ABSTRACT

Title of Thesis:	Development of a Python-Driven Framework for Automation and Collaboration in Data Sharing
Degree Candidate:	Ethan Ty Cooper
Degree and Year:	Master of Science, 2024
Thesis Directed By:	David Heddle, Ph.D., Professor of Physics, Department of Physics, Computer Science, and Engineering

Scientific users are increasingly able to collaborate with other institutions thanks to advancements in technology. However, hurdles such as the lack of a robust and easy to implement framework in which users can perform their necessary data operations prevent efficient data sharing. A solution to this problem must be able to easily send, receive, and convert data amongst its users. It must also provide authentication measures and possess adequate system resources due to the typically large file sizes of scientific data. Ease of use should also be a priority, offering a comprehensive experience to all users regardless of technical knowledge. This research proposes a data-sharing framework utilizing the RabbitMQ message broker as well as the Python programming language. It is a branch of a previously developed Java-driven API which also implemented RabbitMQ. The main goal for this project is to develop a platform that heavily automates the data-sharing process.

DEVELOPMENT OF A PYTHON-DRIVEN FRAMEWORK FOR AUTOMATION AND COLLABORATION IN DATA SHARING

by

Ethan Ty Cooper

Thesis submitted to the Graduate Faculty of
Christopher Newport University in partial
fulfillment of the requirements
for the degree of
Master of Science
2024

Approved:

David Heddle, Chair _____

Edward Brash _____

William Phelps _____

DEDICATION

I would like to dedicate this thesis to all of my friends and family, who without whose love and support I could not have gotten this far.

ACKNOWLEDGEMENTS

I would like to extend my thanks to the members of this committee who provided valuable insight in developing this research

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I:
INTRODUCTION

Statement of Problem

Advancements in technology such as higher internet speeds, more robust computer systems, and improved cloud architectures grant scientific organizations and their respective user bases a greater ability to collect and analyze data. Despite these more localized advancements, collaboration between organizations is still hindered by multiple factors. Currently, the data-sharing process typically requires a large amount of user input and prior knowledge of a data-sharing platform on both the sending and receiving ends. Rather than sending messages directly to an intended recipient in a specific file format, they must also be aware of which formats are used by which specific organizations as well as middleman organizations who are able to convert data from one format to another.

Take for example the following scenario: We identify a sender (A), an intended recipient of A's data (B), and a middlemen institution (C). Under ideal circumstances, A wishes to share a .pdf file directly to B. However, B's institution utilizes .csv files rather than .pdf files. This prompts A to share their file with C, who is able to convert .pdf files into the requested .csv files, then forward the converted data to B. While this issue is less severe with smaller and/or local networks, it can quickly scale to become tedious and inefficient as factors such as the size of the data being transferred and the number of middle organizations required increase the time, resource, and human cost of performing data transfers.

This research is a continuation of an API framework established by Cassandra Villarreal and Andrew Nguyen. It seeks to simplify and automate the data-sharing process by providing a common platform in which users can upload, share, and convert data amongst themselves. This iteration of the research will see the development of a Python based script and associated shell application that will connect users to a remote RabbitMQ server and transfer files using MagicWormhole. These scripts and applications are to be designed with ease of access and comprehensiveness in mind, as well as utilizing open-source tools and

libraries wherever possible. The ResearchAPI framework communicates with an established central CloudAMQP server and uses RabbitMQ to send and receive messages from the users connected to it. In addition, MagicWormhole provides a method of secure file transfer and addresses the size limits of RabbitMQ messaging. The primary language used in this research will be Python and JSON.

We summarize and hopefully clarify the problem by pointing out that the use-case for this project, which is to create a collaborative scientific network, is very different from the use-case for large social networks. The latter scale to accommodate many anonymous users exchanging many messages possibly with a tight latency constraint and a relatively small payload. The messages use a small set of supported, standard formats. By contrast our use-case envisions a small number of users, known to one another as part of a scientific collaboration, exchanging large data sets (up to several GB) in arbitrary formats in which speed, while not ignored, is not a premium. The other difference is that while social networks are the "app", our scientific network seeks to make it simply for existing scientific models (typically in the form of complete, special-purpose desktop applications) to become "clients" on the network.

A final distinction. While there is a flavor of a service oriented architecture (SOA) to this project, it is not the manner that SOAs have been employed in physics applications to date. A physics SOA implies, at least to most people, decomposing a large applications and models into atomic functional components and redeploying each functional piece as a loosely-coupled service. Here we do not require large, complex applications to be redone. We will share the data (inputs and outputs) of large applications without requiring them to be broken into smaller pieces.

Assumptions

This research is being conducted under a number of assumptions which revolve around the user-base and capabilities of the application. The user-base for this application is expected

to remain relatively small with users who are already familiar with one another. While the initial ResearchAPI framework (i.e., the most recent version of this project) is designed to be modified by an institution in order to best fit their needs, users are expected to use the provided command line and GUI-driven files to interact with this version of the framework. Additionally, users are expected to at least be familiar with services such as RabbitMQ and MagicWormhole as well as the concepts of cloud computing and utilizing command line arguments. While not explicitly required as a result of the GUI-driven experience (the shell application provided by this work), it is nevertheless recommended that users have knowledge of either or both the Java and Python programming languages. This research and all future branches are expected to continue the usage of free and open-source tools, although we will acknowledge that it is likely that paid cloud computing resources may be used thanks to their availability and computing power. We will also be creating and implementing security measures in this version in order to ensure system safety, controlled scaling, and address the need for data privacy / confidentiality. If users take issue with these additional measures, they are able to customize either this implementation or the previous public Java framework.

