



**Department of Electrical,
Computer, & Biomedical Engineering**
Faculty of Engineering & Architectural Science

Course Title:	
Course Number:	
Semester/Year (e.g.F2016)	

Instructor:	
--------------------	--

<i>Assignment/Lab Number:</i>	
<i>Assignment/Lab Title:</i>	

<i>Submission Date:</i>	
<i>Due Date:</i>	

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: <http://www.ryerson.ca/senate/current/pol60.pdf>

COE 892 Lab 2 Report

Introduction

The primary objective of this lab 2 assignment was to take the work that was done in lab 1 part 2 and implement it in a client-server-like architectural style with the help of the grpc module. Grpc is used by client application to call methods on server applications directly a another machine as if it were a local object. Therefore, in the context of the lab 2 assignment, we are essentially creating an extension to the lab 1 assignment by turning said assignment into a distributed application with various services that are distributed amongst both a client program and a server program. The services of this program which I am asked to implement for the lab 2 assignment are similar to the services that I was asked to implement for the lab 1 assignment. These include the following:

1. Get map from ground control (e.g., “The 2D land array”)
2. Get a stream of commands (e.g., “RMLMMMMMDLMMRMD”)
3. Get a mine serial number
4. Let the server know if they have executed all commands successfully, or not (a mine might explode with a wrong series of commands)
5. Share a mine PIN with the server

The server program for the case of this assignment represents the ground control whose purpose is to feed instructions into the rover so that the rover can do its job of properly disarming mines on the land space. The client program on the other hand represents the rover object itself.

Part 1 – Creating the protocol buffer file

First and foremost, before I went ahead and began to code the server and client programs, I decided to work on a protocol or proto file first. Creating a proto file for this program is absolutely necessary for incorporating the grpc module into my code because grpc uses the proto file as a bridge that helps to connect the server and client programs together. It also used to establish the function definitions that are to be used by the server itself. Below is a screenshot of my proto file for the lab 2 assignment:

```
1  syntax = "proto3";
2
3  service GroundControl {
4      rpc getMap (mapRequest) returns (mapResponse) {}
5
6      rpc getCommands (commandsRequest) returns (commandsResponse) {}
7
8      rpc getSerialNum (mineSerialNumRequest) returns (mineSerialResponse) {}
9
10     rpc sharePin (pinRequest) returns (pinResponse) {}
11
12     rpc checkCompleted (checkCompletionRequest) returns (checkCompletionResponse) {}
13 }
```

Figure 1: RPC function definitions for server program

```
15  message mapRequest {
16      string mapFile = 1;
17  }
18
19  message mapResponse {
20      int32 rows = 1;
21      int32 cols = 2;
22      repeated string map = 3;
23  }
24
25  message commandsRequest{
26      int32 commandListNum = 1;
27  }
28  message commandsResponse {
29      repeated string commands = 1;
30  }
31
32  message mineSerialNumRequest {
33      int32 roverXPos = 1;
34      int32 roverYPos = 2;
35  }
36
```

Figure 2: Request and Response data fields of proto file

```

37  message mineSerialResponse {
38      |      string serialNum = 1;
39  }
40
41  message pinRequest {
42      |      int32 pin = 1;
43  }
44
45  message pinResponse {
46      |      string ack = 1;
47  }
48
49  message checkCompletionRequest {
50      |      string roverStatus= 1;
51  }
52
53  message checkCompletionResponse {
54      |      string ack = 1;
55  }
56
57
58

