roll back so that clients receive a consistent view of the document.

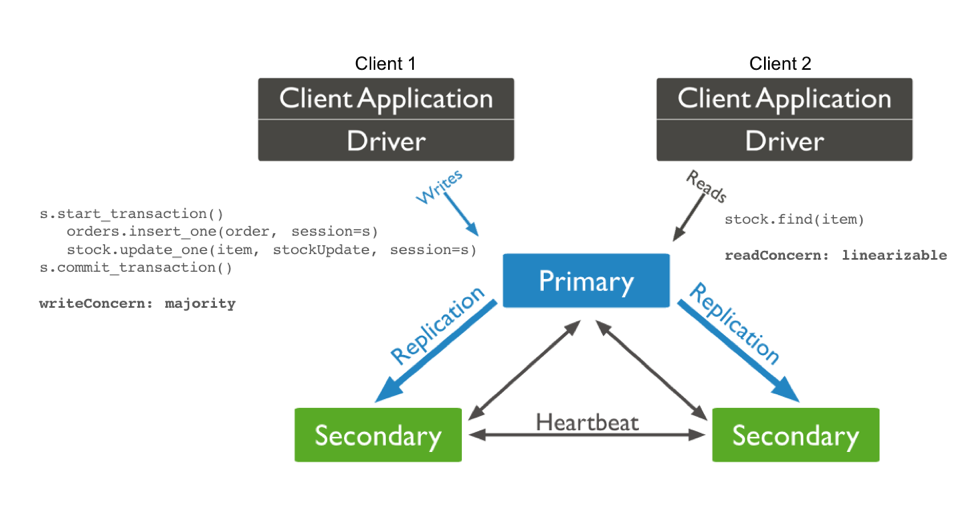


Figure Wired Tiger Transactions

Distributed transactions refer to multi-document transactions on sharded clusters and replica sets. Multi-document transactions (whether on sharded clusters or replica sets) are also known as distributed transactions

## Implementation of Banking Application

The complete code has been attached in Appendix A and Appendix B.

To start with the server/client we first need to define our service and our protocol buffer messages so the server/client know how to *marshal* and *de-marshal* the data.

package zirconium.bank;

**// Interface exported by the server**

**// Here we have 3 remote procedures that can be called**

**// UpdateAmount, BalanceEnquiry and TransferAmount with their respective arguments**

**// and return messages**

service WebService {

rpc UpdateAmount(RequestAmount) returns (Balance) {}

rpc BalanceEnquiry(AuthUser) returns (Balance) {}

rpc TransferAmount(TransferReq) returns (Balance) {}

}

message TransferReq {

AuthUser from\_user = 1;

string to\_user = 2;

int32 amount = 3;

}

message AuthUser {

string username = 1;

string pin = 2;

}

message RequestAmount {

AuthUser user = 1;

int32 value = 2;

}

message Balance {

int32 value = 1;

}

Now that the protocol buffers are defined, we then need to implement our service at the backend, using any of the supported language of gRPC, We’ve chosen C++ to make our backend, since it is very robust and high performant, which is very essential for a Bank Server.

All of the Code has been attached at the Appendix, here we’ll discuss the important aspects of the server.

All of our code is multithreaded, gRPC takes care of it for us, any client that calls a remote procedure is assigned their own threaded procedure call. Our MongoDB also supports multithreading, so we need to create a multi-threaded connection pool, then we can request for a client from this pool and then start working on the data.

This method will run the service and expose a port so that clients can connect to it,

void RunServer(const std::string& server\_addr, const std::string& mongo\_uri) {

    WebServiceImpl service;

    // initialize the service with mongo db name "zirconium"

    service.initiaize(mongo\_uri);

    ServerBuilder builder;

    // Listen on the given address without any authentication mechanism.

    builder.AddListeningPort(server\_addr, grpc::InsecureServerCredentials());

    // Register "service" as the instance through which we'll communicate with

    // clients. In this case it corresponds to an \*synchronous\* service.

    builder.RegisterService(&service);

    // Finally assemble the server.

    std::unique\_ptr<Server> server(builder.BuildAndStart());

    LOG\_S(INFO) << "Server listening on " << server\_addr;

    // Wait for the server to shutdown. Note that some other thread must be

    // responsible for shutting down the server for this call to ever return.

    server->Wait();

}

MongoDB Connection Pool

    void initiaize(const std::string& mongo\_uri) {

        conn\_pool\_ = std::make\_unique<mongocxx::pool>(mongocxx::uri{mongo\_uri});

    };

Let’s look at one of the RPC method TransferAmount to see how it works

The Basic Transaction here is:

1. START TRANSACTION
2. curr\_to\_user.amount = Read(to\_user)
3. curr\_from\_user.amount = Read(from\_user)
4. check if to\_user exists
5. check if from\_user has sufficient balance
6. curr\_from\_user.balance -= amount
7. curr\_to\_user.balance += amount
8. Write(to\_user)
9. Write(from\_user)
10. COMMIT TRANSACTION
11. /\*\*
12. Transfer Amount: Transfer amount from current user to to\_user
13. Tested Works ✅
14. \*/

This is the TransferAmount RPC method, any client can call this with the appropriate arguments

1. Status TransferAmount(ServerContext\* context, const TransferReq\* transfer\_req, Balance\* balance) override {
2. LOG\_S(INFO) << "TransferAmount called by peer: " << context->peer();
3. // get a client from the pool
4. auto client = conn\_pool\_->acquire();
5. // get the "zirconium" db
6. auto db = (\*client)["zirconium"];

Now we bind the data received from the client to local variables

1. // curr\_user
2. const auto auth\_user = transfer\_req->from\_user();
3. const auto to\_user = transfer\_req->to\_user();
4. // guard to check if the to\_user is same as curr\_user
5. if (to\_user == auth\_user.username()) {
6. return Status(StatusCode::FAILED\_PRECONDITION, "to\_user is same as curr\_user");
7. }
8. logAuthData\_(auth\_user.username(), auth\_user.pin());
9. try {
10. auto [is\_authenticated, user] = getAuthData\_(auth\_user.username(), auth\_user.pin(), (\*client));
11. // check if the user is authenticated
12. if (not is\_authenticated) {
13. LOG\_S(WARNING) << "UNAUTHENTICATED User: " << auth\_user.username();
14. return Status(StatusCode::UNAUTHENTICATED, "username or pin wrong!");
15. }
16. // the user is authenticated

This function is our main function, this is the actual transaction that will take place when this RPC is executed

1. // transaction function
2. auto transfer\_user\_amount = [&](mongocxx::client\_session& session) {
3. // get the transaction options
4. auto txn\_opts = zirconium::transaction\_mgr::get\_transaction\_opts();

This function call will start the transaction with the current transaction options

1. // initiate the transaction
2. session.start\_transaction(txn\_opts);
3. // container for the updated user
4. bsoncxx::stdx::optional<bsoncxx::document::value> updated\_from\_user;

Transaction begins from here

1. // try the transaction
2. try {
3. // read to\_user value from db
4. auto curr\_to\_user = db["bank"].find\_one(session, make\_document(kvp("username", to\_user)));
5. // this is unnecessary, can be ommited
6. if (not curr\_to\_user) {
7. // to\_user does not exist, abort transaction
8. throw zirconium::exception("to\_user does not exist");
9. }
10. // read from\_user value from db
11. auto curr\_from\_user = db["bank"].find\_one(session, make\_document(kvp("username", auth\_user.username())));
12. auto curr\_from\_user\_balance = curr\_from\_user->view()["balance"].get\_int32().value;
13. // guard to check if from\_user has necessary balance
14. if (curr\_from\_user\_balance - transfer\_req->amount() < 0) {
15. // insufficient balance
16. throw zirconium::exception("insufficient balance");
17. }
18. // decrement the balance of from\_user
19. updated\_from\_user = db["bank"].find\_one\_and\_update(
20. session,
21. curr\_from\_user->view(),
22. make\_document(
23. kvp("$inc", make\_document(
24. kvp("balance", -transfer\_req->amount())))),
25. mongocxx::options::find\_one\_and\_update().return\_document(mongocxx::options::return\_document::k\_after));
26. // increment the balance of to\_user
27. db["bank"].find\_one\_and\_update(
28. session,
29. curr\_to\_user->view(),
30. make\_document(
31. kvp("$inc", make\_document(
32. kvp("balance", transfer\_req->amount())))),
33. mongocxx::options::find\_one\_and\_update());
34. // set the new balance as responce
35. balance->set\_value(updated\_from\_user->view()["balance"].get\_int32());
36. } catch (const mongocxx::operation\_exception& oe) {
37. session.abort\_transaction();
38. throw oe;
39. } catch (const zirconium::exception& ex) {
40. session.abort\_transaction();
41. throw ex;
42. }
43. // run this if the transaction was successful
44. auto on\_success = [&](void) {
45. LOG\_S(INFO) << "UPDATED: " << bsoncxx::to\_json(\*updated\_from\_user) << "\n";
46. auto update\_type = transfer\_req->amount() >= 0 ? "credited" : "debited";
47. LOG\_S(INFO) << "UpdateAmount [ Account " << updated\_from\_user->view()["username"].get\_utf8().value.to\_string() << " " << update\_type << " with ₹ " << std::abs(transfer\_req->amount()) << " new balance: " << updated\_from\_user->view()["balance"].get\_int32() << " ]";
48. };
49. // commit the transaction and retry if failure
50. zirconium::transaction\_mgr::commit\_with\_retry(session, on\_success);
51. };
52. // create a session and start a transaction
53. auto session = client->start\_session();
54. try {
55. zirconium::transaction\_mgr::run\_transaction\_with\_retry(transfer\_user\_amount, session);
56. } catch (const mongocxx::operation\_exception& oe) {
57. // some expection occured during commit
58. LOG\_S(ERROR) << "Error during commit: " << oe.what() << ", for session: " << bsoncxx::to\_json(session.id());
59. }
60. } catch (zirconium::exception& ex) {
61. LOG\_S(ERROR) << "Precondition failed: " << ex.what();
62. return Status(StatusCode::FAILED\_PRECONDITION, ex.what());
63. } catch (const std::exception& ex) {
64. LOG\_S(ERROR) << "Internal Server Error: " << ex.what();
65. return Status(StatusCode::INTERNAL, "Internal Server Error, Cannot Transfer Amount");
66. } catch (...) {
67. // something notorious has happened if you reach here
68. // stop the server
69. LOG\_S(FATAL) << "shutting down";
70. }
71. return Status::OK;
72. }