Tools

As stated in previous iterations, this version of the framework will continue to implement open-source and industry vetted tools wherever possible. The main components of this framework (RabbitMQ, CloudAMQP, and the Python Language) are all free to download and use in contrast to paid services such as Amazon Web Services and Microsoft Azure. CloudAMQP runs off of smaller AWS servers that can be upgraded for an additional cost, but the current usage of a free server is sufficient for testing and development purposes. MagicWormhole was chosen as the file transfer tool since a large portion of the user base is expected to be familiar with it. The Pycharm IDE was the primary development environment used in this research due to its ability to manage packages and to be able to test both the shell application and the command line parser versions of the framework. Pycharm was chosen due to personal preference. This framework will continue the trend of

being available for usage on a Github repository. Github and its associated tool Git-Bash helped provide a storage medium with version control and easy collaboration. This helps us to keep track of the changes in our files and helps identify any errors or bugs found during development. [1]

Terminology

GENERAL TERMINOLOGY

- Java: A high-level object-oriented programming language
- Python: An interpreted, high-level programming language best known for its simplicity and readability. It supports object-oriented and functional programming
- Metadata: Data that provides information about other data such as format, structure, source, location, etc. In respect to this project, RabbitMQ messages will follow a standard metadata format that specifies facets such as the sender and receiver of a message and what the formats of their requested and available data formats are.
- Shell: A command-line interface (CLI) provided by an operating system that allows for a user to interact with said system through commands and scripts. This research will see a Python-based shell application for user interaction with the framework.
- API: Application Programming Interfaces are a set of rules, protocols, and tools that allow different software applications to communicate and interact with one another. The previous iteration of this project was a Java-based API.
- JAR: Java Archives are a commonly-used file format for packaging Java software into a single archive file. The Java version of this framework was distributed as research.jar and contained the full library of necessary files.
- Pycharm: A dedicated Python integration development environment (IDE), provides users with necessary tools for code editing, debugging, testing, and version control.

- Pika: Pika is a dedicated Python library for communicating with RabbitMQ. Provides a high-level API for creating producers, consumers, and handling message queues. Its primary purpose is to simplify the process of integrating Python with RabbitMQ.[2][**<empty citation>**]

- Eclipse: An open-source IDE compatible with multiple languages, notable C++ and Java. Offers tools for editing, debugging, version control, and plugin support. The previous iteration of this research was editing and debugged inside of Eclipse before being translated into Python in Pycharm.

- Github: A web-based platform for hosting, sharing, and collaborating on software projects using Git version control. Provides features such as code repositories, issue tracking, forking, pull requests, and code reviews. Can be both public and private, which allows for easily controllable distribution.

- Git Bash: A command-line interface and shell environment for Windows. Provides users with a Unix-like experience and enables them to interact with Git repositories, execute shell commands/scripts, and perform various systems tasks.

RABBITMQ-SPECIFIC TERMINOLOGY

- CloudAMQP: A managed message queue service that is provided as part of a cloud computing platform. It is based on the RabbitMQ service and reliable message queuing functionality.

- RabbitMQ: An open-source message broker software that implements the Advanced Message Queuing Protocol (AMQP). RabbitMQ enables communication between distributed applications by routing, queuing, and delivering messages asynchronously.[3][4][**<empty cita**

- Producer: An entity in a messaging system that generates and sends messages to a broker or queue. Communication is initiated by publishing messages to a specified destination within a messaging system.

- Consumer: An entity within a messaging system that receives and processes messages from a broker or queue. Consumers subscribe to specific destinations within the messaging system and actively retrieve messages for processing and consumption
- Queue: A data structure used within a messaging system to temporarily store messages sent by producers until they are consumed by a consumer. Queues follow the First-In-First-Out (FIFO) principle, ensuring that messages are processed in the order that they arrive.
- Translator: A translator in the context of a messaging system is responsible for converting messages from one format, protocol, or representation to another. Translators allow for interoperability between different systems and applications.
- UUID: Acronym for Universally Unique Identifier, typically a 128-bit value represented as a hexadecimal string that is globally unique. This is useful for identifying resources, objects, and entities in distributed systems without central coordination.
- Cloud Computing: A model of computer architecture in which computer resources are accessible on-demand over the internet
- Exchanges: In the context of RabbitMQ, exchanges are message routing mechanisms that determine how a message is delivered to a queue.

(The following terms are the specific type of message exchanges in RabbitMQ)

- Routing Key: A piece of metadata attached to a message by the producer, used by a message broker to determine how a message should be routed to queues. Typically string values that are matched against queue binding keys.
- Binding Key: A criteria or pattern that is defined by a queue when it binds to an exchange in RabbitMQ. The binding key specifies the conditions under which messages should be routed from the exchange to the given queue. The exchange uses this binding key in conjunction with the message's routing key in order to determine the delivery location of a message. Binding keys are also typically string values.

- Direct Exchange: Messages are delivered to queues based on an exact match between the message's routing key and the binding key of the queue. Each message is routed to one queue, which is determined by its routing key.
- Fanout Exchange: Messages are broadcast to all of the queues bound to it and then delivered to every queue regardless of the routing and binding key.
- Topic Exchange: Messages are routed to queues based on pattern matching of the message's routing key.
- Headers Exchange: Type of exchange where messages are routed to queues based on header attributes and values. Messages are routed to queues with header attributes that fulfill certain criteria, which allows for flexible message routing.

CHAPTER II:
SYSTEM DESIGN AND METHODOLOGY

RabbitMQ

The central point of this research is to establish a common platform in which scientific users are able to automatically share and convert data while also being able to collaborate with other users and organizations. The Python programs utilizing RabbitMQ will be able to connect to the central server, which will then allow for them to perform their requested actions. Message brokers allow for greater amount of freedom on a user's end since they will "hold" the data until it is retrieved, whereas a message queue may expire or timeout.

While message brokers can quickly scale as more users use their services, this issue is avoided in this research due to our small testing pool. The previous iteration of this research chose to use RabbitMQ as the message broker system due to its low-cost and open-source nature, as well as extensive official documentation and community support. It is also known as being easy to use and implement with multiple programming languages such as Java, Python, and C++.

RabbitMQ's basic structure is defined by a producer, a consumer, and a queue, all of which have been defined in the terminology section. Messages that are sent and received in RabbitMQ utilize a specified message protocol. The main difference between the various message protocols is the format, which affects both exchanges between applications and processing. RabbitMQ supports a large number of messaging protocols such as various versions of AMQP and MQTT, however this iteration continues with the previous usage of AMQP 0-9-1 [1][5]. The basic structure of a RabbitMQ implementation is demonstrated below in Figure 1 [6][7].