```

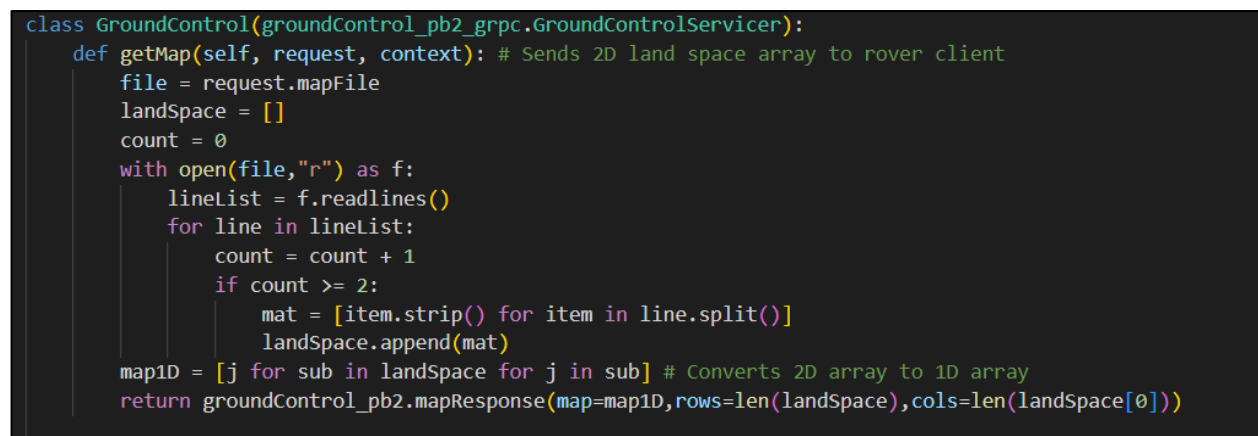
Figure 3: Request and Response data fields of proto file part 2

As you can see from the proto file screenshots above, the protocol buffer file contains the grpc function definitions for the functions that are to be used by the server as well as data field values that are used by the server to clarify what the return values for a particular server function should be, and they are also used by the client program to specify what the input values should be for the server methods. As you can see in the proto file screenshots above, all the request and response messages that were created in the proto file essentially reference a particular service that is to be implemented in the main client-server grpc program. For example, the mapRequest message and the corresponding mapResponse message is used by the ground control server to send the 2D land space map to the rover client based on the map.txt file (borrowed from lab 1 assignment) that is received by the server. Likewise, the command request message and the corresponding command response messages are used to send the list of sequential commands that is to be performed by the rover to the rover and other similar logic can also be applied for the other request-response

messages declared in the proto file as well. In order to use these data field values in both the client and server programs, I had to use the proto file to compile and convert a set of files called protobuf files with the following nomenclature: pb2, pb2_grpc and pb2.pyi. More information on how I accessed these proto file data fields can be noted in the following two part-sections.

Part 2: Server Creation

The purpose of the ground control server is to provide its rover client with the necessary information and tools that it needs to traverse the land space and disarm any mine objects that comes into contact with. This includes sending the 2D land space array to the rover client upon the reception of the map.txt file by the server, sending the list of sequential commands that is to be executed by each rover client, the sending of a mine object's corresponding serial number whenever the rover comes into contact and attempts to disarm a mine object on the land space, the retrieval of a mine object's pin and verifying if the rover has finished executing all of its commands. For the map sending function, it takes in the map file object as an input request, opens it, copies the contents of said file into a 2D array, and then flattens the 2D array into a 1D array and returns the 1D array with the map data to the rover client via a grpc map response message call. Below is a screenshot of how this was implemented.

A screenshot of a code editor showing the implementation of the getMap method in a Python class named GroundControl. The code is as follows:

```
class GroundControl(groundControl_pb2_grpc.GroundControlServicer):
    def getMap(self, request, context): # Sends 2D land space array to rover client
        file = request.mapFile
        landSpace = []
        count = 0
        with open(file, "r") as f:
            lineList = f.readlines()
            for line in lineList:
                count = count + 1
                if count >= 2:
                    mat = [item.strip() for item in line.split()]
                    landSpace.append(mat)
        map1D = [j for sub in landSpace for j in sub] # Converts 2D array to 1D array
        return groundControl_pb2.mapResponse(map=map1D, rows=len(landSpace), cols=len(landSpace[0]))
```