Transactions Manager Code

inline auto commit\_with\_retry(client\_session& session, on\_success\_func succ\_fun) {

    while (true) {

        try {

            session.commit\_transaction();  // Uses write concern set at transaction start.

            LOG\_S(INFO) << "Transaction commited : " << bsoncxx::to\_json(session.id());

            succ\_fun();

            break;

        } catch (const operation\_exception& oe) {

            // Can retry commit

            if (oe.has\_error\_label("UnknownTransactionCommitResult")) {

                LOG\_S(WARNING) << "UnknownTransactionCommitResult, retrying commit operation for session: " << bsoncxx::to\_json(session.id());

                continue;

            } else {

                throw oe;

            }

        }

    }

};

using transaction\_func = std::function<void(client\_session& session)>;

inline auto run\_transaction\_with\_retry(transaction\_func txn\_func, client\_session& session) {

    while (true) {

        try {

            txn\_func(session);  // performs transaction.

            break;

        } catch (const operation\_exception& oe) {

            LOG\_S(ERROR) << "Transaction aborted. Caught exception during transaction: " << oe.what() << ", for session: " << bsoncxx::to\_json(session.id());

            // If transient error, retry the whole transaction.

            if (oe.has\_error\_label("TransientTransactionError")) {

                LOG\_S(WARNING) << "TransientTransactionError, retrying transaction for session: " << bsoncxx::to\_json(session.id());

                continue;

            } else {

                throw oe;

            }

        }

    }

};

Running the application

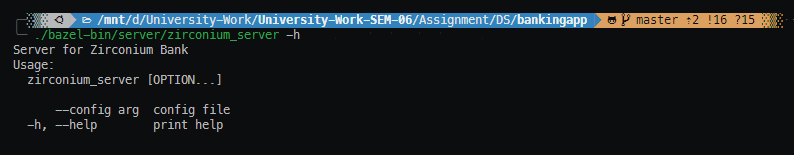


Figure server help option

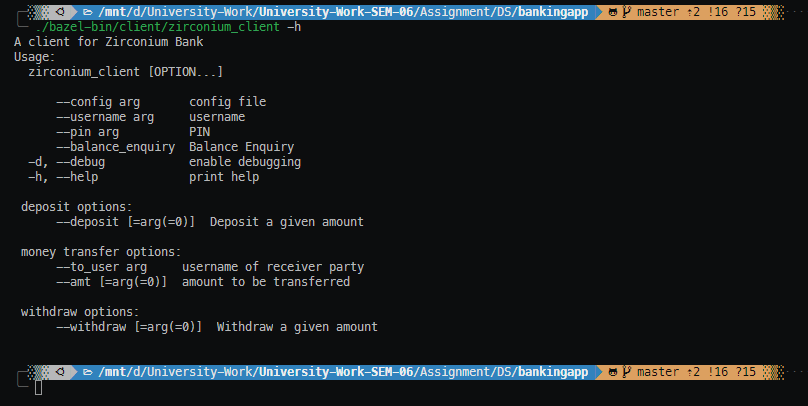


Figure client help option

## Testing of developed Banking Application

Let’s see how the server works with only single operation, then we can move on to more rigorous testing,

Client Logs:

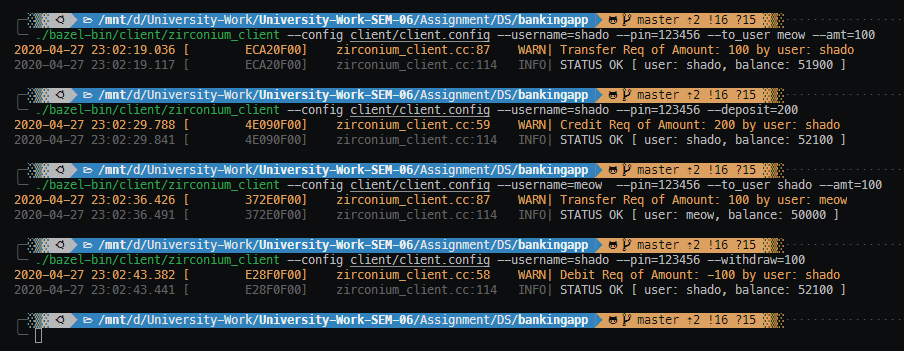


Figure client logs

Server Logs:

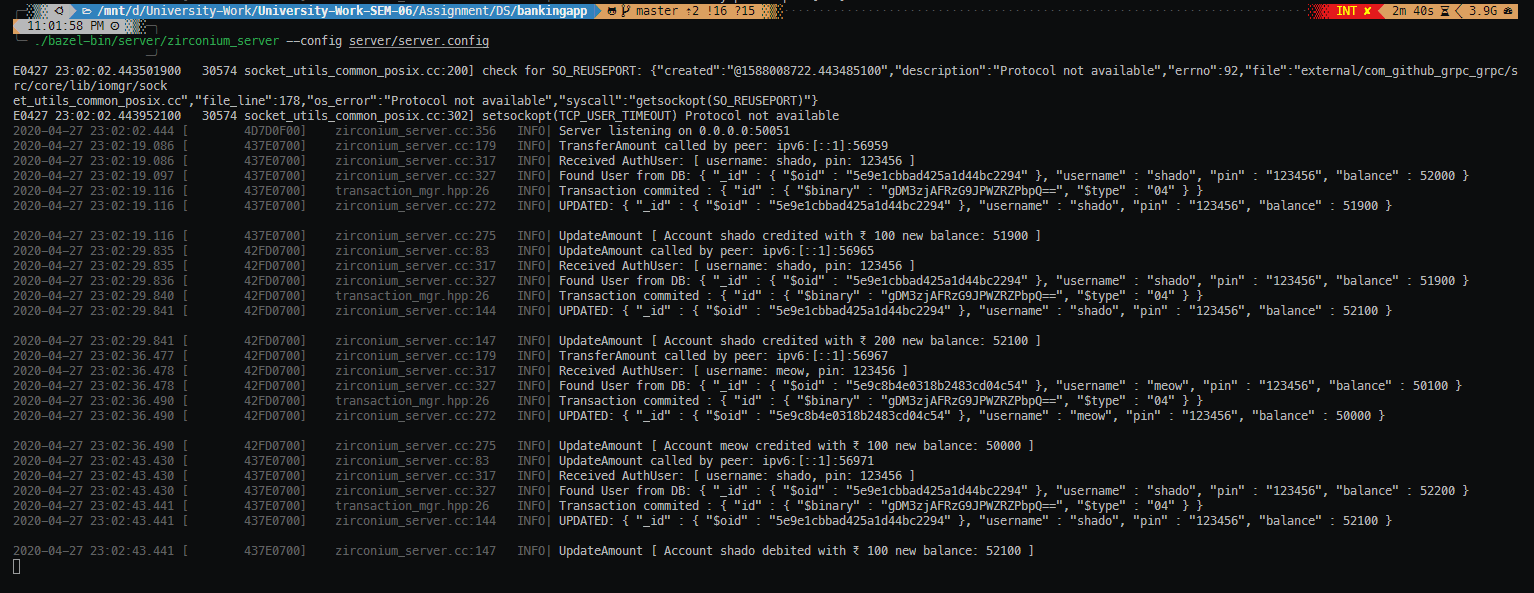


Figure server logs

As we can see the transactions went through successfully, and the balance was updated as intended, also there were no transaction conflicts since we are running the clients serially.