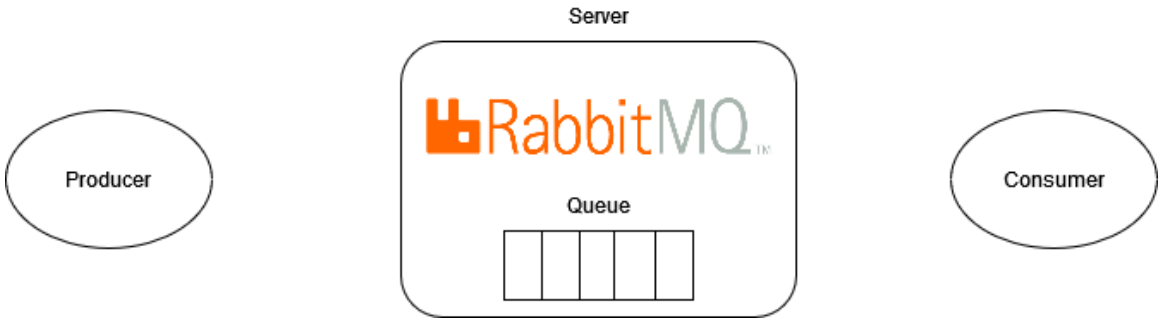

Figure 1: A Basic RabbitMQ Model

Exchanges are buffers in a RabbitMQ implementation which help to determine the location of where a message must be delivered. As defined in the terminology, three such exchanges are the fanout, topic, and direct exchange. As a refresher, fanout exchanges send messages to all known queues, direct exchanges match a queue's binding key that matches a message's routing key, and a topic exchange functions similarly to the direct exchange with the exception of allowing for non-exact matching. Visualizations of these exchanges are demonstrated below in Figures 2, 3, and 4.

Figure 2: Direct Exchange Model

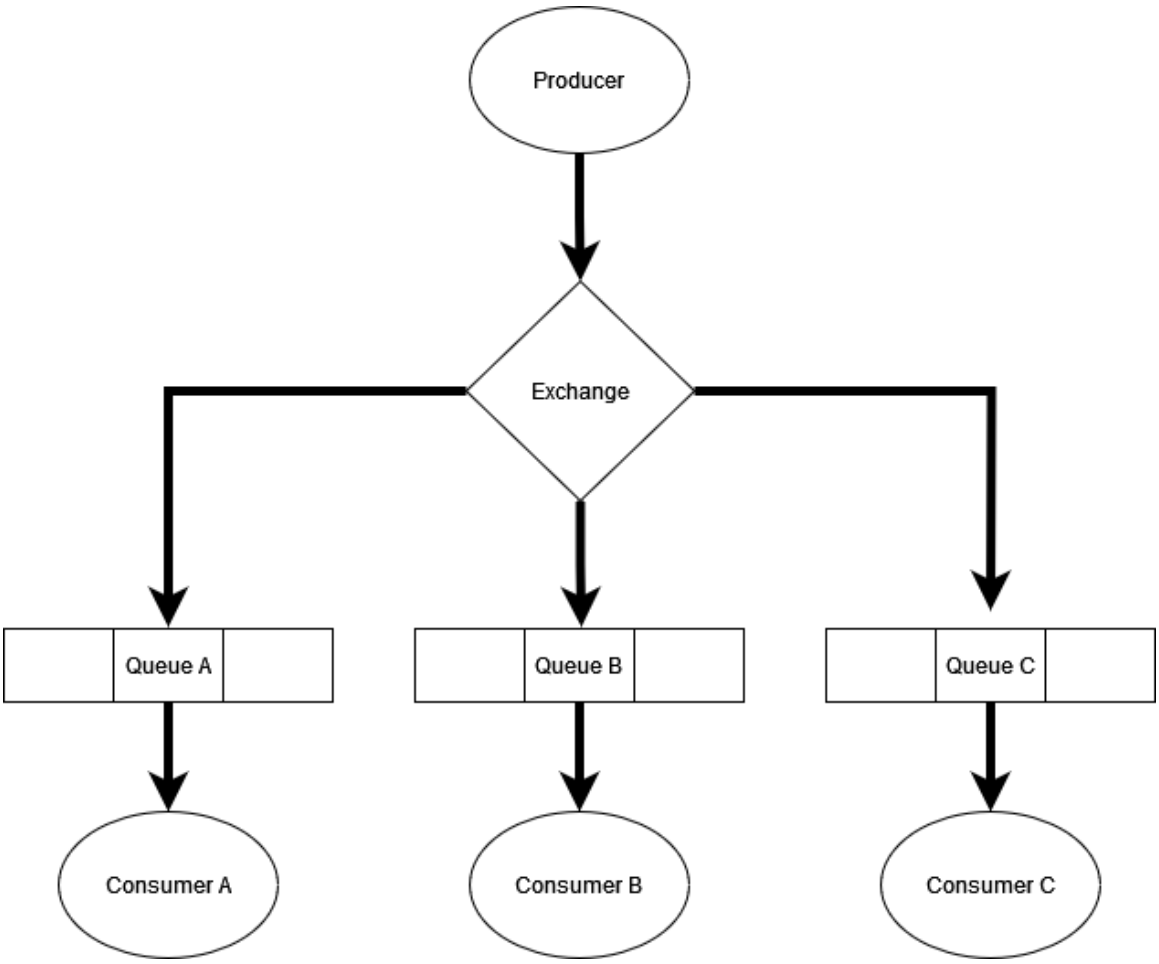

Figure 3: Fanout Exchange Model

Figure 4: Topic Exchange Model

Our framework makes large usage of the direct exchange as users will be sharing data between each other. Topic exchanges will be useful for groups of collaborators. Fanout exchanges will be useful for informing users of administrative information.

RabbitMQ does have the drawback of having a set size limit for the messages it is able to send. A message has a proposed maximum size of 2GB, but recommends that messages are kept below 128MB [1]. Since the data we can expect users to share being in the realm of giga- and terabytes worth of information, RabbitMQ alone will not be able to handle all

of the data-sharing requests we wish to have, requiring the use of an additional process or program to handle such large file sizes and datasets.

MagicWormHole

MagicWormhole is both a command line tool and software library that provides users an avenue to share and transfer data. Unlike RabbitMQ, there are no predetermined size limits for the data that is being transferred, with transfer speed instead being determined by internet speed and the size of the data.

MagicWormhole functions by creating an exchange between two devices through the usage of a server. The server itself is only a medium to accept and forward messages rather than acting as another contributing entity in the exchange. The sender and the receiver connect to the rendezvous server and are sent the first message known as the wormhole code, which is generated by MagicWormhole itself. The wormhole code is a unique human-readable identifier consisting of the server ID and two random words. The code is used to create a secure session key. Once the key has been created, the two machines exchange IP addresses. Messages sent between the two computers are encrypted with the session key and are sent via the TCP protocol ports. File data is sent via an encrypted record pipe generated by a MagicWormhole object.

