Ryerson University

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

Course Title:	Distributed Cloud Computing
Course Number:	COE 892
Semester/Year	Winter 2023

Instructor:	Dr. Muhammad Jaseemuddin

Assignment/Lab Number:	Lab 3
Assignment/Lab Title:	Concurrency vs Parallelism

Submission Date:	March22nd, 2023
Due Date:	March22nd, 2023

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Sadig	Daniel	500894225	042	D.A

*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

Introduction

The objective of this lab is to utilize RabbitMq to simulate the communication between the rover server and client. The client will publish to the “Demine-Queue”, they will publish the rover serial number and rover number to the queue. The Deminer will act as our broker it will route the queue's message to the “Defused-Mines” queue. If a message is not routed to the server it will be held in the queue until the broker is started and routed to the appropriate queue. To complete this lab we will be building off the grpc commands and files we created in the last lab.

For this lab we are only implementing 3 of the 5 functions from the last lab, we are only using the GetMap, GetMoves, and the SerialNumber function. In, figure 0, we have the updated **proto** for lab 3. After, running the proto commands the subsequent files created by grpc where also updated.

```
syntax = "proto3";

package lab2;

service Bomb{
    //Unary
    rpc GetMap (GetMapRequest) returns (GetMapReply) {};

}

service RoverMoves{
    rpc GetMoves (GetRoverMoves) returns (ReturnRoverMoves) {};
}

service SerialNumber{
    rpc GetSerialNum (SerialNumReq) returns (SerialNumReply) {};
}
```

Set Up

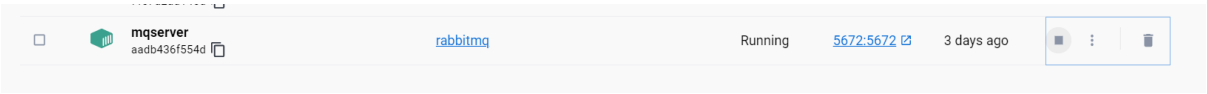

Figure 1: Docker Container (mqserver)

In order to create the queues necessary to complete the lab we needed to initialize RabitMq and dockerized containers. This container will allow us to hold data in out queue which can later be used by our client, server, or deminer

Client Side

```
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

channel.queue_declare(queue='Demine-Queue')

channel.basic_publish(exchange='',
                      routing_key='Demine-Queue',
                      body=f'Position: {ans[0]}, {ans[1]} id: {fileNum} serial: {serial_num} ')
print(" [x] Sent 'Hello World!'")

#cracked = crack(encoding, str(serial_num))
```

Figure 2: Client side

On the client side we will be sending the position of the rover, the rover id, and the serial number of the bomb. The code shown in, **figure 2,** is how we will establish a connection with the Demine Queue. For this lab when a rover hits a bomb we will invoke the RabitMq procedure call from above. In. **figure 3,** we added 4 bombs to the 'Demine-Queue'. This information will be used by the deminers.py (broker) file. In the output of the client side when we see [x] Sent Hello World, this is where we are adding our serial number and rover number to the queue.

```
Please enter rover Number
1
[[0, 0, 0, '2', 0, 0, 0, 0], ['1', 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0,
0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0]]
 [x] Sent 'Hello World!'
[0, 3]
Enter exit to exit the prog
Please enter rover Number
6
[[0, 0, 0, '2', 0, 0, 0, 0], ['1', 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0,
0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0]]
 [x] Sent 'Hello World!'
 [x] Sent 'Hello World!'
[6, 1]
Enter exit to exit the prog
Please enter rover Number
2
[[0, 0, 0, '2', 0, 0, 0, 0], ['1', 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0,
0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0]]
 [x] Sent 'Hello World!'
[2, 1]
Enter exit to exit the prog
Please enter rover Number
```