To test the application, we’ve created two users and done concurrent transactions between them, i.e.

TEST Algorithm

1. Transfer Rs.100 from shadow to meow
2. Deposit Rs. 200 to shadow
3. Transfer Rs.100 from meow to shadow
4. Withdraw Rs. 100 from shadow

At the end of one such instance of this algorithm shado’s account will get gain of Rs. 100, so let’s call this Rs 100 shado gain test

All the above operations must run parallelly, so we made use of Linux’s Background processes and GNU Parallel,

Here we do Run the Test Algorithm, everything parallelly, a total of 10 Times, At the end we expect that shadow’s account must have a gain of Rs.1000 while meow’s account’s balance should remain same.

Initially we start with these values,



Figure DB before transaction

Now we run the transaction test, 10 times

seq 10 | parallel -j 10 --workdir $PWD ./test/test\_transactions.sh {}

Client Logs:

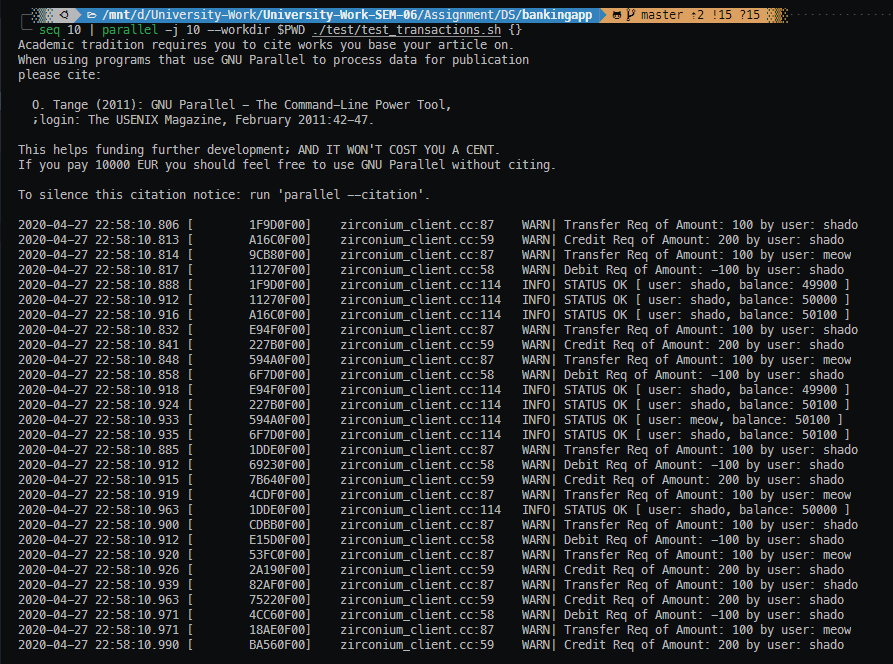


Figure client logs

Server Logs:

Figure server logs

What’s really interesting to observe is that, since we are performing concurrent transactions on a distributed database, there are many conflicts, here our server takes really good care of them, by detecting such conflicts and quickly retrying the transaction again.

DB after running the test



Figure DB after transaction

As expected, we have shado’s account with gain of Rs. 1000, this makes sure that our transactions are working as expected, now let’s try to run our test operation 100 times, with 10 parallel threads, before this we reset the accounts to have Rs 50,000 each.



Figure DB after transaction

As expected now we have Rs 10,000 gain in shado’s account as we ran the Rs 100 gain test a 100 times.

# Question 2

Solution to Part B

## RPC, the need for it and two sample programs

The main usage scenarios of RPC are

* Efficiently connecting polyglot services in microservices style architecture
* Connecting mobile devices, browser clients to backend services
* Generating efficient client libraries

Remote Procedure Calls (RPC)

1. Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
2. Stubs – client-side proxy for the actual procedure on the server.
3. The client-side stub locates the server and marshalls the parameters.
4. The server-side stub receives this message, unpacks the marshalled parameters, and peforms the procedure on the server

The main goal of RPC is to hide the existence of the network from a program. As a result, RPC doesn't quite fit into the OSI model:

The message-passing nature of network communication is hidden from the user. The user doesn't first open a connection, read and write data, and then close the connection. Indeed, a client often doesn not even know they are using the network!

RPC often omits many of the protocol layers to improve performance. Even a small performance improvement is important because a program may invoke RPCs often. For example, on (diskless) Sun workstations, every file access is made via an RPC.

### Example 1

Here we’ve made a simple Streaming NodeJS gRPC Client and Server, where the Server ingests weather data from a stored file and streams the data to a client when the procedure is called.

**server.js**

var PROTO\_PATH = 'weather.proto';

var fs = require('fs');

var \_ = require('lodash');

var grpc = require('grpc');

var protoLoader = require('@grpc/proto-loader');

var packageDefinition = protoLoader.loadSync(

    PROTO\_PATH,

    {

        keepCase: true,

        longs: String,

        enums: String,

        defaults: true,

        oneofs: true

    });

var weather\_proto = grpc.loadPackageDefinition(packageDefinition).weather;

let rawdata = fs.readFileSync('city\_temps.json');

let city\_data = JSON.parse(rawdata);

/\*\*

 \* Implements the GetWeather Service

 \*/

function getWeather(call) {

    var cityName = call.request.city\_name;

    console.log('GetWeather Called by: ' + call.metadata.\_internal\_repr['user-agent']);

    \_.each(city\_data, function (data) {

        data\_res = {

            timestamp: data.timestamp,

            city: cityName,

            temp: data.temp,

            weather\_status: "cloudy",

        }

        call.write(data\_res);

    });

    call.end();

}

/\*\*

 \* Starts an RPC server that receives requests for the Greeter service at the

 \* sample server port

 \*/

function main() {

    var server = new grpc.Server();

    server.addService(weather\_proto.Weather.service, { getWeather: getWeather });

    server.bind('0.0.0.0:50051', grpc.ServerCredentials.createInsecure());

    console.log("Server Started on 0.0.0.0:50051");

    server.start();

}

main();

**client.js**

var PROTO\_PATH = 'weather.proto';

var grpc = require('grpc');

var protoLoader = require('@grpc/proto-loader');

var packageDefinition = protoLoader.loadSync(

    PROTO\_PATH,

    {

        keepCase: true,

        longs: String,

        enums: String,

        defaults: true,

        oneofs: true

    });

var weather\_proto = grpc.loadPackageDefinition(packageDefinition).weather;

var client = new weather\_proto.Weather('localhost:50051',

    grpc.credentials.createInsecure());

function runGetWeather(callback) {

    console.log('STREAMING WEATHER DATA FOR Bangalore');

    var city = {

        city\_name: "bangalore",

    }

    var call = client.getWeather(city);

    call.on('data', function (weather\_data) {

        console.log(weather\_data);

    });

    call.on('end', callback);

}

function main() {

    runGetWeather(function () {

        console.log("END OF STREAM");

    });

}

main();

**weather.proto**

syntax = "proto3";

package weather;

service Weather {

  rpc GetWeather (CityRequest) returns (stream WeatherReply) {}

}

// The request message containing the city name

message CityRequest {

  string city\_name = 1;

}

// The stream of reply containing the weather data

message WeatherReply {

  string timestamp = 1;

  string city = 2;

  string temp = 3;

  string weather\_status = 4;

}

Running the Server

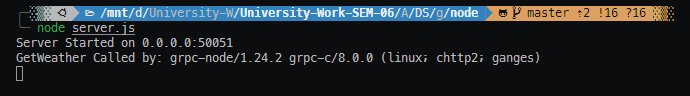


Figure Node gRPC Server

Running the Client

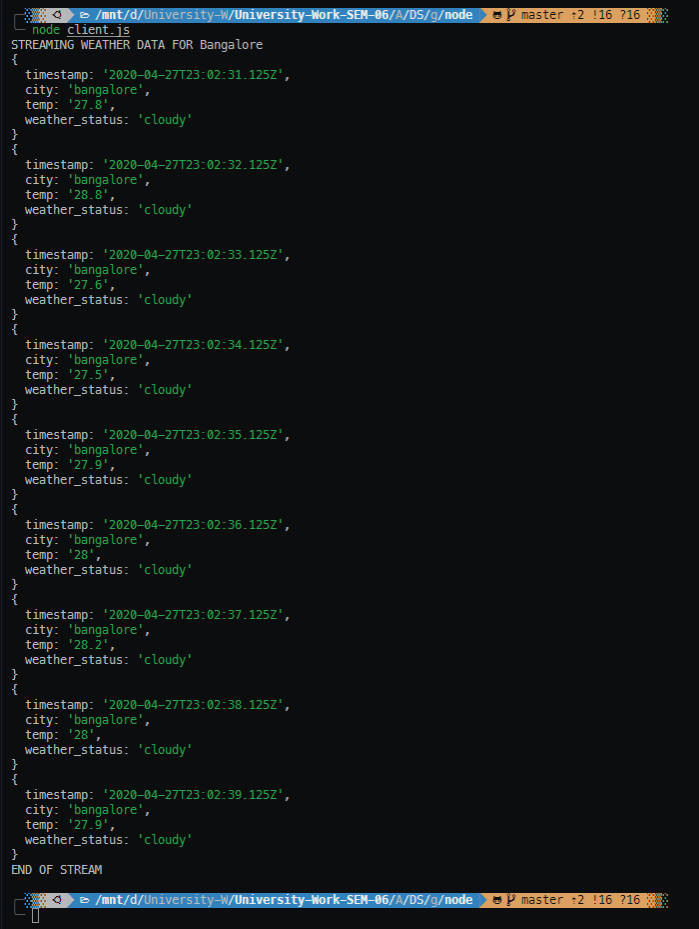


Figure Node gRPC client

The Client here receives weather updates from the server every minute, with the temperature and current weather status.