MagicWormhole is accessed through the computer terminal application and utilizes commands preceded with the word ”wormhole”. Connections are initiated with the ”wormhole send” command, with the user also attaching the file name at the end. The ”wormhole receive” command must be entered by the intended recipient of the message, upon which they must enter the secure session key in order to download the sent data. The framework automatically handles these steps for the users.

```
% wormhole send README.md
Sending 7924 byte file named 'README.md'
On the other computer, please run: wormhole receive
Wormhole code is: 7-crossover-clockwork

Sending (<-10.0.1.43:58988)..
100%|=========================| 7.92K/7.92K [00:00<00:00, 6.02MB/s]
File sent.. waiting for confirmation
Confirmation received. Transfer complete.
```

Figure 5: Example of MagicWormhole's "send" command

```
% wormhole receive
Enter receive wormhole code: 7-crossover-clockwork
Receiving file (7924 bytes) into: README.md
ok? (y/n): y
Receiving (->tcp:10.0.1.43:58986)..
100%|=========================| 7.92K/7.92K [00:00<00:00, 120KB/s]
Received file written to README.md
```

Figure 6: Example of MagicWormhole's "receive" command

CloudAMQP

CloudAMQP is a cloud platform offering multiple RabbitMQ servers through providers such as Microsoft Azure and Amazon Web Services. Several pricing tiers with respective data limits and spaces are offered. This research uses a free AWS cloud server due to our relatively low testing and research needs. Larger scaled implementations of this framework can easily switch to another server, either on CloudAMQP, another platform, or a self-hosted cloud architecture system.

CloudAMQP provides several advantages and benefits to its users such as specializing in managing RabbitMQ instances, providing extensive monitoring and administration options, and allows for ease of use/maintenance. A CloudAMQP server is hosted on a unique web page, which will then be used in the program files as a global variable. This means that every implementation of this framework will need to modify this variable in order to fit their respective site.

For this research, we created a CloudAMQP server on the official website, which offers users multiple tiers with differing resources and performances. Our server is hosted by Amazon Web Services under the free tier of CloudAMQP and allows for 20 connections, 150 queues with a length of 10,000 each, and 1 million messages. We do not expect to exceed these limits as this is a strict testing environment, but future and larger-scale implementations must take these constraints into account when designing their cloud platforms.

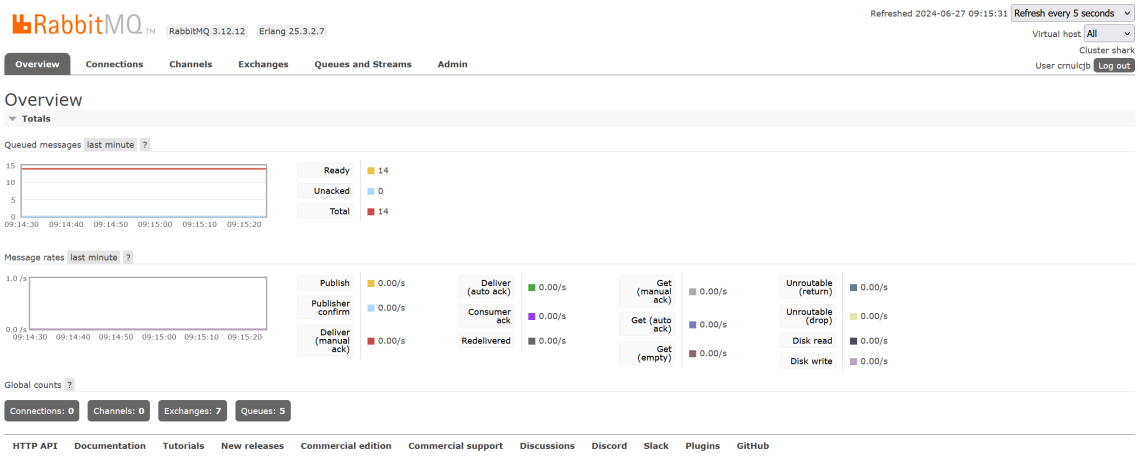

Figure 7: The administration window of the RabbitMQ server

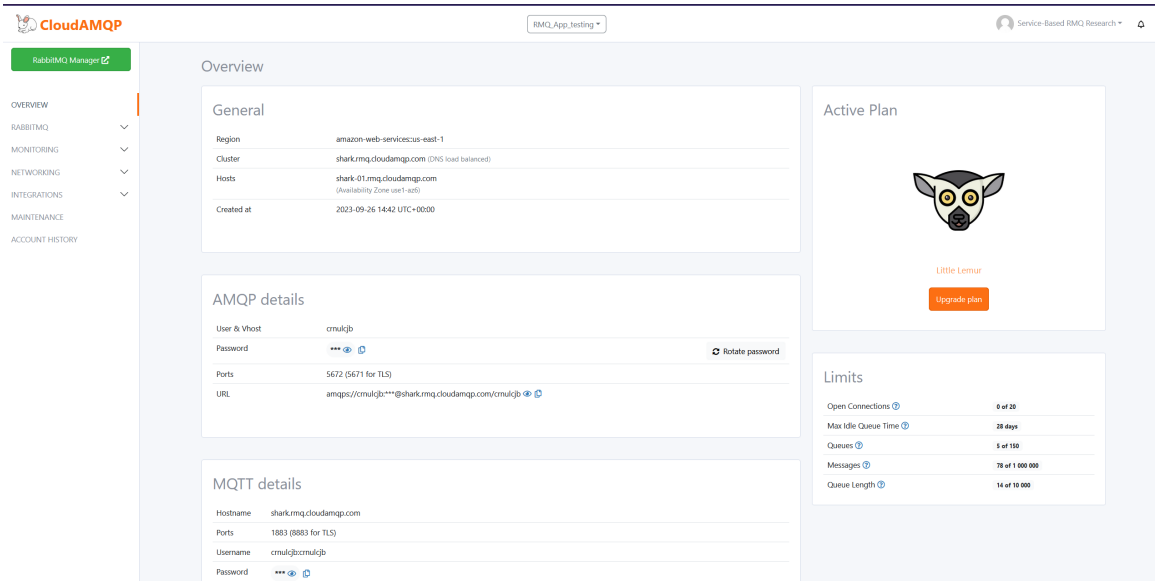

Figure 8: Overview of CloudAMQP server

Java

Initially, it was planned to further improve the Andrew Nguyen's Java implementation as well as create a new Python implementation for the RabbitMQ framework. The entire Java implementation of the framework is publicly available on Github, containing all the program files independently as well as a cumulative JAR file, which is designed to be easily implemented into a user's framework. The proposed improvements of the Java system would have included general code refinement, additional testing files, and implementing security measures. These changes would have addressed some of the previous research's concerns with future implementations of this system. These changes were not fully implemented in the Java version due to resource and time constraints. The most notable use of Java in this iteration was to provide a structural reference for the Python implementation. The JAR file and its components were converted into Python code, which served as the initial version of this framework. Further refinements were made in order to better utilize and adapt to Python's unique constraints and rules.[1]

