COE 892 - Distributed Cloud Computing

Winter 2025

Dr. Khalid A. Hafeez

Lab 3 - RabbitMQ

Submission Date: March 7, 2025

Eric Muzzo

501019745

# Introduction

This lab builds on lab 2 by introducing RabbitMQ, a messaging broker used to send and receive messages between distributed applications. In simple terms, RabbitMQ is a message broker that accepts, stores and routes messages. Producers send messages to RabbitMQ and consumers receive these messages. In addition to producers and consumers, RabbitMQ uses channels and queues. A channel is a virtual connection inside a real TCP connection that connects an application to the RabbitMQ server. A queue is a buffer that stores messages while they wait to be consumed by a consumer. In this lab, we use two queues, 'Demine-Queue' which is where rovers send their tasks to be solved by deminers, and 'Defused-Mines' which is where deminers send their completed tasks to be received and printed by the Ground Control server.

The main difference between this lab and Lab 2 is that Rovers no longer handle the processing of demining a mine on the map. Instead, they fetch the serial number of the mine from the Ground Control Server via a remote procedure call, identical to Lab 2, and then publish a demining task to the 'Demine-Queue' queue. The Deminers are the consumers of the queue, and they are responsible for computing the mine pin. Upon finding the pin, they publish a message to the 'Defused-Mines' queue where the ground control server consumes them and prints them to the console.

Note that this report will not cover the implementation details regarding data structures, map processing or rover traversal, and gRPC since that was covered in labs 1 and 2. This report will focus on the messaging protocol to and from the RabbitMQ server.

# Program Design & Overview

For instructions on running the application, refer to the README.md within the submitted zip file or visit the source code repository.

## Deminers

A "Deminer" class is created to perform the operations necessary for finding the pin for a given serial number. This logic was directly cut and pasted from the Rover class methods mine() and hashKey() in lab 2, to the Deminer class. The Deminer class initializes the connection and

channel to the RabbitMQ server, as well as declaring the queues that it will be interacting with:

```
class Deminer:
    """An instance of a deminer object used for demining mines"""

    def __init__(self, id):
        self.id = id

        self.rabbit_connection = pika.BlockingConnection(pika.ConnectionParameters("localhost", 5672))
        self.rabbit_channel = self.rabbit_connection.channel()

        self.rabbit_channel.queue_declare(queue="Demine-Queue")
        self.rabbit_channel.queue_declare(queue="Defused-Mines")
```

I define a class method start() which begins consuming messages on the ‘Demine-Queue’ queue. This involves defining a callback function to execute when a message is consumed from the queue. The callback function simply extracts parameters from the request body, one of them being the serial number of the mine requested from a Rover, and computes the pin. From here, the deminer publishes a new message to the ‘Defused-Mines’ queue to indicate to ground control that it completed a task.

## Rover

**NOTE:** *The program is designed on the assumption that when a Rover encounters a mine and publishes a task for demining, it assumes that the deminer will solve the pin at some point and therefore sets the value of the cell to “EMPTY”. Since the lab manual did not specify otherwise, there is no waiting for the Deminer to actually solve the mine; the rover does not care and moves on.*

In lab 2, the Rover class resided within the models.py module. However I decided to move this to its own module (src/rovers.py) since it no longer represents a data structure but now is an active entity that communicates with other distributed applications. Almost all of the implementation is kept the same with a few exceptions.

The initializer function adds two additional parameters for the RabbitMQ connection and channel. It also declares the ‘Demine-Queue’ since it will need to publish tasks to this queue as it traverses the map.

A class method publish_demine_task() is defined which is responsible for generating the message payload that includes the mine location and serial number. This method then publishes the payload to the ‘Demine-Queue’ queue where a Deminer will now take care of the demining process.

The move() method removes the match case where command == ‘D’ since the server removes D commands from the command stream.

The run() method checks if the rover is currently on a mine. If it is, an RPC is made to ground control for the serial number of that mine. The publish_demine_task method is then called, passing the serial number as an argument.

## Server

The GroundControlService class from lab 2 is regenerated via the proto buffer files since the ReportStatus and ShareMinePin remote procedure calls are no longer needed. Thus, the GroundControlService class removes these 3 methods. In order to allow both the gRPC server and consuming of messages within the same python file, I defined two additional methods outside the GroundControlService class. subscribeToDefusedQueue() runs the lines of code to establish a connection and channel, as well as declare the queue 'Defused-Mines', to connect the server to the RabbitMQ server. This method subscribes the server to that queue so that it can print the defused mine pins to the console when it consumes them from the queue. Inside this method, I defined the callback function which is the simple logic that the server does (variable extraction and print statements) when a message arrives in the queue.
To run both the message consumption procedure and the gRPC server, in __main__ I use a separate thread that runs the subscribeToDefusedQueue() method. After I start that thread, I run the serve() method, which serves the gRPC services the same as lab 2.

# Conclusion

RabbitMQ provides a sophisticated messaging pipeline that allows producers to publish messages to an arbitrary service without worrying about the architecture on the receiving end. This allows components of distributed applications to operate independently, with mutual trust that the producers and consumers will do their job. It seems that RabbitMQ facilitates independent scalability of applications since RabbitMQ acts as a middleman between two applications; decoupling their dependency on one another. The lab was successfully implemented using the mentioned concepts to achieve task management of each component.