### Example 2

**server.js**

var PROTO\_PATH = 'marks.proto';

var fs = require('fs');

var \_ = require('lodash');

var grpc = require('grpc');

var protoLoader = require('@grpc/proto-loader');

var packageDefinition = protoLoader.loadSync(

    PROTO\_PATH,

    {

        keepCase: true,

        longs: String,

        enums: String,

        defaults: true,

        oneofs: true

    });

var marks\_proto = grpc.loadPackageDefinition(packageDefinition).marks;

let rawdata = fs.readFileSync('marks.json');

let marks\_data = JSON.parse(rawdata);

/\*\*

 \* Implements the GetMarks Service

 \*/

function getMarks(call) {

    var start\_time = process.hrtime();

    console.log('GetMarks Called by: ' + call.metadata.\_internal\_repr['user-agent']);

    call.on('data', function (student) {

        var data = {

            student\_name: student.name,

            marks: marks\_data[student.name]

        }

        call.write(data);

    });

    call.on('end', function () {

        console.log({ elapsed\_time: process.hrtime(start\_time) });

        call.end();

    });

}

/\*\*

 \* Starts an RPC server that receives requests for the Greeter service at the

 \* sample server port

 \*/

function main() {

    var server = new grpc.Server();

    server.addService(marks\_proto.MarksPortal.service, { getMarks: getMarks });

    server.bind('0.0.0.0:50051', grpc.ServerCredentials.createInsecure());

    console.log("Server Started on 0.0.0.0:50051");

    server.start();

}

main();

**client.js**

var PROTO\_PATH = 'marks.proto';

var \_ = require('lodash');

var grpc = require('grpc');

var protoLoader = require('@grpc/proto-loader');

var packageDefinition = protoLoader.loadSync(

    PROTO\_PATH,

    {

        keepCase: true,

        longs: String,

        enums: String,

        defaults: true,

        oneofs: true

    });

var marks\_proto = grpc.loadPackageDefinition(packageDefinition).marks;

var student = ['satyajit', 'shubham', 'samhitha', 'shikhar', 'shobhan'];

var client = new marks\_proto.MarksPortal('localhost:50051',

    grpc.credentials.createInsecure());

function runGetMarks(callback) {

    console.log('START STREAM');

    var call = client.getMarks();

    call.on('data', function (marks\_data) {

        console.log(marks\_data);

    });

    call.on('end', callback);

    \_.each(student, function (std) {

        var my\_student = { name: std };

        call.write(my\_student);

    });

    call.end();

}

function main() {

    runGetMarks(function () {

        console.log("END OF STREAM");

    });

}

main();

**marks.proto**

syntax = "proto3";

package marks;

service MarksPortal {

  rpc GetMarks (stream StudentRequest) returns (stream MarksReply) {}

}

// The request message containing the student  name

message StudentRequest {

  string name = 1;

}

// The reply containing the marks

message MarksReply {

  string student\_name = 1;

  Marks marks = 2;

}

message Marks {

  string DS = 2;

  string DBMS = 3;

  string Compilers = 4;

  string Graphics = 5;

}

Start Server

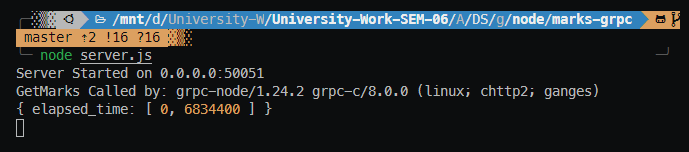


Figure Node Marks Server

Start Client

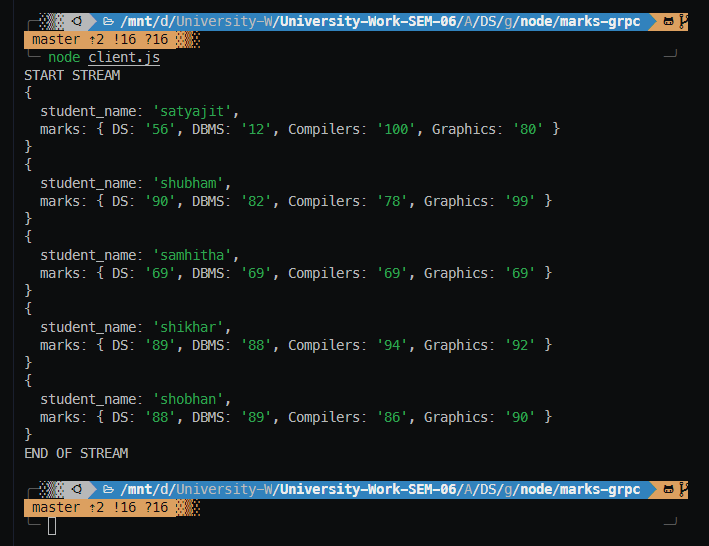


Figure Node Marks Client

Unlike normal procedure calls, many things can go wrong with RPC. Normally, a client will send a request, the server will execute the request and then return a response to the client. What are appropriate semantics for server or network failures? Possibilities:

* Just hang forever waiting for the reply that will never come. This models regular procedure call. If a normal procedure goes into an infinite loop, the caller never finds out. Of course, few users will like such semantics.
* Time out and raise an exception or report failure to the client. Of course, finding an appropriate timer value is difficult. If the remote procedure takes a long time to execute, a timer might time-out too quickly.
* Time out and retransmit the request.

## RMI, the need for it and two sample programs

Remote Method Invocation (RMI)

* RPC + Object Orientation
* Allows objects living in one process to invoke methods of an object living in another process

RMI Programming

* Proxy
  + Behaves like remote object to clients (invoker)
  + Marshals arguments, forwards message to remote object, unmarshals results, returns results to client
* Skeleton
  + Server side stub;
  + Unmarshals arguments, invokes method, marshals results and sends to sending proxy’s method
* Dispatcher
  + Receives the request message from communication module, passes on the message to the appropriate method in the skeleton

The goals for supporting distributed objects in the Java programming language are:

* Support seamless remote invocation on objects in different virtual machines
* Support callbacks from servers to applets
* Integrate the distributed object model into the Java programming language in a natural way while retaining most of the Java programming language's object semantics
* Make differences between the distributed object model and local Java platform's object model apparent
* Make writing reliable distributed applications as simple as possible
* Preserve the type-safety provided by the Java platform's runtime environment
* Support various reference semantics for remote objects; for example live (nonpersistent) references, persistent references, and lazy activation
* Maintain the safe environment of the Java platform provided by security managers and class loaders Underlying all these goals is a general requirement that the RMI model be both simple (easy to use) and natural (fits well in the language). **(Java Docs)**

Advantages of RMI:

* Simple and clean to implement that leads to more robust, maintainable and flexible applications
* Distributed systems creations are allowed while decoupling the client and server objects simultaneously
* It is possible to create zero-install client for the users.
* No client installation is needed except java capable browsers
* At the time of changing the database, only the server objects are to be recompiled but not the server interface and the client remain the same.

Disadvantages of RMI:

* Less efficient than Socket objects.
* Assuming the default threading will allow ignoring the coding, being the servers are thread- safe and robust.
* Cannot use the code out of the scope of java.
* Security issues need to be monitored more closely.