Python

Python was chosen as the main programming language for this research due to personal preference and familiarity. While it was noted that in the previous research that Python was also initially considered, the platform shifted to Java. This changed due scientific users testing the app's own familiarity and preferences. We have decided to shift back to Python due to finding alternatives for the various libraries and packages utilized by the Java ResearchAPI, and in an attempt to streamline the data-sharing process even further. This is accomplished through both the command line and Python shell application versions of this framework. Python's more natural syntax and existing libraries for scientific computing such as pandas and matplotlib also make it a natural choice. Python has its own dedicated RabbitMQ-handling library named pika and also natively supports the MagicWormhole protocol we implement to transfer files.[8]

CHAPTER III:
IMPLEMENTATION

Metadata

The messages sent and received within the framework contain metadata that provides important identifying information such as file content and size. The previous Java API created by Andrew Nguyen defined a specified JSON format for this metadata that proceeds as follows:

1. user_id (str): The ID of the user.
2. message_type (str): The type of the message.
3. message_id (str): The ID of the message.
4. data (list): The data of the message.
5. data_request_formats (list): The request formats of the message.
6. data_convert_formats (dict): The convert formats of the message.
7. origin_message_id (str): The origin message ID of the message.
8. source_user_id (str): The source user ID of the message.
9. timestamp (str): The timestamp of the message.

This metadata is generated through user interaction with the framework. The User ID is created and added to messages upon account creation, the data formats are manually added, and others are automatically generated by the framework. There are currently 4 message types are defined by the framework. These types allow for easy identification for messages without looking through each field in the metadata:

1. announce_data: Indicates that a user has generated data and is able to share the data field in the metadata format.

2. can_translate: Indicates that a user can convert the announced data to a different format that is specified in the data_convert_formats metadata field.

3. request_data: Indicates that a sender wants to receive a data file or is requested the data to be converted to another format.

4. sent_data: Indicates that the magic-wormhole process has been initiated and is waiting on the message receiver to complete the process. This process is automatically completed by the API.

```
{
    "metadata": {
        "user_id": "0b1a9d59-8aca-4979-8884-e7e55a9f5b8c",
        "message_type": "announce_data",
        "message_id": "1620ddcf-1704-429e-8032-b18c7ac415fe",
        "data": [
            {
                "filename": "test-data.csv",
                "filesize": 10
            }, {
                "filename": "test-scatter-data.csv",
                "filesize": 6832
            }
        ],
        "data_convert_formats": [
            {
                "original_format": "csv",
                "destination_formats": [
                    "json",
                    "pdf"
                ]
            }, {
                "original_format": "json",
                "destination_formats": [
                    "csv"
                ]
            }
        ],
        "data_request_formats": [
            "csv",
            "json"
        ],
        "origin_message_id": "a9ce7936-a7f1-4b0b-86b0-7905050a0ee2",
        "source_user_id": "47101c5c-7f79-4c02-9816-e07c982346c5",
        "time_stamp": "2022-10-20T18:29:47Z"
    },
    "content": "ExampleJSONofmessage."
}
```


Figure 9: An example of a JSON message generated by the framework

Python

The previous code base was primarily written in the Java programming language due to both its popularity and similarity to other C-based languages, which were noted to be common knowledge amongst the user base testing the Java framework.

Python was chosen for its multiple and easy to implement libraries that allow for performing data science operations as well as creating a general user interface. The Python code base was initially created by crafting rough translations from the original Java project. Fine-tuning this translation was the bulk of the coding work for this research due to the numerous improvements we sought to make and the structural and terminology differences between the two languages.

Furthermore, we sought to simplify the code base to make it more readable and comprehensible to users who wish to customize their implementations of the framework. The components of the Python framework and how they interact with the command line and the GUI shell applications are detailed below.

API

The API directory contains ResearchAPI.py which is the main class for the framework. It handles user interactions, file management, and messaging. It implements the User.py class as well as the Message and MagicWormhole classes from the Message directory in order to handle these numerous tasks. The add_want_formats, add_file, and add_convert_format methods are used to add desired file formats, add files, and add file conversion formats, respectively. The connect method is used to establish a connection to a RabbitMQ server, and the start_listening method is used to start listening for messages from the RabbitMQ server. ResearchAPI.py also contains the MessageThread class, which is a subclass of threading.Thread and is used to process messages in multiple threads, making our application more resource-efficient. Threads are started with the run method and the process method is used to process messages.

```python
import os
import threading
from pathlib import Path
from ..user import User
from ..message import Message, ProcessMessage, MagicWormhole
from ..my_logging import Log
import pika

class ResearchAPI:
    def __init__(self, log_type, log_level):
        Log.setOutput(log_type, log_level)
        self.user = User()
        self.connection = None
        self.received_filename = None

    def add_want_formats(self, *want_formats):  self.user.add_want(want_formats)
    def add_file(self, filepath):
        file = Path(filepath)
        if not file.exists():
            Log.error(f"Filepath does not exist for: '{filepath}'", "add_file")
            return
        self.user.add_filepaths(filepath)
        if self.connection is None:
            self.connect()
            if self.connection is None:
                return
        announce_data = Message(self.user.get_user_id(), Constants.ANNOUNCE_MESSAGE)
        for path in self.user.get_filepaths():
            announce_data.add_file_path(str(path))
        announce_data.add_content("I have data.")
        self.connection.basic_publish(exchange='', routing_key='announce', body=announce_data.to_json())

    def add_convert_format(self, original_format, destination_format):
        self.user.add_convert(original_format, destination_format)

    def connect(self, uri=""):
        self.connection = pika.BlockingConnection(pika.URLParameters(uri))
        self.channel = self.connection.channel()

    def start_listening(self):
        if self.connection is None or self.channel is None:
            return
        else:
            MessageThread(self).start()
            Log.other(" [*] Begin listening to RabbitMQ server.")

    def get_received_file(self):
        cwd = os.getcwd()
        received_files_dir = Path( *args: cwd, "received-files")

        if self.received_filename and received_files_dir.exists():
            file = Path( *args: received_files_dir, self.received_filename)
            self.received_filename = None
            return [str(file), file.name.split(".")[1]]
        return [None, None]

class MessageThread(threading.Thread):...
class Constants:...
```

