# Códigos

## 1 *numbers.proto*

```
syntax = "proto3";

package numbers;

service ArrayOperator {
    rpc ArrayPow2(NumberArray) returns (NumberArray) {}

    rpc ArrayMultiplyBy(NumberArrayParameter) returns (NumberArray) {}

    rpc ArrayInc(NumberArray) returns (NumberArray) {}
}

message NumberArray {
    repeated double value = 1;
}

message NumberArrayParameter {
    repeated double value = 1;
    double parameter = 2;
}
```

## 2 *array_rpc_server.cpp*

```
#include <iostream>
#include <memory>
#include <string>

#include <grpc++/grpc++.h>

#include "numbers.grpc.pb.h"

using grpc::Server;
using grpc::ServerBuilder;
using grpc::ServerContext;
using grpc::Status;
using grpc::ServerReaderWriter;
using numbers::NumberArray;
using numbers::NumberArrayParameter;
using numbers::ArrayOperator;
```

```
// Logic and data behind the server's behavior.
class ArrayOperatorServiceImpl final : public ArrayOperator::Service {
  /*
  All three functions just read each value in the 'in' parameter,
  operate it and write it back to out
  */
  Status ArrayPow2(ServerContext* context, const NumberArray* in,
                   NumberArray* out) override {
    for (int i = 0; i < in->value_size(); ++i) {
      out->add_value(in->value(i)*in->value(i));
    }
    return Status::OK;
  }

  Status ArrayInc(ServerContext* context, const NumberArray* in,
                   NumberArray* out) override {
    for (int i = 0; i < in->value_size(); ++i) {
      out->add_value(in->value(i)+1.0);
    }
    return Status::OK;
  }

  Status ArrayMultiplyBy(ServerContext* context, const NumberArrayParameter* in,
                   NumberArray* out) override {
    for (int i = 0; i < in->value_size(); ++i) {
      out->add_value(in->value(i)*in->parameter());
    }
    return Status::OK;
  }
};

void RunServer() {
  std::string server_address("0.0.0.0:50051");
  ArrayOperatorServiceImpl service;

  ServerBuilder builder;
  // Listen on the given address without any authentication mechanism.
  builder.AddListeningPort(server_address, 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());
  std::cout << "Server listening on " << server_address << std::endl;

  // 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) {
  RunServer();

  return 0;
```

```
}
```

## 3 *array_rpc_client.cpp*

```
#include <thread>
#include <iostream>
#include <memory>
#include <string>
#include <cstring>
#include <cstdio>
#include <random>
#include <unistd.h>
#include <sys/time.h>

#include <grpc++/grpc++.h>

#include "numbers.grpc.pb.h"

using grpc::Channel;
using grpc::ClientContext;
using grpc::Status;
using grpc::ClientReaderWriter;
using numbers::NumberArray;
using numbers::NumberArrayParameter;
using numbers::ArrayOperator;

class ArrayOperatorClient {
 public:
  ArrayOperatorClient(std::shared_ptr<Channel> channel)
      : stub_(ArrayOperator::NewStub(channel)) {}


  void arrayPow2(double* vector, size_t vecSize) {
    //break rpc call into chunks
    this->chunkify(&ArrayOperatorClient::_arrayPow2, vector, vecSize, 0.0);
  }

  void arrayInc(double* vector, size_t vecSize) {
    //break rpc call into chunks
    this->chunkify(&ArrayOperatorClient::_arrayInc, vector, vecSize, 0.0);
  }

  void arrayMultiplyBy(double* vector, size_t vecSize, double parameter) {
    //break rpc call into chunks
    this->chunkify(&ArrayOperatorClient::_arrayMultiplyBy, vector, vecSize, parameter);
  }


 private:

  //Break thread vector into smaller chunks to avoid grpc warnings or errors regarding big messages
  void chunkify ( void (ArrayOperatorClient::*function)(double*, size_t, double),
```

```
                double* vector,
                size_t vecSize,
                double parameter)
{
  //adequate limit for double quantity
  const size_t limit = 524288;
  //amount of doubles already sent
  size_t chunkSent = 0;

  if (vecSize <= limit) {
    //vecSize <= limit: OK TO SEND
    (this->*function)(vector, vecSize, parameter);
  } else {
    size_t iterations = (vecSize/limit);
    // vecSize > limit: BREAK IN iterations+1 chunks
    //full chunks are those which have 'limit' doubles
    for (int i = 0; i < iterations; ++i)
    {
      chunkSent = i*limit;
      (this->*function)(vector+chunkSent, limit, parameter);
    }
    //last chunk, may have 0 to 'limit' doubles
    (this->*function)(vector+((iterations)*limit), vecSize-(iterations*limit), parameter);
  }
}

void _arrayPow2(double* vector, size_t vecSize, double garbage) {
  // Data we are sending to the server. Copy values
  NumberArray in;
  for (int i = 0; i < vecSize; ++i) {
    in.add_value(vector[i]);
  }

  // Container for the data we expect from the server.