### Example 1

**Calculator.java**

import java.rmi.\*;

public interface Calculator extends Remote {

    public String add(double x, double y) throws RemoteException;

    public String sub(double x, double y) throws RemoteException;

    public String mult(double x, double y) throws RemoteException;

    public String div(double x, double y) throws RemoteException;

}

**CalculatorRemote.java**

import java.rmi.\*;

import java.rmi.server.\*;

public class CalculatorRemote extends UnicastRemoteObject implements Calculator {

    CalculatorRemote() throws RemoteException {

        super();

    }

    public String add(double x, double y) {

        String result;

        result = "Sum : " + (x + y);

        return result;

    }

    public String sub(double x, double y) {

        String result;

        result = "Difference(first value - second value) : " + (x - y);

        return result;

    }

    public String mult(double x, double y) {

        String result;

        result = "Product : " + (x \* y);

        return result;

    }

    public String div(double x, double y) {

        String result;

        if (y == 0)

            result = "Division: Math Error! You cannot divide by 0.";

        else

            result = "Quotient : " + (x / y) + "\nRemainder : " + (x % y);

        return result;

    }

}

**MyClient.java**

import java.rmi.\*;

import java.util.Scanner; //for taking input from the user

public class MyClient {

    public static void main(String args[]) {

        try {

            Calculator stub = (Calculator) Naming.lookup("rmi://localhost:5001/calculator");

            System.out.println("\nSimple Calculator!");

            System.out.println("===========================");

            Scanner reader = new Scanner(System.in);

            char a;

            do {

                System.out.print("\n\nEnter two numbers: ");

                // nextDouble() reads the next double from the keyboard

                double a1 = reader.nextDouble();

                double a2 = reader.nextDouble();

                System.out.print("Enter an operator (+, -, \*, /): ");

                char op = reader.next().charAt(0);

                switch (op) {

                    case '+':

                        System.out.println(stub.add(a1, a2));

                        break;

                    case '-':

                        System.out.println(stub.sub(a1, a2));

                        break;

                    case '\*':

                        System.out.println(stub.mult(a1, a2));

                        break;

                    case '/':

                        System.out.println(stub.div(a1, a2));

                        break;

                    default:

                        System.out.printf("Error: Invalid operator!");

                        return;

                }

                System.out.print("Enter Y/y to continue and anything else to exit: ");

                a = reader.next().charAt(0);

            } while (a == 'Y' || a == 'y');

        } catch (Exception e) {

            System.out.println(e);

        }

    }

}

**MyServer.java**

import java.rmi.\*;

import java.rmi.registry.\*;

public class MyServer {

    public static void main(String args[]) {

        try {

            Calculator stub = new CalculatorRemote();

            Naming.rebind("rmi://localhost:5001/calculator", stub);

        } catch (Exception e) {

            System.out.println(e);

        }

    } // main

} // MyServer

Execution

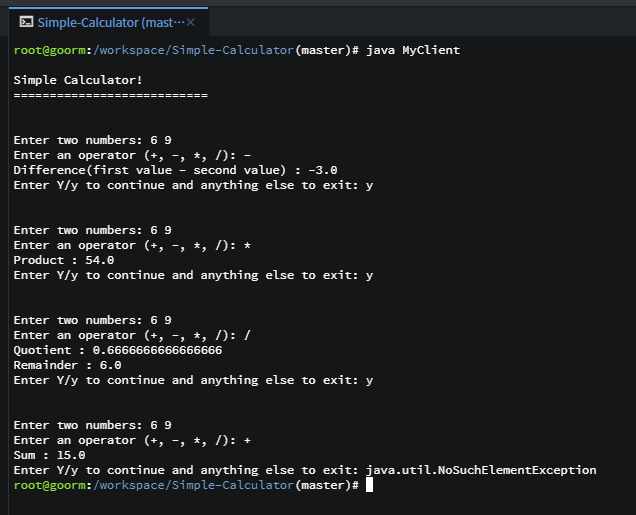


Figure Simple Calculator RMI

### Example 2

RMIInterface.java

package com.example.rmiinterface;

import java.rmi.Remote;

import java.rmi.RemoteException;

public interface RMIInterface extends Remote {

    public String helloTo(String name) throws RemoteException;

}

ClientOperation.java

package com.mkyong.rmiclient;

import java.net.MalformedURLException;

import java.rmi.Naming;

import java.rmi.NotBoundException;

import java.rmi.RemoteException;

import javax.swing.JOptionPane;

import com.example.rmiinterface.RMIInterface;

public class ClientOperation {

    private static RMIInterface look\_up;

    public static void main(String[] args) throws MalformedURLException, RemoteException, NotBoundException {

        look\_up = (RMIInterface) Naming.lookup("//localhost/MyServer");

        String txt = JOptionPane.showInputDialog("What is your name?");

        String response = look\_up.helloTo(txt);

        JOptionPane.showMessageDialog(null, response);

    }

}

ServerOperation.java

package com.example.rmiserver;

import java.rmi.Naming;

import java.rmi.RemoteException;

import java.rmi.server.UnicastRemoteObject;

import com.mkyong.rmiinterface.RMIInterface;

public class ServerOperation extends UnicastRemoteObject implements RMIInterface{

    private static final long serialVersionUID = 1L;

    protected ServerOperation() throws RemoteException {

        super();

    }

    @Override

    public String helloTo(String name) throws RemoteException{

        System.err.println(name + " is trying to contact!");

        return "Server says hello to " + name;

    }

    public static void main(String[] args){

        try {

            Naming.rebind("//localhost/MyServer", new ServerOperation());

            System.err.println("Server ready");

        } catch (Exception e) {

            System.err.println("Server exception: " + e.toString());

          e.printStackTrace();

        }

    }

}

Execution

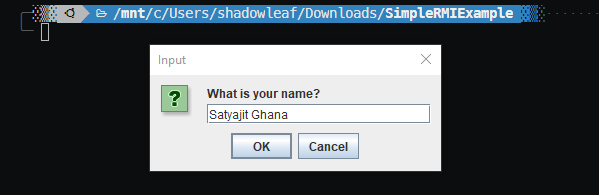


Figure RMI Client

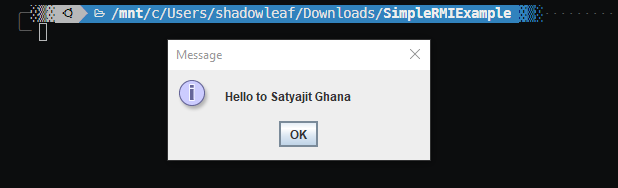


Figure RMI Server

## Similarities and Differences between RPC and RMI

RMI:

* The remote objects are accessed by the references.
* Implements object to object implementation among different java objects to implement distributed communication model.
* RMI passes the objects as parameters to remote methods.
* RMI invokes the remote methods from the objects.

RPC:

* The process is through methods / functions.
* Proxy server is involved in processing the procedure calls.
* Calls a procedure remotely like invoking the methods.
* The remoteness is not exactly transparent to the client.

Difference between RPC and RMI.

* RPC can be used to invoke functions through a proxy functions.
* RMI can used to invoke methods of an object.
* RPC supports procedural programming paradigms thus is C based, while RMI supports object-oriented programming paradigms and is java based.
* RPC protocol generates more overheads than RMI.
* The parameters passed in RPC must be “in-out” which means that the value passed to the procedure and the output value must have the same datatypes. In contrast, there is no compulsion of passing “in-out” parameters in RMI.
* In RPC, references could not be probable because the two processes have the distinct address space, but it is possible in case of RMI.
* The parameters passed to remote procedures in RPC are the ordinary data structures. On the contrary, RMI transits objects as a parameter to the remote method.

The similarities in RPC and RMI

* Both support programming with interfaces
* Both typically constructed on top of request-reply protocols and can offer a range of call semantics such as at-least-once and at-most-once
* Both offer a similar level of transparency –that is, local and remote calls employ the same syntax
* Both RPC and RMI follow the call/return style, where communication is initiated by the client, and the server reactively responds.
* Both RPC and RMI provide communication via local stubs, which support an interface for method calling.
* In RPC and RMI calling and returning is implemented via message passing.
* Both use separate mechanisms for dynamic binding (e.g., object registry, JINI).

# Bibliography

1. Refer to this for complete code

[**https://github.com/satyajitghana/ProjektZirconium**](https://github.com/satyajitghana/ProjektZirconium)

1. <https://platformlab.stanford.edu/Seminar%20Talks/gRPC.pdf>
2. <https://docs.mongodb.com/manual/faq/concurrency/>
3. <https://cs.gmu.edu/~setia/cs571-F02/slides/lec7a.pdf>
4. <https://web.cs.wpi.edu/~cs4514/b98/week8-rpc/week8-rpc.html>
5. <https://mkyong.com/java/java-rmi-hello-world-example/>
6. <http://docs.oracle.com/javase/1.4.2/docs/guide/rmi/spec/rmi-intro3.html>
7. <https://www.careerride.com/RMI-vs-RPC.aspx>
8. <https://techdifferences.com/difference-between-rpc-and-rmi.html>

# Appendix A

Server Code

**zirconium\_server.cc**

// to use c++ style logging

// specifically define this before including loguru.hpp

#define LOGURU\_WITH\_STREAMS 1

#include <grpcpp/grpcpp.h>

#include <bsoncxx/builder/stream/document.hpp>

#include <bsoncxx/json.hpp>

#include <iostream>

#include <map>

#include <memory>

#include <mongocxx/client.hpp>

#include <mongocxx/exception/operation\_exception.hpp>

#include <mongocxx/instance.hpp>

#include <mongocxx/pool.hpp>

#include <mongocxx/stdx.hpp>

#include <mongocxx/uri.hpp>

#include <stdexcept>

#include <string>

#include <thread>

#include "protos/zirconiumbank.grpc.pb.h"