Figure 4: getMap method for server program

As you can see in the screenshot above, the way I invoked the map response message field from my ground control proto file is by referencing the map response class that is created in my protocol buffer file. This is because when the protocol buffer files for my proto file are created class instances of every single message variable within the proto file is created as well. Therefore, in order to reference the message variables from my proto file, I simply need to reference it from my ground control protocol buffer file. Furthermore, the reason why I had to flatten my 2D map array to a 1D map array is because proto files do not support 1D array data types. For my command list retrieval service, I implemented it incredibly similarly to how I implemented it in my lab 1 assignment. That is, I loaded the JSON file with the commands as a string variable, then extracted the command list string from that JSON string variable and finally I converted that command list string into a list of command characters and returned the command character list to the rover client. See screenshot below for implementation:

```

def getCommands(self, request, context): # Sends a list of commands to rover client
    url = "https://coe892.reev.dev/lab1/rover/" + str(request.commandListNum)
    fp = urllib.request.urlopen(url)
    byteArr = fp.read()
    dict_string = byteArr.decode("utf-8")
    fp.close()
    rover_dict = json.loads(dict_string)

    for key in rover_dict:
        if type(rover_dict[key]) is dict:
            command_dict = rover_dict[key]
            break

    for key in command_dict:
        if key == "moves":
            command_dict[str(request.commandListNum)] = command_dict[key]
            del command_dict[key]
            break

    new_CD = command_dict

    for key in new_CD:
        if int(key) == request.commandListNum:
            commandList = list(new_CD.values())
            break

    commands = ''.join(commandList)
    commandChars = list(commands)
    # Try the following command list to test mine disarming feature: ['M','M','M','D','M','D','M','M']
    return groundControl_pb2.commandsResponse(commands=commandChars)

```

Figure 5: Get commands method for server program.

When returning the list of command characters, I am calling the commands response class similar to how I did it with the get map method. Next I implemented the service for sending the serial number for a mine which the rover comes into contact with and to do that, first I loaded in the necessary information for the mine objects from a mines.txt file (borrowed from the lab 1 assignment) into a 1D array. This file contains the xy-coordinate location of a mine followed by its corresponding serial identification number. See below for a sample screenshot of its format:

```

≡ mines.txt
1 03 1cc83792-a35a-11ed-a8fc-0242ac120002
2 08 2ab41d3a-a35a-11ed-a8fc-0242ac120002
3 11 3aa5570e-a35a-11ed-a8fc-0242ac120002
4 25 5390e6fc-a35a-11ed-a8fc-0242ac120002
5 30 e2948ab0-a35b-11ed-a8fc-0242ac120002
6 40 fc441cb4-a35b-11ed-a8fc-0242ac120002
7 42 072e2af2-a35c-11ed-a8fc-0242ac120002
8 44 92d9912c-a35c-11ed-a8fc-0242ac120002
9 57 9cb4f5f6-a35c-11ed-a8fc-0242ac120002
10 59 a9fb1f60-a35c-11ed-a8fc-0242ac120002
11 60 83393dc4-a35e-11ed-a8fc-0242ac120002
12 61 8d012f24-a35e-11ed-a8fc-0242ac120002
13 62 93f37490-a35e-11ed-a8fc-0242ac120002
14 67 9c7b7bc6-a35e-11ed-a8fc-0242ac120002
15 68 cec5575e-a35f-11ed-a8fc-0242ac120002
16 69 d9e3d386-a35f-11ed-a8fc-0242ac120002