Figure 10: ResearchAPI.py

Constants

The Constants.py file in this directory stores constant values used throughout the project, although it is not instantiated itself. This is guarded by a ValueError being raised if an instantiation is attempted in a user's implementation. These constants are used by RabbitPy.py and other files in the project to ensure consistency. The Constants class contains several string constants that represent different types of messages and metadata keys. For example, ANNOUNCE_MESSAGE is a constant for the string "announce_data", which is a message type for announcing data. Similarly, USER_ID is a constant for the string "user_id", which is a key for user ID in metadata. The Constants class also contains the RABBITMQ_URI constant, which stores the link to the RabbitMQ server and is used to access it. This constant is unique to this implementation and must be manually changed by a user or administrator to their respective RabbitMQ server.

```python
class Constants:
    """..."""
    # Message Types
    ANNOUNCE_MESSAGE = "announce_data"  # Message type for announcing data
    CAN_TRANSLATE = "can_translate"  # Message type for indicating translation capability
    REQUEST_DATA = "request_data"  # Message type for requesting data
    SENT_DATA = "sent_data"  # Message type for indicating sent data

    # Metadata Keys
    METADATA = "metadata"  # Key for metadata
    USER_ID = "user_id"  # Key for user ID
    MESSAGE_ID = "message_id"  # Key for message ID
    MESSAGE_TYPE = "message_type"  # Key for message type
    METADATA_FILEDATA = "data"  # Key for file data within metadata
    DATA_CONVERT_FORMATS = "data_convert_formats"  # Key for data conversion formats
    DATA_REQUEST_FORMATS = "data_request_formats"  # Key for data request formats
    ORIGIN_MESSAGE_ID = "origin_message_id"  # Key for original message ID
    SOURCE_USER_ID = "source_user_id"  # Key for source user ID
    TIMESTAMP = "time_stamp"  # Key for timestamp
    ORIGINAL_FORMAT = "original_format"  # Key for original format
    DESTINATION_FORMATS = "destination_formats"  # Key for destination formats
    FILENAME = "filename"  # Key for filename
    FILESIZE = "filesize"  # Key for filesize
    CONTENT = "content"  # Key for content

    ALLOWED_FORMATS = ['.pdf', '.csv', '.txt', '.json','.jpg', '.png',
                       '.jpeg', '.gif', '.bmp', '.tiff', '.svg']

    FORMAT_CONVERSIONS = [
        'csv_to_pdf','pdf_to_csv','csv_to_json','text_to_csv',
        'json_to_csv','csv_to_text','pdf_to_text','text_to_pdf']

    RABBITMQ_URI = "amqps://crnulcjb:jTi5qkc_4BJQy-J4fmMk6CEJn1_phN3x@shark.rmq.cloudamqp.com/crnulcjb"

    # jTi5qkc_4BJQy-J4fmMk6CEJn1_phN3x
    def __init__(self):
        """..."""
        raise ValueError("Constants class should not be instantiated.")
```

Figure 11: Constants.py

Example

This directory contains the message.json file, which establishes the standardized format we are trying to model our user data towards. As users build their profiles and send messages, these values will mostly be automatically assigned to a message. Additionally, the example directory contains 6 test files consisting of 2 .csvs, 2 .pdfs, and 2 .txts. These files are meant to help demonstrate message sending/processing as well as format conversion inside of the framework.

Message

1.) Executive.py: Designed to execute commands in a separate threads and handle communication with a RabbitMQ Server. The Executive class has several instance variables, including done, process, running, connection, user_id, filepath, origin_message_id, request_user_id, and request_message. These variables are used to manage the state of the class and store information necessary for executing commands and communicating with the RabbitMQ server. The execute method is used to execute a command in a separate thread. It creates a temporary file script with the command, then uses the subprocess.Popen function to execute the script. The output and error streams of the process are read in a separate thread, and any output is logged. If the output contains a specific string, a message is sent to the RabbitMQ server. The send_message method is used to send a message to the RabbitMQ server. It creates a Message object with the necessary information and sends it using the direct method of the RabbitMQConnection object.

```python
import os
import subprocess
import threading
import tempfile
from pathlib import Path
from ..message import Message
from ..rmq import RabbitMQConnection
from ..my_logging import Log

class Executive:
    def __init__(self):...

    def set_cwd(self, cwd):     self._cwd = cwd
    def set_connection(self, connection):   self.connection = connection
    def set_user_id(self, user_id):     self.user_id = user_id
    def set_filepath(self, filepath):   self.filepath = filepath

    def set_required_message_content(self, message):
        self.origin_message_id = message.get_origin_message_id()
        self.request_user_id = message.get_sender_id()

    def set_request_message(self, message):     self.request_message = message

    def send_message(self, command):
        send_data = Message(self.user_id, "sent_data")
        send_data.add_file_path(self.filepath)
        send_data.add_origin_message_id(self.origin_message_id)
        send_data.add_source_user_id(self.user_id)
        send_data.add_content(command)
        self.connection.direct(send_data, self.request_user_id)

    def request_data_again(self):
        origin_sender_id = self.request_message.get_source_user_id()
        self.connection.direct(self.request_message, origin_sender_id)

    def temp_file_script(self, command):
        file = tempfile.NamedTemporaryFile(prefix="bCNU", delete=False)
        with open(file.name, "w") as script_file:
            script_file.write("#!/bin/bash\n")
            if self._cwd is not None:
                script_file.write(f'cd {self._cwd}\n')
            script_file.write(f"{command}\n")
        return file

    def execute(self, command):
        file = self.temp_file_script(command)
        if file is None:
            return
        process = subprocess.Popen( args: ["bash", file.name], stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True,
                                    shell=False)
        std_out_reader = process.stdout
        std_err_reader = process.stderr
        class_name = self.__class__.__name__

        def runnable():
            process.wait()
            self.done = True
```

Figure 12: Executive.py

```
    def reader():
        while not self.done:
            line = std_out_reader.readline()
            if line:
                Log.debug(line, class_name, command)
                if "wormhole receive" in line:
                    self.send_message(line)
                if "ERROR" in line and "receive" in command:
                    self.request_data_again()
            else:
                threading.Event().wait(0.05)
        std_out_reader.close()
        std_err_reader.close()
        os.remove(file.name)

    self.running = threading.Thread(target=runnable)
    self.running.start()
    threading.Thread(target=reader).start()

def get_running_thread(self):
    return self.running

@staticmethod
def execute_static(command, dir=None):
    executive = Executive()
    if dir and dir.exists() and dir.is_dir():
        executive.set_cwd(dir)
    executive.execute(command)
    return executive.get_running_thread()

@staticmethod
def execute_static_with_connection(command, dir, connection, request_message):
    executive = Executive()
    if dir and dir.exists() and dir.is_dir():
        executive.set_cwd(dir)
    executive.set_request_message(request_message)
    executive.set_connection(connection)
    executive.execute(command)
    return executive.get_running_thread()

@staticmethod
def execute_static_with_params(command, dir, connection, user_id, message, filepath):
    executive = Executive()
    if dir and dir.exists() and dir.is_dir():
        executive.set_cwd(dir)
    executive.set_required_message_content(message)
    executive.set_filepath(filepath)
    executive.set_user_id(user_id)
    executive.set_connection(connection)
    executive.execute(command)
```