#include "server/transaction\_mgr.hpp"

#include "utils/cxxopts.hpp"

#include "utils/loguru.hpp"

#include "utils/zirconium\_utils.hpp"

using bsoncxx::builder::basic::kvp;

using bsoncxx::builder::basic::make\_array;

using bsoncxx::builder::basic::make\_document;

using bsoncxx::builder::stream::close\_array;

using bsoncxx::builder::stream::close\_document;

using bsoncxx::builder::stream::document;

using bsoncxx::builder::stream::finalize;

using bsoncxx::builder::stream::open\_array;

using bsoncxx::builder::stream::open\_document;

// gRPC namespaces

using grpc::Server;

using grpc::ServerBuilder;

using grpc::ServerContext;

using grpc::Status;

using grpc::StatusCode;

// zirconium grpc-proto namespaces

using zirconium::bank::AuthUser;

using zirconium::bank::Balance;

using zirconium::bank::RequestAmount;

using zirconium::bank::TransferReq;

using zirconium::bank::WebService;

class WebServiceImpl final : public WebService::Service {

    /\*\*

     \* BalanceEnquiry: Gets the balance of a AuthUser

    \*/

    Status BalanceEnquiry(ServerContext\* context, const AuthUser\* auth\_user, Balance\* balance) override {

        // get a client from the pool

        auto client = conn\_pool\_->acquire();

        // log the auth data from the request

        logAuthData\_(auth\_user->username(), auth\_user->pin());

        // check if the user is authenticated and also fetch the document

        auto [is\_authenticated, user] = getAuthData\_(auth\_user->username(), auth\_user->pin(), \*(client));

        if (not is\_authenticated) {

            LOG\_S(WARNING) << "UNAUTHENTICATED User: " << auth\_user->username();

            return Status(StatusCode::UNAUTHENTICATED, "username or pin wrong!");

        }

        // the user is authenticated

        balance->set\_value(user->view()["balance"].get\_int32());

        return Status::OK;

    }

    /\*\*

     \* UpdateAmount : updates the balance of the user with the given amount

     \* Tested Works ✅

    \*/

    Status UpdateAmount(ServerContext\* context, const RequestAmount\* update\_req, Balance\* balance) override {

        LOG\_S(INFO) << "UpdateAmount called by peer: " << context->peer();

        // get a client from the pool

        auto client = conn\_pool\_->acquire();

        // get the "zirconium" db

        auto db = (\*client)["zirconium"];

        const auto auth\_user = update\_req->user();

        logAuthData\_(auth\_user.username(), auth\_user.pin());

        try {

            auto [is\_authenticated, user] = getAuthData\_(auth\_user.username(), auth\_user.pin(), (\*client));

            if (not is\_authenticated) {

                LOG\_S(WARNING) << "UNAUTHENTICATED User: " << auth\_user.username();

                return Status(StatusCode::UNAUTHENTICATED, "username or pin wrong!");

            }

            // the user is authenticated

            auto update\_user\_amount = [&](mongocxx::client\_session& session) {

                // get the transaction options

                auto txn\_opts = zirconium::transaction\_mgr::get\_transaction\_opts();

                // initiate the transaction

                session.start\_transaction(txn\_opts);

                // container for the updated user

                bsoncxx::stdx::optional<bsoncxx::document::value> updated\_user;

                // try the transaction

                try {

                    // read fuser value from db

                    auto curr\_user = db["bank"].find\_one(session, make\_document(kvp("username", auth\_user.username())));

                    auto curr\_user\_balance = curr\_user->view()["balance"].get\_int32().value;

                    // guard to check if from\_user has necessary balance

                    if (curr\_user\_balance + update\_req->value() < 0) {

                        // insufficient balance

                        throw zirconium::exception("insufficient balance");

                    }

                    // increment/decrement the balance of user

                    updated\_user = db["bank"].find\_one\_and\_update(

                        session,

                        curr\_user->view(),

                        make\_document(

                            kvp("$inc", make\_document(kvp("balance", update\_req->value())))),

                        mongocxx::options::find\_one\_and\_update().return\_document(mongocxx::options::return\_document::k\_after));

                    // set the new balance as responce

                    balance->set\_value(updated\_user->view()["balance"].get\_int32());

                } catch (const mongocxx::operation\_exception& oe) {

                    session.abort\_transaction();

                    throw oe;

                }

                // run this if the transaction was successful

                auto on\_success = [&](void) {

                    LOG\_S(INFO) << "UPDATED: " << bsoncxx::to\_json(\*updated\_user) << "\n";

                    auto update\_type = update\_req->value() >= 0 ? "credited" : "debited";

                    LOG\_S(INFO) << "UpdateAmount [ Account " << updated\_user->view()["username"].get\_utf8().value.to\_string() << " " << update\_type << " with ₹ " << std::abs(update\_req->value()) << " new balance: " << updated\_user->view()["balance"].get\_int32() << " ]";

                };

                // commit the transaction and retry if failure

                zirconium::transaction\_mgr::commit\_with\_retry(session, on\_success);

            };

            // create a session and start a transaction

            auto session = (\*client).start\_session();

            try {

                zirconium::transaction\_mgr::run\_transaction\_with\_retry(update\_user\_amount, session);

            } catch (const mongocxx::operation\_exception& oe) {

                LOG\_S(ERROR) << "Error during commit: " << oe.what() << ", for session: " << bsoncxx::to\_json(session.id());

            }

        } catch (zirconium::exception& ex) {

            LOG\_S(ERROR) << "Precondition failed: " << ex.what();

            return Status(StatusCode::FAILED\_PRECONDITION, ex.what());

        } catch (std::exception& ex) {

            LOG\_S(ERROR) << "Internal Server Error: " << ex.what();

            return Status(StatusCode::INTERNAL, "Internal Server Error, Cannot Update Amount");

        }

        return Status::OK;

    }

    /\*\*

     \* Transfer Amount: Transfer amount from current user to to\_user

     \* Tested Works ✅

    \*/

    Status TransferAmount(ServerContext\* context, const TransferReq\* transfer\_req, Balance\* balance) override {

        LOG\_S(INFO) << "TransferAmount called by peer: " << context->peer();

        // get a client from the pool

        auto client = conn\_pool\_->acquire();

        // get the "zirconium" db

        auto db = (\*client)["zirconium"];

        // curr\_user

        const auto auth\_user = transfer\_req->from\_user();

        const auto to\_user = transfer\_req->to\_user();

        // guard to check if the to\_user is same as curr\_user

        if (to\_user == auth\_user.username()) {

            return Status(StatusCode::FAILED\_PRECONDITION, "to\_user is same as curr\_user");

        }

        logAuthData\_(auth\_user.username(), auth\_user.pin());

        try {

            auto [is\_authenticated, user] = getAuthData\_(auth\_user.username(), auth\_user.pin(), (\*client));

            // check if the user is authenticated

            if (not is\_authenticated) {

                LOG\_S(WARNING) << "UNAUTHENTICATED User: " << auth\_user.username();

                return Status(StatusCode::UNAUTHENTICATED, "username or pin wrong!");

            }

            // the user is authenticated

            // transaction function

            auto transfer\_user\_amount = [&](mongocxx::client\_session& session) {

                // get the transaction options

                auto txn\_opts = zirconium::transaction\_mgr::get\_transaction\_opts();

                // initiate the transaction

                session.start\_transaction(txn\_opts);

                // container for the updated user

                bsoncxx::stdx::optional<bsoncxx::document::value> updated\_from\_user;

                // try the transaction

                try {

                    // read to\_user value from db

                    auto curr\_to\_user = db["bank"].find\_one(session, make\_document(kvp("username", to\_user)));

                    // this is unnecessary, can be ommited

                    if (not curr\_to\_user) {

                        // to\_user does not exist, abort transaction

                        throw zirconium::exception("to\_user does not exist");

                    }

                    // read from\_user value from db

                    auto curr\_from\_user = db["bank"].find\_one(session, make\_document(kvp("username", auth\_user.username())));

                    auto curr\_from\_user\_balance = curr\_from\_user->view()["balance"].get\_int32().value;

                    // guard to check if from\_user has necessary balance

                    if (curr\_from\_user\_balance - transfer\_req->amount() < 0) {

                        // insufficient balance

                        throw zirconium::exception("insufficient balance");

                    }

                    // decrement the balance of from\_user

                    updated\_from\_user = db["bank"].find\_one\_and\_update(

                        session,

                        curr\_from\_user->view(),

                        make\_document(

                            kvp("$inc", make\_document(

                                            kvp("balance", -transfer\_req->amount())))),

                        mongocxx::options::find\_one\_and\_update().return\_document(mongocxx::options::return\_document::k\_after));