```

Figure 6: mines.txt file

Because of how my mines.txt file is formatted, the way that I implemented it is that first I took the array with the mine data and iterated through it with a for-loop. Then, I stored the xy location of the current mine's serial number (with respect to the current array index noted in the for-loop) in a variable called mineLoc. I then checked to see if the location stored in the mineLoc variable matched the current rover's current xy-position for the mine that it has found on the land space. If there was a match, then the mine that the rover has found at its current position on the map, has the serial number that is associated with that position (i.e if the rover matches with location 03, then the mine at location 03 has the serial number 1cc83792 and so on). I also used a counter variable that keeps track of the number of location matches that rover gets and if the loop reaches the last serial number in the array and is unable to get a match with the rover's current position, then the method will break out of the loop otherwise, it will proceed with the loop's next iteration. At the end of the loop, it simply returns the corresponding serial number of a mine object if it finds one. Otherwise, it returns an empty string. See screenshot below for the implementation.

```
def getSerialNum(self, request, context): # Returns the corresponding serial number for a mine to the rover client
    mineMatchcount = 0
    serial_num = ""
    mineData = []
    file = "mines.txt"
    with open(file, "r") as f:
        lineList = f.readlines()
        for line in lineList:
            mineData.append(line.strip())

    for j in range(len(mineData)):
        mineLoc = mineData[j][0] + mineData[j][1]
        if mineLoc == str(request.roverXPos) + str(request.roverYPos):
            mineMatchcount = mineMatchcount + 1
            serial_num = mineData[j][3:]
        elif mineMatchcount == 0 and j == len(mineData) - 1:
            break
        else:
            continue

    return groundControl_pb2.mineSerialResponse(serialNum=serial_num)
```

Figure 7: Get Serial Number Method

Next, I created a method which is essentially used to receive the PIN that is sent by the rover to indicate that the mine has been disarmed. Then, I created a method, that checks if the rover has executed all of the commands that are noted in its command list. If the rover has executed all of its commands in its command list, then the server will output a success message to the server console. Otherwise, it will print out a message to the server console saying that the rover has been destroyed by a mine bomb. Finally, I made one last method for the server whose primary purpose is to create an instance of the server that runs indefinitely until the user enters a termination command (i.e. Control + C) into the console that is running the server program. See the screenshot below for this implementation.

```

76     def sharePin(self, request, context): #Rover client shares valid pin number with the ground control server
77         pin = request
78         print(["Pin " + pin + " Received"])
79         return groundControl_pb2.pinResponse(ack="Pin received")
80
81     def checkCompleted(self, request, context):
82         print(request.roverStatus)
83         return groundControl_pb2.checkCompletionResponse(ack="Completion Acknowledged")
84
85 def serve(): #Primary function that is used to run the server
86     server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
87     groundControl_pb2_grpc.add_GroundControlServicer_to_server(GroundControl(),server)
88     server.add_insecure_port('[::]:50051')
89     server.start()
90     server.wait_for_termination()
91
92 if __name__ == '__main__':
93     logging.basicConfig()
94     serve()

```

Figure 8: Pin Sharing and Command Completion Methods

Part 3: Client Creation

Creating the client was a simple process as it only required the creation of four methods. First, I created a method that takes a 1D array and converts said 1D array into a 2D array of a certain row size and column size depending on the row and column parameters that are fed into the method. The reason why I was required to do this is because proto can accept and return only 1-dimensional type arrays and is strictly unable to support the input and output of n-dimensional arrays due to how the proto3 syntax is structured. Then I reused my direction changer method from lab 1 to use for my lab 2 assignment because I will need to call it when the rover has to traverse the 2D land space and look for mines. Then I also reused my mine digging procedure function from my lab 1 assignment because I will need to call it in my land space traversal whenever the rover comes into contact with a mine bomb on the land space. Lastly I created a land traversal function that is similar to the map traversal function that I created for my lab 1 assignment, with the key differences being that I have now added a few lines of code that properly integrate the module connections and features into the client program and help link the client program together with the server program. This includes creating a communication channel that links the client program to the server program and creating protocol buffer stub that is used to call all of the rpc methods that were declared in the proto file and properly developed in the server program. I also added an a boolean variable called roverAlive that is used to check if the rover has properly executed all of its commands. This variable is to be used in conjunction with the checkCompleted method that was developed in the server program. To clarify a bit more, the checkCompleted method sends a success or failure message string to the server which the server outputs to its console based on whether the rover is still alive or not. For example, if the boolean roverAlive is true then this means that the rover is still and that the client sends a success message to the server. Otherwise it sends a failure message to the server. Both these messages are printed out to the server's output console upon reception. The last new addition that I added to this new land space traversal function was that I added an input variable that asked the user what rover that they want to traverse the land space to find and disarm mine bombs. The client program also constantly asks the user to enter the rover number