Figure 13: Executive.py (Part 2)

2.) FileData: Represents a file, specifically its name and size. The FileData class has two instance variables, file name and file size. These are initialized in the __init__ method, which

takes the file name and file size as arguments. The class provides getter methods for both of these instance variables. The get_file_name method returns the name of the file, and the get_file_size method returns the size of the file. Finally, the FileData class overrides the __str__ method to provide a string representation of the object. This representation includes both the file name and file size.

```
class FileData:

    def __init__(self, filename, filesize):
        self.filename = filename
        self.filesize = filesize

    def get_file_name(self):
        return self.filename

    def get_file_size(self):
        return self.filesize

    def __str__(self):
        return f"({self.filename}, {self.filesize})"
```

Figure 14: FileData.py

3.) MagicWormhole: Wormhole.py is designed to handle file transfers using the MagicWormhole protocol. As stated in previous sections, MagicWormhole allows us to send files securely and directly between two machines with no predetermined size limit. The check_filename method is used to ensure that the filename of a file being received is unique.

If a file with the same name already exists in the receiving directory, the method appends a number to the filename to make it unique. The receive method is used to receive a file. It constructs a command to receive a file using Magic Wormhole, checks if the receiving directory exists and creates it if necessary, ensures the filename is unique, and then executes

the command in a separate thread. The send method is used to send a file. It constructs a command to send a file using Magic Wormhole and then executes the command in a separate thread.

The execute method is used to execute a command in a separate thread. It creates a new thread and starts it, with the thread's target function being a function that executes the command using the subprocess.Popen function. The ReceiveObj class is a nested class within the Wormhole class. It represents a file that is being received. It stores the new filename, the thread executing the receive command, the ID of the user sending the file, and the original filename.

```python
import os
import subprocess
from threading import Thread
from typing import List
from message import Message

class Wormhole:
    cwd = os.getcwd()

    def check_filename(existing_filenames: List[str], filename: str) -> str:
        if filename in existing_filenames:
            while filename in existing_filenames:
                name, file_format = filename.rsplit( sep: ".",  maxsplit: 1)
                num = name.rsplit("-", 1)[-1]
                filename = f"{name.rsplit('-', 1)[0]}-{int(num) + 1 if num.isdigit() else 2}.{file_format}"
        return filename

    def receive(connection, request_message, command, filename, sender_id):
        command_builder = [command, "--accept-file"]
        received_dir = os.path.join(Wormhole.cwd, "received-files")
        if not os.path.exists(received_dir):
            os.mkdir(received_dir)
        filename = Wormhole.check_filename(os.listdir(received_dir), filename)
        command_builder.append(f"-o {filename}")
        running = Wormhole.execute(command_builder, received_dir, connection, request_message)
        return Wormhole.ReceiveObj(filename, running, sender_id, filename)

    def send(connection, user_id, message, filepath):
        command = ["wormhole", "send", filepath]
        Wormhole.execute(command, Wormhole.cwd, connection, user_id, message, filepath)

    def execute(command, working_directory, connection=None, user_id=None, message=None, filepath=None):
        def run_thread():
            process = subprocess.Popen(command, cwd=working_directory, shell=True, stdout=subprocess.PIPE,
                                       stderr=subprocess.PIPE, text=True)
            process.communicate()
            if connection is not None and user_id is not None and message is not None and filepath is not None:
                connection.direct(message, user_id, f"File '{os.path.basename(filepath)}' sent successfully.")

        thread = Thread(target=run_thread)
        thread.start()
        return thread

    class ReceiveObj:
        def __init__(self, new_filename, running, source_user_id, original_filename):
            self.new_filename = new_filename
            self.running = running
            self.source_user_id = source_user_id
            self.original_filename = original_filename

        def get_new_filename(self):          return self.new_filename
        def get_running_thread(self):        return self.running
        def get_source_user_id(self):        return self.source_user_id
        def get_original_filename(self):     return self.original_filename
        def get_file_format(self):           return self.new_filename.split(".")[1]
```

Figure 15: MagicWormhole.py

4.) Message: This file is designed to replicate the structure of a typical message and

contains the necessary attributes and methods for file conversion and messaging. The two main attributes of this class are metadata and content. The metadata attribute is an instance of the Metadata class while the content attribute is a string containing the content of the message. The class contains methods which allow for manipulating a message's metadata such as the add_file_path and add_request_formats methods. The to_json method converts our message into a JSON string, which is then passed into the from_json class method in order to create a Message object from it to allow for easy serialization and deserialization. Additionally, the Message class also contains getter methods for the various metadata components.

```python
import json
from pathlib import Path
from message import Metadata  # Assuming you have a Metadata class in a separate file


class Message:

    def __init__(self, user_id, message_type):
        self.metadata = Metadata(user_id, message_type)
        self.content = None

    def add_file_path(self, file_path):
        try:
            valid_path = Path(file_path)
            self.metadata.set_data(valid_path)
        except (FileNotFoundError, IsADirectoryError, IOError) as e:
            print(e)

    def request_file(self, file_data):      self.metadata.set_data(file_data)

    def add_request_formats(self, *wants):      self.metadata.set_data_request_formats(wants)

    def add_request_formats_from_list(self, wants):
        for want in wants:
            self.metadata.set_data_request_formats(want)

    def add_convert_format(self, original_format, destination_format):
        self.metadata.set_data_convert_formats(original_format, destination_format)

    def add_origin_message_id(self, message_id):
        self.metadata.set_origin_message_id(message_id)

    def add_source_user_id(self, source_user_id):
        self.metadata.set_source_user_id(source_user_id)

    def add_content(self, content):     self.content = content

    def to_json(self):
        message = {
            'metadata': self.metadata.to_json(),
            'content': self.content
        }
        return json.dumps(message)
```