                    // increment the balance of to\_user

                    db["bank"].find\_one\_and\_update(

                        session,

                        curr\_to\_user->view(),

                        make\_document(

                            kvp("$inc", make\_document(

                                            kvp("balance", transfer\_req->amount())))),

                        mongocxx::options::find\_one\_and\_update());

                    // set the new balance as responce

                    balance->set\_value(updated\_from\_user->view()["balance"].get\_int32());

                } catch (const mongocxx::operation\_exception& oe) {

                    session.abort\_transaction();

                    throw oe;

                } catch (const zirconium::exception& ex) {

                    session.abort\_transaction();

                    throw ex;

                }

                // run this if the transaction was successful

                auto on\_success = [&](void) {

                    LOG\_S(INFO) << "UPDATED: " << bsoncxx::to\_json(\*updated\_from\_user) << "\n";

                    auto update\_type = transfer\_req->amount() >= 0 ? "credited" : "debited";

                    LOG\_S(INFO) << "UpdateAmount [ Account " << updated\_from\_user->view()["username"].get\_utf8().value.to\_string() << " " << update\_type << " with ₹ " << std::abs(transfer\_req->amount()) << " new balance: " << updated\_from\_user->view()["balance"].get\_int32() << " ]";

                };

                // commit the transaction and retry if failure

                zirconium::transaction\_mgr::commit\_with\_retry(session, on\_success);

            };

            // create a session and start a transaction

            auto session = client->start\_session();

            try {

                zirconium::transaction\_mgr::run\_transaction\_with\_retry(transfer\_user\_amount, session);

            } catch (const mongocxx::operation\_exception& oe) {

                // some expection occured during commit

                LOG\_S(ERROR) << "Error during commit: " << oe.what() << ", for session: " << bsoncxx::to\_json(session.id());

            }

        } catch (zirconium::exception& ex) {

            LOG\_S(ERROR) << "Precondition failed: " << ex.what();

            return Status(StatusCode::FAILED\_PRECONDITION, ex.what());

        } catch (const std::exception& ex) {

            LOG\_S(ERROR) << "Internal Server Error: " << ex.what();

            return Status(StatusCode::INTERNAL, "Internal Server Error, Cannot Transfer Amount");

        } catch (...) {

            // something notorious has happened if you reach here

            // stop the server

            LOG\_S(FATAL) << "shutting down";

        }

        return Status::OK;

    }

   public:

    void initiaize(const std::string& mongo\_uri) {

        conn\_pool\_ = std::make\_unique<mongocxx::pool>(mongocxx::uri{mongo\_uri});

    };

   private:

    std::unique\_ptr<mongocxx::pool> conn\_pool\_;

    void logAuthData\_(std::string user, std::string pin) {

        LOG\_S(INFO) << "Received AuthUser: [ username: " << user << ", pin: " << pin << " ]";

    }

    std::tuple<bool, bsoncxx::stdx::optional<bsoncxx::document::value>> getAuthData\_(std::string username, std::string pin, mongocxx::client& client) {

        auto db = client["zirconium"];

        bool is\_authenticated = false;

        bsoncxx::stdx::optional<bsoncxx::document::value> user = db["bank"].find\_one(document{} << "username" << username << finalize);

        if (user) {

            LOG\_S(INFO) << "Found User from DB: " << bsoncxx::to\_json(\*user);

            if (pin == user->view()["pin"].get\_utf8().value.to\_string()) {

                is\_authenticated = true;

            }

        } else {

            LOG\_S(ERROR) << username << " not found !";

        }

        return std::make\_tuple(is\_authenticated, user);

    }

};

void RunServer(const std::string& server\_addr, const std::string& mongo\_uri) {

    WebServiceImpl service;

    // initialize the service with mongo db name "zirconium"

    service.initiaize(mongo\_uri);

    ServerBuilder builder;

    // Listen on the given address without any authentication mechanism.

    builder.AddListeningPort(server\_addr, grpc::InsecureServerCredentials());

    // Register "service" as the instance through which we'll communicate with

    // clients. In this case it corresponds to an \*synchronous\* service.

    builder.RegisterService(&service);

    // Finally assemble the server.

    std::unique\_ptr<Server> server(builder.BuildAndStart());

    LOG\_S(INFO) << "Server listening on " << server\_addr;

    // Wait for the server to shutdown. Note that some other thread must be

    // responsible for shutting down the server for this call to ever return.

    server->Wait();

}

int main(int argc, char\* argv[]) {

    // turn off the printing thread info in the output

    // loguru::g\_preamble\_thread = false;

    // turn off the uptime in the output

    loguru::g\_preamble\_uptime = false;

    cxxopts::Options options("zirconium\_server", "Server for Zirconium Bank");

    options

        .positional\_help("[optional args]")

        .show\_positional\_help();

    options

        .add\_options()("config", "config file", cxxopts::value<std::string>())("h,help", "print help");

    auto result = options.parse(argc, argv);

    if (result.count("help")) {

        std::cout << options.help() << "\n";

        exit(0);

    }

    // check for the config file

    if (!result.count("config")) {

        LOG\_S(ERROR) << "config file not provided";

    }

    auto config = zirconium::parse\_config(result["config"].as<std::string>());

    // check if the config was provided correctly

    if ((config.find("server\_addr") == config.end()) or (config.find("mongo\_addr") == config.end())) {

        throw std::runtime\_error("config file incorrect, server\_addr or mongo\_addr not found");

    }

    // instantiate the mongo connection

    mongocxx::instance instance{};  // This should be done only once.

    std::string mongo\_addr(config["mongo\_addr"]);

    // create a pool

    // mongocxx::pool pool{uri};

    std::string server\_addr(config["server\_addr"]);

    // run the server with the pool

    RunServer(server\_addr, mongo\_addr);

    return 0;

}

transaction\_mgr.hpp

#pragma once

// to use c++ style logging

// specifically define this before including loguru.hpp

#define LOGURU\_WITH\_STREAMS 1

#include <bsoncxx/json.hpp>

#include <iostream>

#include <mongocxx/client.hpp>

#include <mongocxx/exception/operation\_exception.hpp>

#include "utils/loguru.hpp"

namespace zirconium {

namespace transaction\_mgr {

using namespace mongocxx;

using on\_success\_func = std::function<void(void)>;

inline auto commit\_with\_retry(client\_session& session, on\_success\_func succ\_fun) {

    while (true) {

        try {

            session.commit\_transaction();  // Uses write concern set at transaction start.

            LOG\_S(INFO) << "Transaction commited : " << bsoncxx::to\_json(session.id());

            succ\_fun();

            break;

        } catch (const operation\_exception& oe) {

            // Can retry commit

            if (oe.has\_error\_label("UnknownTransactionCommitResult")) {

                LOG\_S(WARNING) << "UnknownTransactionCommitResult, retrying commit operation for session: " << bsoncxx::to\_json(session.id());

                continue;

            } else {

                throw oe;

            }

        }

    }

};

using transaction\_func = std::function<void(client\_session& session)>;

inline auto run\_transaction\_with\_retry(transaction\_func txn\_func, client\_session& session) {

    while (true) {

        try {

            txn\_func(session);  // performs transaction.

            break;

        } catch (const operation\_exception& oe) {

            LOG\_S(ERROR) << "Transaction aborted. Caught exception during transaction: " << oe.what() << ", for session: " << bsoncxx::to\_json(session.id());

            // If transient error, retry the whole transaction.

            if (oe.has\_error\_label("TransientTransactionError")) {

                LOG\_S(WARNING) << "TransientTransactionError, retrying transaction for session: " << bsoncxx::to\_json(session.id());

                continue;

            } else {

                throw oe;

            }

        }

    }

};

inline auto get\_transaction\_opts() {

    mongocxx::options::transaction txn\_opts;

    // read and write concerns and read preference

    mongocxx::write\_concern wc\_majority{};

    wc\_majority.acknowledge\_level(mongocxx::write\_concern::level::k\_majority);

    mongocxx::read\_concern rc\_local{};

    rc\_local.acknowledge\_level(mongocxx::read\_concern::level::k\_local);

    mongocxx::read\_preference rp\_primary{};

    rp\_primary.mode(mongocxx::read\_preference::read\_mode::k\_primary);

    txn\_opts.write\_concern(wc\_majority);

    txn\_opts.read\_concern(rc\_local);

    txn\_opts.read\_preference(rp\_primary);

    return txn\_opts;

};

}  // namespace transaction\_mgr

class exception : public std::exception {

   public:

    exception(const std::string& message) : msg\_(message) {}

    virtual const char\* what() const throw() { return msg\_.c\_str(); }

   private:

    std::string msg\_;

};

}  // namespace zirconium

**zirconiumbank.proto**

syntax = "proto3";

// ZirconiumBank (ZrB)