Figure 3: Client-side output

Deminer.py

On the deminer side there are 3 main functions that we will implement, read from the queue, defuse the bomb, and publish to a queue. in **figure 1,** you can see the implementation of the read from 'Demine-Queue'. We are invoking a similar method to call to that of the server side. On the deminer side, we will read from the queue, then output the number of messages remaining in the queue. Next, we will call the **callback** function, this splits the message we read from the queue. Then we call the **crack** method, this will get the pin needed to defuse the mine **figure 5**.

```
def main():
    connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
    channel = connection.channel()

    queue = channel.queue_declare(queue='Demine-Queue')

    def callback(ch, method, body):
        response = body.split()[-1]
        encoding = 'ascii'
        print(f' \n {body} and {response} \n')
        ans = crack1(encoding, str(response))
        send(ans)
        print(f" [x] Received {body}")


    channel, method, body = channel.basic_get(queue='Demine-Queue', auto_ack=True)
    print(f' \n There are {queue.method.message_count} messages in the queue \n')
    if(queue.method.message_count == 0):
        print("empty ")
        return "Error"
    callback(channel, method, body)
    #channel.start_consuming()

    print(' [*] Waiting for messages. To exit press CTRL+C')
    #channel.start_consuming()
```

Figure 4: Deminer side, read from 'Demine-Queue'

```
def crack1(encoding, serial_Num):
        print(encoding)

        def passwords(encoding):
            chars = [c.encode(encoding) for c in printable]
            for length in count(start=1):
                for pwd in product(chars, repeat=length):
                    yield b''.join(pwd)

        for pwd in passwords(encoding):
            input_string = serial_Num.encode(encoding) + pwd
            hash_result = hashlib.sha256(input_string).hexdigest()
            if hash_result.startswith("0"*6):
                print(hash_result)
                print(pwd.decode(encoding))
                return pwd.decode(encoding)
```

Figure 5: Deminer side, find the appropriate pin for the serial number

Once, we defuse the mine we call the **send** or **send2** function, this function will publish to the 'Defused-mine' queue. This will publish the serial number pin for the corresponding rover to the appropriate queue. Then the server will read the published message from the queue. Until the server reads the message, it will be held in the queue, once the server reads the message it will terminate the message in the queue. The implementation of this process can be seen in **figure 6**.

```python
#Send
def send(response):
    connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
    channel = connection.channel()

    channel.queue_declare(queue='Defused-Mines')

    channel.basic_publish(exchange='',
                          routing_key='Defused-Mines',
                          body=f'from 1: {response}')
    print(" [x] Sent from deminer 1")

    connection.close()
```

Figure 6: Deminer side, publish to 'Defused-Queue'

Output of the Deminer

When running the deminer we have to fist pick wich demier we have to use, if we type 1 we use deminer 1, if we type 2 we use deliminer 2. If we type 1 while delimiter 1 is still runing then the uer will be prompted with a message indicating that the user must wait till the process is complete. However, if they choose a deminer that is not busy then the process will run on deminer 2. This is made possible because we use threads in the code wich allows us to run the 2 deminers in parallel. The output of the code can be seen in **figure 7**.

```
Hello we have 2 deminers please select wich one you want to run !

Enter 1 for Deminer 1 and 2 for 21
enter 1 or 2 again
 There are 4 messages in the queue


 b'Position: 1, 0 id: 1 serial: 12 ' and b'12'

ascii
1
True
t1 is busy please wait
enter 1 or 2 again2
enter 1 or 2 againascii
```

Figure 7: Output of deminer code

Once the deminer reads the message from the client it will begin to call the crack function wich will decode the pin needed to defuse the bomb. Once the bomb is defused, the deminer will output the pin for the appropriate serial number. Then it wil send a function wich will invoke publish function wich will publish a new message to the queue. In **figure 8,** we see the return of the deminer 1 and deminer 2 from wich we invoked earlier, once they are

decoded the position, id, serial and pin are sent to the server. The results on the server can be seen in **figure 9.**