  NumberArray out;

  // 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_->ArrayPow2(&context, in, &out);

  // Restore values
  for (int i = 0; i < vecSize; ++i) {
    vector[i] = out.value(i);
  }

  // Act upon its status. No treatment implemented
  if (status.ok()) {
    return;
  } else {
    return;
  }
```

```
}

void _arrayInc(double* vector, size_t vecSize, double garbage) {
  // Data we are sending to the server. Copy values
  NumberArray in;
  for (int i = 0; i < vecSize; ++i) {
    in.add_value(vector[i]);
  }

  // Container for the data we expect from the server.
  NumberArray out;

  // 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_->ArrayInc(&context, in, &out);

  // Restore values
  for (int i = 0; i < vecSize; ++i) {
    vector[i] = out.value(i);
  }

  // Act upon its status. No treatment implemented
  if (status.ok()) {
    return;
  } else {
    return;
  }
}

void _arrayMultiplyBy(double* vector, size_t vecSize, double parameter) {
  // Data we are sending to the server. Copy values
  NumberArrayParameter in;
  in.set_parameter(parameter);
  for (int i = 0; i < vecSize; ++i) {
    in.add_value(vector[i]);
  }

  // Container for the data we expect from the server.
  NumberArray out;

  // 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_->ArrayMultiplyBy(&context, in, &out);

  //Restore values
  for (int i = 0; i < vecSize; ++i) {
    vector[i] = out.value(i);
  }
```

```
    // Act upon its status. No treatment implemented
    if (status.ok()) {
      return;
    } else {
      return;
    }
  }

  std::unique_ptr<ArrayOperator::Stub> stub_;
};

// Thread function for thread 'tid' to fill 'n' values on array 'element'
void fillArrayCorrect(int tid, long n, double* element, int numThreads) {
  // Use C++11 Mersenne Twister's random number generator
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_int_distribution<> dis(-100,100);
  // Number of elements generated per thread
  int nPerThread = n/numThreads;
  // Starting index for this thread
  int index = tid * nPerThread;
  for ( int i = index; i < index + nPerThread; i ++) {
    element[i] = dis(gen);
  }
}

// Fill vector 'element' with size 'n' using 'numThreads' threads
void randomAllocatedVector (double* element, long n, int numThreads) {
  std::thread *th = new std::thread[numThreads];
  //spawn worker threads and join after the work is done
  for( int i = 0; i < numThreads ; i++) {
    th[i] = std::thread(fillArrayCorrect,i,n,element, numThreads);
  }
  for(int i = 0; i < numThreads; i++) {
    th[i].join();
  }
  delete [] th;
}

// Run one execution of application
// This function was created to make it easier to run 10 tests
//and take timing average and standard deviation
double runApplication (int& NUM_THREADS,
                       double* vector,
                       long& n,
                       size_t& nPerThreads,
                       int& opcode,
                       double& parameter)
{

  std::thread* clients = new std::thread[NUM_THREADS];

  //create random vectors
```

```
  randomAllocatedVector(vector, n, NUM_THREADS);

  //start timing
  struct timeval start, end;
  gettimeofday(&start, NULL);

  for (int i = 0; i < NUM_THREADS; ++i) {
    // launch thread with c++11's lambda function
    // lambda is defined as: [captured variables](parameters){function}
    clients[i] = std::thread([i, vector, nPerThreads, opcode, parameter](){
      ArrayOperatorClient greeter(grpc::CreateChannel(
      "localhost:50051", grpc::InsecureChannelCredentials()));
      // find out where this thread's vector part begins with
      //pointer arithmetic
      double *localVector = vector + nPerThreads*i;
      // Based on 'opcode' choose operation
      switch (opcode) {
        case 0:
          greeter.arrayInc(localVector, nPerThreads);
          break;
        case 1:
          greeter.arrayPow2(localVector, nPerThreads);
          break;
        case 2:
          greeter.arrayMultiplyBy(localVector, nPerThreads, parameter);
          break;
        default:
          break;
      }

    });
  }
  // join clients after they're done
  for (int i = 0; i < NUM_THREADS; ++i) {
    clients[i].join();
  }

  //end timing
  gettimeofday(&end, NULL);
  double seconds = ((end.tv_sec  - start.tv_sec) * 1000000u +
    end.tv_usec - start.tv_usec) / 1.e6;

  //avoid memory leaks
  delete [] clients;

  // print to have a notion of progress
  std::cout<<"."<<std::flush;

  return seconds;
}

int main(int argc, char** argv) {
  // Usage explanation
  if (argc < 5)
```

```
{
  std::cout<<"Usage is:\n"<<argv[0]<<
    " <amount of numbers to generate>"<<
    " <amount of threads>" <<
    " <opcode = 0,1,2>" <<
    " <parameter>\n" <<
    " <optional: number of executions. default = 1>"
    " OP CODES:\n" <<
    "    0 - Increment by 1;\n" <<
    "    1 - Apply power of 2;\n" <<
    "    2 - Multiply by parameter;\n" <<
    "\nDISCLAIMER: Even though not every operation uses a parameter," <<
    " you have to specify one when using the 'number of executions'"<<
    " input.\n" <<
    std::endl;
  return 0;
}

if (!isdigit(argv[1][0]) || !isdigit(argv[2][0]) || !isdigit(argv[3][0]))
{
  std::cerr<<"Arguments should be numbers!"<<std::endl;
  return 0;
}
// size of vector
long n = atol(argv[1]);
// number of threads
int NUM_THREADS = atoi(argv[2]);
// operation
int opcode = atoi(argv[3]);
if (opcode < 0 || opcode > 2) {
  std::cerr<<"IMPOSSIBLE OPCODE!"<<std::endl;
  return 0;
}

// Parameter for parameterized operations
double parameter = 4.0;
if (isdigit(argv[4][0])){
  parameter = atof(argv[4]);
}

// How many times is execution repeated
int repeat = 1;
if (argc > 5) {
  if (isdigit(argv[5][0])){
    repeat = atoi(argv[5]);
  }
}

// Number of elements processed by each thread
size_t nPerThreads = n/NUM_THREADS;
// Vector to be operated
double* vector = new double[n];
// Execution times, for standard deviation calculation
double* times = new double[repeat];
```

```
  // Mean time
  double meanTime = 0.0;
  // Run all executions
  for (int i = 0; i < repeat; ++i) {
    times[i] = runApplication(NUM_THREADS, vector, n,
                              nPerThreads, opcode, parameter);
    meanTime += times[i]/(double)repeat;
  }
  // Calculate standard deviation
  double stdDev = 0.0;
  for (int i = 0; i < repeat; ++i) {
    stdDev = ((meanTime-times[i])*(meanTime-times[i]))/(double)repeat;
  }
  stdDev = std::sqrt(stdDev);
  // Log results to console
  std::cout<<"\nMean time: "<<meanTime<<" seconds"<<std::endl;
  std::cout<<"\nStandard Deviation: "<<stdDev<<" seconds"<<std::endl;
  // Write results to file
  char fname[80];
  FILE* fout;
  sprintf(fname,"times.log");
  fout = fopen(fname,"a");
  fprintf(fout, "vector size: %ld", n);
  fprintf(fout, " nThreads: %d opcode: %d", NUM_THREADS, opcode);
  fprintf(fout, " sdev: %f s \n", stdDev);
  fprintf(fout, " mean time: %f s \n", meanTime);
  fclose(fout);

  //avoid memory leaks
  delete [] vector;
  return 0;
}
```

# 4 *client.lua*

```
-- load namespace
local socket = require "socket"

-- server host and port
local host, port = "localhost", 51034

-- convert host name to ip address
local ip = assert(socket.dns.toip(host))
-- create a new UDP object
local udp = assert(socket.udp())

-- function used for sleeping
-- hacked from socket
-- works with subsecond values
function sleep(sec)
    socket.select(nil, nil, sec)
end
```

```lua
-- main code
function writer()
    -- writer name
    local name = arg[1]~=nil and arg[1] or "Bob"