/\* The User can requestt for the following from the server

\* 1. Update Amount

\* 2. Transfer Amount

\* 3. Balance Enquiry

\* \*/

/\* The Request should always contain

\* 1. Username

\* 2. 6-digit PIN

\* \*/

// The C++ namesapce will be zirconium::bank

package zirconium.bank;

// Interface exported by the server

service WebService {

rpc UpdateAmount(RequestAmount) returns (Balance) {}

rpc BalanceEnquiry(AuthUser) returns (Balance) {}

rpc TransferAmount(TransferReq) returns (Balance) {}

}

message TransferReq {

AuthUser from\_user = 1;

string to\_user = 2;

int32 amount = 3;

}

message AuthUser {

string username = 1;

string pin = 2;

}

message RequestAmount {

AuthUser user = 1;

int32 value = 2;

}

message Balance {

int32 value = 1;

}

# Appendix B

Client Code

**zirconium\_client.cc**

// to use c++ style logging

// specifically define this before including loguru.hpp

#define LOGURU\_WITH\_STREAMS 1

#include <grpcpp/grpcpp.h>

#include <iostream>

#include <memory>

#include <sstream>

#include <string>

#include "protos/zirconiumbank.grpc.pb.h"

#include "utils/cxxopts.hpp"

#include "utils/loguru.hpp"

#include "utils/zirconium\_utils.hpp"

// gRPC namespaces

using grpc::Channel;

using grpc::ClientContext;

using grpc::Status;

// zirconium grpc-proto namespaces

using zirconium::bank::AuthUser;

using zirconium::bank::Balance;

using zirconium::bank::RequestAmount;

using zirconium::bank::TransferReq;

using zirconium::bank::WebService;

class ZirconiumClient {

   public:

    ZirconiumClient(std::shared\_ptr<Channel> channel) : stub\_(WebService::NewStub(channel)) {}

    std::string BalanceEnquiry(const std::string& user, const std::string& pin) {

        // Data that we will send to the server

        AuthUser authuser\_req;

        authuser\_req.set\_username(user);

        authuser\_req.set\_pin(pin);

        // The Response we will get

        Balance balance\_res;

        // Context for the client. It could be used to convey extra information to

        // the server and/or tweak certain RPC behaviors.

        ClientContext context;

        // The actual RPC

        Status status = stub\_->BalanceEnquiry(&context, authuser\_req, &balance\_res);

        return getResponse\_(status, user, balance\_res);

    }

    std::string UpdateAmount(const std::string& user, const std::string& pin, int update\_amt) {

        // the despoit amount request

        RequestAmount update\_amt\_req;

        LOG\_IF\_S(WARNING, update\_amt == 0) << "update amount = 0";

        LOG\_IF\_S(WARNING, update\_amt < 0) << "Debit Req of Amount: " << update\_amt << " by user: " << user;

        LOG\_IF\_S(WARNING, update\_amt > 0) << "Credit Req of Amount: " << update\_amt << " by user: " << user;

        // create the request payload

        update\_amt\_req.mutable\_user()->set\_username(user);

        update\_amt\_req.mutable\_user()->set\_pin(pin);

        update\_amt\_req.set\_value(update\_amt);

        // The Response we will get

        Balance balance\_res;

        ClientContext context;

        // The DepositAmount RPC

        Status status = stub\_->UpdateAmount(&context, update\_amt\_req, &balance\_res);

        return getResponse\_(status, user, balance\_res);

    }

    std::string TransferAmount(const std::string& user, const std::string& pin, const std::string& to\_user, int trans\_amt) {

        // the transfer amount request

        TransferReq trans\_amt\_req;

        if (trans\_amt < 0) {

            LOG\_S(ERROR) << "Transfer Amount cannot be negative, wtf you trying to do?";

            exit(0);

        }

        LOG\_IF\_S(WARNING, trans\_amt == 0) << "transfer amount = 0";

        LOG\_IF\_S(WARNING, trans\_amt > 0) << "Transfer Req of Amount: " << trans\_amt << " by user: " << user;

        trans\_amt\_req.mutable\_from\_user()->set\_username(user);

        trans\_amt\_req.mutable\_from\_user()->set\_pin(pin);

        trans\_amt\_req.set\_to\_user(to\_user);

        trans\_amt\_req.set\_amount(trans\_amt);

        // The Response we get

        Balance balance\_res;

        ClientContext context;

        // The Transfer Amount RPC

        Status status = stub\_->TransferAmount(&context, trans\_amt\_req, &balance\_res);

        return getResponse\_(status, user, balance\_res);

    }

   private:

    std::unique\_ptr<WebService::Stub> stub\_;

    ClientContext ctx;

    std::string getResponse\_(Status status, std::string user, Balance balance\_res) {

        if (status.ok()) {

            std::stringstream response;

            response << "STATUS OK [ user: " << user << ", balance: " << std::to\_string(balance\_res.value()) << " ]";

            LOG\_S(INFO) << response.str();

            return response.str();

        } else {

            std::stringstream response;

            response << "RPC FAILED [ " << status.error\_code() << ": " << status.error\_message();

            LOG\_S(ERROR) << response.str();

            return response.str();

        }

    }

};

int main(int argc, char\* argv[]) {

    // turn off the printing thread info in the output

    // loguru::g\_preamble\_thread = false;

    // turn off the uptime in the output

    loguru::g\_preamble\_uptime = false;

    try {

        cxxopts::Options options("zirconium\_client", "A client for Zirconium Bank");

        options

            .positional\_help("[optional args]")

            .show\_positional\_help();

        options.allow\_unrecognised\_options()

            .add\_options()("config", "config file", cxxopts::value<std::string>())("username", "username", cxxopts::value<std::string>())("pin", "PIN", cxxopts::value<std::string>())("balance\_enquiry", "Balance Enquiry", cxxopts::value<bool>()->implicit\_value("false"))("d,debug", "enable debugging")("h,help", "print help");

        options.add\_options("withdraw", {{"withdraw", "Withdraw a given amount", cxxopts::value<int>()->implicit\_value("0")}});

        options.add\_options("deposit", {{"deposit", "Deposit a given amount", cxxopts::value<int>()->implicit\_value("0")}});

        options.add\_options("money transfer", {{"to\_user", "username of receiver party", cxxopts::value<std::string>()},

                                               {"amt", "amount to be transferred", cxxopts::value<int>()->implicit\_value("0")}});

        auto result = options.parse(argc, argv);

        if (result.count("help")) {

            std::cout << options.help() << "\n";

            exit(0);

        }

        // check for the config file

        if (!result.count("config")) {

            LOG\_S(ERROR) << "config file not provided";

        }

        auto config = zirconium::parse\_config(result["config"].as<std::string>());

        // check if the config was provided correctly

        if (config.find("server\_addr") == config.end()) {

            throw std::runtime\_error("config file incorrect, server\_addr not found");

        }

        std::string server\_addr(config["server\_addr"]);

        // Instantiate the client. It requires a channel, out of which the actual RPCs

        // are created. This channel models a connection to an endpoint (in this case,

        // localhost at port 50051). We indicate that the channel isn't authenticated

        // (use of InsecureChannelCredentials()).

        ZirconiumClient zirconium\_bank(grpc::CreateChannel(server\_addr, grpc::InsecureChannelCredentials()));

        // check if the username or the pin is not given as argument

        if (!(result.count("username") and result.count("pin"))) {

            LOG\_S(ERROR) << "username or pin not given";

            exit(1);

        }

        // get the given values from the arguments

        std::string username = result["username"].as<std::string>();

        std::string pin = result["pin"].as<std::string>();

        // check if the strings are empty

        if (username.empty() or pin.empty()) {

            LOG\_S(ERROR) << "username or pin empty"

                         << "\n";

            exit(1);

        }

        // the username and pin is provided properly

        // deposit action argument passed

        if (result.count("deposit")) {

            // the deposit amount

            int dep\_amt = result["deposit"].as<int>();

            std::string reply = zirconium\_bank.UpdateAmount(username, pin, dep\_amt);

        }

        // withdraw action argument passed

        if (result.count("withdraw")) {

            // the withdraw amount

            int with\_amt = result["withdraw"].as<int>();

            std::string reply = zirconium\_bank.UpdateAmount(username, pin, -with\_amt);

        }

        // money transfer argument passed

        if (result.count("to\_user")) {

            int trans\_amt = result["amt"].as<int>();

            std::string to\_user = result["to\_user"].as<std::string>();

            if ((to\_user == username) or to\_user.empty()) {

                LOG\_S(ERROR) << "to\_user cannot be emoty nor the same user as current user";

                exit(1);

            }

            std::string reply = zirconium\_bank.TransferAmount(username, pin, to\_user, trans\_amt);

        }

        // balance enquiry argument passed

        if (result["balance\_enquiry"].as<bool>()) {

            std::string reply = zirconium\_bank.BalanceEnquiry(username, pin);

        }

    } catch (const cxxopts::OptionException& e) {

        LOG\_S(ERROR) << "error parsing options: " << e.what();

        exit(1);

    }

    return 0;

}