into the client program until they enter something that is not a number into the program. This was made possible with the help of the `isDigit` method for string objects which checks if the string data entered by the user is an integer string. I did this in order to make it so the user can test the commands of all 10 rovers within a single execution of both the server and the client program. I also added some modifications to pre-existing code which helps the client program properly behave as it should with the outputs that are returned by all the methods that were created in the server program. See below for screenshots of all of the implementations of these methods.

```

client.py > to_matrix
1  from __future__ import print_function
2  import copy
3  import os
4  from hashlib import sha256
5  import grpc
6  import groundControl_pb2
7  import groundControl_pb2_grpc
8  import logging
9
10 def to_matrix(m,n,lst): #Converts 1D array into a 2D array of varying dimensions. m is rows n is cols
11     if m == n:
12         return [lst[i:i+n] for i in range(0, len(lst), n)]
13     else:
14         return [lst[x: x + n] for x in range(0, len(lst),n)][:m]
15
16 def directionChange(command,currentDirection):
17     # LEFT COMMAND CASES
18     if command == "L" and currentDirection == "S":
19         currentDirection = "E"
20     elif command == "L" and currentDirection == "E":
21         currentDirection = "N"
22     elif command == "L" and currentDirection == "N":
23         currentDirection = "W"
24     elif command == "L" and currentDirection == "W":
25         currentDirection = "S"
26     # RIGHT COMMAND CASES
27     elif command == "R" and currentDirection == "S":
28         currentDirection = "W"
29     elif command == "R" and currentDirection == "W":
30         currentDirection = "N"
31     elif command == "R" and currentDirection == "N":
32         currentDirection = "E"
33     elif command == "R" and currentDirection == "E":
34         currentDirection = "S"
35     return currentDirection

```

Figure 9: Direction Changer Method and 1D-to-2D array converter method - Client Program

```

37 def digProces(id): #Digging and disarming procedure for the rover 1 when rover 1 hits a mine and receives a 'D' command
38     validPin = False
39     pin = 0
40     while not validPin:
41         mineKey = str(pin) + id
42         hashedValue = sha256(mineKey.encode('utf-8')).hexdigest()
43         if hashedValue[0:6] == "000000":
44             validPin = True
45         else:
46             pin = pin + 1
47     results = [pin,validPin]
48     return results

```

Figure 10: Mine Digging Process Function Reused from Lab 1

```

50 def run():
51     channel = grpc.insecure_channel('localhost:50051')
52     stub = groundControl_pb2_grpc.GroundControlStub(channel)
53     response = stub.getMap(groundControl_pb2.mapRequest(mapFile="map.txt"))
54
55     landMat = to_matrix(response.rows,response.cols,response.map)
56
57     rovNum = input("Please enter the rover number. If a rover number is not entered, the client program will exit: ")
58     while rovNum.isdigit():
59         response2 = stub.getCommands(groundControl_pb2.commandsRequest(commandListNum=int(rovNum)))
60         commandMat = response2.commands
61         pathMat = copy.deepcopy(landMat)
62         pathRows = len(pathMat)
63         pathCols = len(pathMat[0])
64         currentFacing = "S"
65         x = 0
66         y = 0
67         roverAlive = True
68         for command in commandMat:
69             if x > pathRows - 1 or y > pathCols - 1 or x < 0 or y < 0: #Check if the rover goes out of bounds for the land space
70                 break
71             else:
72                 # Cases where command is M
73                 if currentFacing == "S" and command == "M":
74                     if pathMat[x][y] == "1": #Rover is on a mine
75                         pathMat[x][y] = "*"
76                         roverAlive = False
77                         break
78                     elif pathMat[x][y] == "0": # you can move and the rover is not over a mine
79                         pathMat[x][y] = "*"
80                     if x < pathRows - 1:
81                         x = x + 1
82                 elif currentFacing == "N" and command == "M":
83                     if pathMat[x][y] == "1": #Rover is on a mine
84                         pathMat[x][y] = "*"