    -- random seed
    local seed = (arg[2]~=nil and type(arg[2])=="number") and arg[2] or os.time()
    math.randomseed(seed)

    -- variable to store temporary response
    local response

    -- first sleep
    sleep(math.random())
    assert(udp:sendto(0x3, ip, port)) --CONNECT

    -- write 100 lines
    for i=1,100 do
        -- contact coordinator, send 0x0 == REQUEST
        assert(udp:sendto(0x0, ip, port))
        -- wait for response
        response, err = assert(udp:receive(1))
        if not response then
            print("Received error: ", err)
        elseif response == tostring(0x2) then
            -- 0x2 == GRANT
            -- critical session start
                local file = io.open("critical.txt", "a")
                file:write(name .. " is writing " .. i .. "\n")
                file:close()
            -- critical session end
            -- send 0x1 == RELEASE to coordinator
            assert(udp:sendto(0x1, ip, port))
            sleep(math.random())
        else
            -- received unknown message
            print("Received: ", response)
        end
    end

    assert(udp:sendto(0x4, ip, port)) --DISCONNECT
end

-- run script
writer()

print("Writer out!")
```

## 5 *coordinator.lua*

```lua
-- load dependencies
local socket = require "socket"
```

```
local queue = require "queue"

-- this coordinator's host and port
local port = 51034
local server = socket.udp()

-- make socket
server:setsockname("*",port)

-- Coordinator execution function
function handler(skt)
    -- Coordinator's log
    local file = io.open("log.txt", "w")
    file:write("COORDINATOR LOG\n")
    file:close()

    -- print to console
    print("UDP mutex coordinator")
    print("receiving...")
    while true do
        -- receive message on socket
        -- store sender infos on variables
        s, err, sktport, sktip = skt:receivefrom(1)
        -- treat what was received
        if not s then
            print("Receive error: ", err)
            break
        else
            -- CENTRALIZED MUTEX ALGORITHM start
                -- Received REQUEST == 0x0
                if s==tostring(0x0) then
                    message = "REQUEST"
                    if Queue.isempty(q) then
                        -- send GRANT == 0x2
                        skt:sendto(0x2, "*", sktport)
                        -- Mark that a grant happened
                        grant = true
                        granted = sktport
                    end
                    Queue.push(q, sktport)

                -- Received RELEASE == 0x1
                elseif s==tostring(0x1) then
                    message = "RELEASE"
                    Queue.pop(q)
                    if not Queue.isempty(q) then
                        -- send GRANT == 0x2
                        skt:sendto(0x2, "*", Queue.head(q))
                        -- Log grant
                        grant = true
                        granted = Queue.head(q)
                    end
            -- CENTRALIZED MUTEX ALGORITHM end
                -- Received NEW CONNECTION == 0x3
```

```
    elseif s==tostring(0x3) then
        message = "CONNECT"
        if not started then
            strated = true
            start_time = os.time()
        end
        clientcount = clientcount + 1
    -- Received END CONNECTION == 0x4
    elseif s==tostring(0x4) then
        message = "DISCONNECT"
        clientcount = clientcount - 1
        if clientcount <= 0 then
            started = false
            end_time = os.time()
            local elapsed_time = os.difftime(end_time, start_time)
            print("Time elapsed: ", elapsed_time)
            -- Write log
            file = io.open("log.txt", "a")
            file:write(
                "ELAPSED (" ..