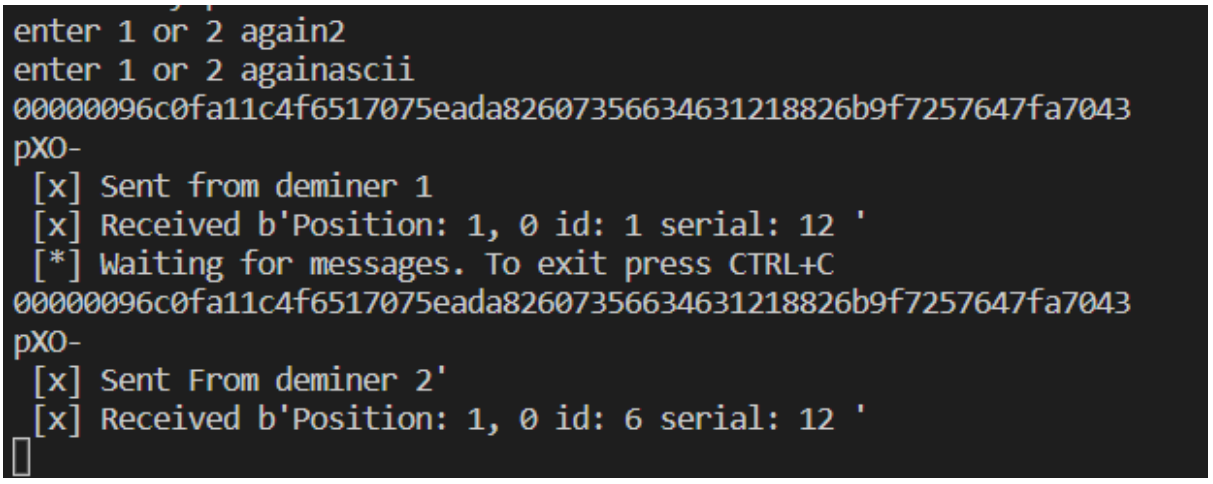

Figure 8: Output of deminer code once pin is decoded

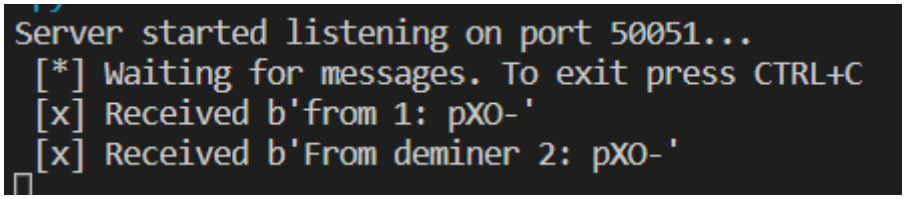

Figure 9: Result of the server reading the messages in the queue

Server Code

To update the server side we needed to add the read queue functionality to the server side. The challenge that came with this is the blocking call created by the RabbitMq connection. To overcome this issue I added 2 threads, one to run the grpc functions and one to run the RabbitMQ server, this helps us overcome the issues mentioned above. The read from queue function is similar to the one implemented in the deminer file. The update to the code can be seen in **figure 10**.

In **figure 11,** we can see what the server reads from the corresponding queue, as it is outputed to the terminal. Once it is read fom the queue it is outputed and deleted from the queu

```python
def main():
    connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
    channel = connection.channel()

    channel.queue_declare(queue='Defused-Mines')

    def callback(ch, method, properties, body):
        print(" [x] Received %r" % body)

    channel.basic_consume(queue='Defused-Mines', on_message_callback=callback, auto_ack=True)

    print(' [*] Waiting for messages. To exit press CTRL+C')
    channel.start_consuming()

if __name__ == '__main__':
    try:

        t1 = threading.Thread( target=main, args=())
        t2 = threading.Thread( target=serve, args=())

        t1.start()
        t2.start()

        t1.join()
        t2.join()
```

Figure 10: The updated code on the server side

```
Server started listening on port 50051...
 [*] Waiting for messages. To exit press CTRL+C
 [x] Received b'from 1: pXO-'
 [x] Received b'From deminer 2: pXO-'
 [x] Received b'from 1: b=v\r'
 [x] Received b'From deminer 2: pXO-'
```

Figure 11: Output of the serverside

Conclusion

This lab was a great way to better understand how RabbitMQ and dockerized containers operate. By successfully sending and receiving the defused pin from the client and server we where able to unserdatnd how the broker (Deminer.py) works.