```

Ln 10, Col 81 Spaces: 4 UTF-8 C

Figure 11: New Map Traversal function for Lab 2 - Client Program Part 1

```

84         pathMat[x][y] = "*"
85         roverAlive = False
86         break
87     elif pathMat[x][y] == "0": # you can move and the rover is not over a mine
88         pathMat[x][y] = "*"
89     if x > 0:
90         x = x - 1
91     elif currentFacing == "E" and command == "M":
92         if pathMat[x][y] == "1": #Rover is on a mine
93             pathMat[x][y] = "*"
94             roverAlive = False
95             break
96     elif pathMat[x][y] == "0": # you can move and the rover is not over a mine
97         pathMat[x][y] = "*"
98     if y < pathCols - 1:
99         y = y + 1
100     elif currentFacing == "W" and command == "M":
101         if pathMat[x][y] == "1": #Rover is on a mine
102             pathMat[x][y] = "*"
103             roverAlive = False
104             break
105     elif pathMat[x][y] == "0": # you can move and the rover is not over a mine
106         pathMat[x][y] = "*"
107     if y > 0:
108         y = y - 1
109     # If command is D and the rover does NOT hit a mine, write a * at that location
110     elif command == "D" and pathMat[x][y] == "0":
111         pathMat[x][y] = "*"
112     elif command == "D" and pathMat[x][y] == "1": # If command is D and the rover does hit a mine, initiate the digging process for th
113         response3 = stub.getSerialNum(groundControl_pb2.mineSerialNumRequest(roverXPos=x,roverYPos=y))
114         if response3.serialNum != "":
115             disarmResults = digProces(response3.serialNum)
116             if disarmResults[1] == True:
117                 print("Mine located at x-coordinate " + str(x) + " and y-coordinate " + str(y) + " has been disarmed successfully!")
118                 pathMat[x][y] = "#" # If mine is successfully disarmed place a # character at that location
119                 response4 = stub.sharePin(groundControl_pb2.pinRequest(pin=disarmResults[0]))
120                 elif response3.serialNum == "":

```

Figure 12: New Map Traversal function for Lab 2 - Client Program Part 2

```

120         elif response3.serialNum == "":
121             print("Mine cannot be disarmed. Unable to identify the mine's serial number.")
122             pathMat[x][y] = "*" #Error handling scenario where the function is unable to identify the mine's serial number
123         elif command == "L" or command == "R":
124             currentFacing = directionChange(command,currentFacing)
125
126     if os.path.exists("path_" + str(rovNum) + ".txt"):
127         os.remove("path_" + str(rovNum) + ".txt")
128
129     fileName = "path_" + str(rovNum) + ".txt"
130     roverPath = open("path_" + str(rovNum) + ".txt","x")
131     roverPath.close()
132     with open(fileName,"w") as pathFile:
133         for row in pathMat:
134             pathFile.write(' '.join([str(a) for a in row])+ '\n')
135
136     if roverAlive == True:
137         response5 = stub.checkCompleted(groundControl_pb2.checkCompletionRequest(roverStatus = "Rover " + str(rovNum) + " has completed its co
138     else:
139         response5 = stub.checkCompleted(groundControl_pb2.checkCompletionRequest(roverStatus = "Rover " + str(rovNum) + " has exploded!"))
140
141     rovNum = input("Please enter the rover number. If a rover number is not entered, the client program will exit: ")
142
143 if __name__ == "__main__":
144     logging.basicConfig()
145     run()

```

Figure 13: New Map Traversal function for Lab 2 - Client Program Part 3

If implemented correctly, the client-server program's combined output should resemble that of lab 1 part 2's sequential output.

Conclusion

As a conclusion for this lab 2 assignment, I have learned how to use the grpc programming module to link a client and server program together in order to create a simulated distributed application of sorts with a variety of distributed services.