                os.date("%X") ..
                ") SECONDS (" ..
                elapsed_time ..
                ")\n"
            )
            file:close()
            return
        end
    -- Received unknown
    else
        message = "UNKNOWN"
    end

-- Write log
file = io.open("log.txt", "a")

file:write(
    message ..
    " (" ..
    os.date("%X") ..
    ") PROCESS (" ..
    sktport ..
    ")\n"
)

file:write(
    "QUEUE (" ..
    os.date("%X") ..
    ") SIZE (" ..
    Queue.size(q) ..
    ")\n"
)

if message=="CONNECT" or message=="DISCONNECT" then
```

```
                file:write(
                    "CONNECTIONS (" ..
                    os.date("%X") ..
                    ") COUNT (" ..
                    clientcount ..
                    ")\n"
                )
            end

            if grant then
                file:write(
                    "GRANT (" ..
                    os.date("%X") ..
                    ") PROCESS (" ..
                    granted ..
                    ")\n"
                )
                grant = false
            end

            file:close()

        end

    end

    print("no more clients...")

end

-- Queue for waiting writers
q = Queue.new()
started = false
grant = false
granted = 0
clientcount = 0
start_time = 0
end_time = 0

-- Create server and start serving
handler(server)
```

## 6 *queue.lua*

```
Queue = {}
function Queue.new ()
  return {first = 0, last = -1}
end

function Queue.push (queue, value)
  local last = queue.last + 1
  queue.last = last
```

```
  queue[last] = value
end

function Queue.pop (queue)
  local first = queue.first
  if first > queue.last then error("queue is empty") end
  local value = queue[first]
  queue[first] = nil        -- to allow garbage collection
  queue.first = first + 1
  return value
end

function Queue.isempty (queue)
    return queue.last < queue.first
end

function Queue.head (queue)
    return queue[queue.first]
end

function Queue.size (queue)
  return queue.last - queue.first + 1
end
```

## 7 *verifier.js*

```
var fs = require('fs');

var arguments = {};
var nThreads  = 0;
var filePath = '';
var objectVirify = {};


process.argv.forEach((val, index) => {
  arguments[index] = val;
});

nThreads = arguments['2'];
filePath = arguments['3'];
if(!filePath) {
  return console.log('enter with the path of the file');
}

fs.readFile(filePath, 'utf8', (err, data) => {
  if(err) {
    return console.log(err);
  }
  var lines = data.split('\n');
  lines.forEach((line, index) => {
    if(line.trim().length == 0){
      return;
```

```
    }
    var elements = line.split(' ');
    if(elements.length != 4) {
      console.log('line ' + index + ' has a problem of write');
      return;
    }
    var name = line.split(' is writing ')[0];
    var number = line.split(' is writing ')[1];
    if( name in objectVirify){
      objectVirify[name].array.push(number);
    } else {
      objectVirify[name] = {};
      objectVirify[name].array = [];
      objectVirify[name].array.push(number);
    }
  });

  for( var name in objectVirify) {
    var obj = objectVirify[name];
    var value = obj.array.reduce((previous, current, index, array) => {
      return parseInt(previous) + parseInt(current);
    });
    console.log(value, name);
    if(value != 5050){
      console.log(name + 'didn`t  write all number');
    }
  }
});
```