Figure 16: Message.py

```
@classmethod
def from_json(cls, message):
    root = json.loads(message)
    metadata_json = root['metadata']
    metadata = Metadata.from_json(metadata_json)
    content = root['content']
    return cls(metadata, content)

def get_sender_id(self):        return self.metadata.user_id

def get_message_type(self):     return self.metadata.message_type

def get_message_id(self):       return self.metadata.message_id

def get_file_data(self):        return self.metadata.data

def get_request_formats(self):      return self.metadata.data_request_formats

def get_convert_formats(self):      return self.metadata.data_convert_formats

def get_origin_message_id(self):    return self.metadata.origin_message_id

def get_source_user_id(self):   return self.metadata.source_user_id

def __str__(self):          return f"metadata = {self.metadata}, content = {self.content}\n

def get_metadata(self):     return self.metadata

def get_content(self):      return self.content
```

Figure 17: Message.py (Part 2)

5.) Metadata: Metadata.py is designed to represent the metadata contained inside of a message. It consists of various attributes such as user ID, message ID, message type, content/data, request formats, and convert formats. The file contains the appropriate getters and setters for these components and also contains the to/from_json methods. A __str__ method provides us with an easily readable string version of the metadata.

```python
import json
import uuid
from datetime import datetime
from message import FileData  # Assuming you have a FileData class in a separat
from pathlib import Path


class Metadata:
    def __init__(self, user_id, message_type):
        self.user_id = user_id
        self.message_type = message_type
        self.message_id = str(uuid.uuid4())
        self.data = []
        self.data_request_formats = []
        self.data_convert_formats = {}
        self.origin_message_id = ""
        self.source_user_id = ""
        self.timestamp = datetime.now().isoformat()

    def set_data(self, file_path):
        try:
            filename = file_path.name
            filesize = str(file_path.stat().st_size)
            self.data.append(FileData(filename, filesize))
        except (IOError, FileNotFoundError) as e:
            print(e)

    def set_data_request_formats(self, *wants):
        self.data_request_formats.extend(wants)

    def set_data_convert_formats(self, original, destination):
        self.data_convert_formats.setdefault(original, []).append(destination)

    def set_origin_message_id(self, origin_message_id):

        self.origin_message_id = origin_message_id

    def set_source_user_id(self, source_user_id):
        self.source_user_id = source_user_id
```

Figure 18: Metadata.py

```
def to_json(self):
    metadata_json = {
        'user_id': self.user_id,
        'message_type': self.message_type,
        'message_id': self.message_id,
        'metadata_filedata': self.data_to_json(),
        'data_request_formats': self.request_formats_to_json(),
        'data_convert_formats': self.convert_formats_to_json(),
        'origin_message_id': self.origin_message_id,
        'source_user_id': self.source_user_id,
        'timestamp': self.timestamp
    }
    return metadata_json

def data_to_json(self):
    return [{'filename': file.get_file_name(), 'filesize': file.get_file_size()} for file in self.data]

def convert_formats_to_json(self):
    return [{'original_format': original, 'destination_formats': destinations} for original, destinations in
            self.data_convert_formats.items()]

def request_formats_to_json(self):
    return self.data_request_formats

@classmethod
def from_json(cls, metadata_json_obj):
    metadata = cls(metadata_json_obj['user_id'], metadata_json_obj['message_type'])
    metadata.message_id = metadata_json_obj['message_id']
    metadata.data = [FileData(file['filename'], file['filesize']) for file in
                     metadata_json_obj['metadata_filedata']]
    metadata.data_request_formats = metadata_json_obj['data_request_formats']
    metadata.data_convert_formats = {entry['original_format']: entry['destination_formats'] for entry in
                                     metadata_json_obj['data_convert_formats']}
    metadata.origin_message_id = metadata_json_obj['origin_message_id']
    metadata.source_user_id = metadata_json_obj['source_user_id']
    metadata.timestamp = metadata_json_obj['timestamp']
    return metadata

def __str__(self):
    return f"user_id: {self.user_id}, message_id: {self.message_id}, message_type: {self.message_type}\n" \
           f"data: {self.data}, data_convert_formats: {self.data_convert_formats}, " \
           f"data_request_formats: {self.data_request_formats}\n" \
           f"timestamp: {self.timestamp}, origin_message_id: {self.origin_message_id}, " \
           f"source_user_id: {self.source_user_id}"
```

Figure 19: Metadata.py (Part 2)

6.) ProcessMessage: This class is designed to process the messages that are received from a RabbitMQ server. It imports and utilizes the variables and methods from the files of the User and Message directories via attributes such as user, message, sender_id, message_type, message_id, user_id, filepaths, want_formats, convert_formats, and connection. The process method serves as the main method of the class. Its intended function is to process a received message, log it, and then add it to the list of a user's received messages. Depending on

the message type, it may request a file path and verify if a user wishes to receive data but does not have it, and also check if a user is able to convert data using their convert_format attribute. If a message's type is "SENT_DATA", a file reception using the Wormhole.receive is initiated.

The has_requested_data method checks if a user has requested data, then sends it if so using the Wormhole.send method. The wants_and_does_not_have_data and can_convert_data methods are used to check if the user wants data and does not have it, or if the user can convert data, respectively. Depending on the message type, these methods may call other methods such as want_data or convert_data_announcement. The want_data method checks if the user wants data and requests it if they do. It creates a new Message object with the request and sends it using the connection.direct method. The convert_data_announcement method announces that the user can convert data. It creates a new Message object with the announcement and sends it using the connection.announce method. The get_filepath method is used to get the file path of a file with a name that matches one of the file names in the message's file data.

```python
import os
import json
from typing import List, Dict, Union, Tuple
from .MagicWormhole import Wormhole
from ..message import MagicWormhole, Message, FileData
from ..rmq import RabbitMQConnection
from ..user import User
from ..constants import Constants
from ..my_logging import Log


class ProcessMessage:
    def __init__(self, user: User, connection: RabbitMQConnection, message: str) -> None:
        self.user = user
        root = json.loads(message)
        metadata = root[Constants.METADATA]
        if metadata[Constants.USER_ID] != user.get_user_id():
            self.message = Message(message)
            self.sender_id = self.message.get_sender_id()
            self.message_type = self.message.get_message_type()
            self.message_id = self.message.get_message_id()
            self.user_id = self.user.get_user_id()
            self.filepaths = self.user.get_filepaths()
            self.want_formats = self.user.get_want_formats()
            self.convert_formats = self.user.get_convert_formats()
            self.connection = connection

    def process(self) -> Union[Wormhole.ReceiveObj, None]:
        if self.message is not None:
            Log.received(f" [x] Received {self.message}")
            self.user.add_received_message(self.message_id, self.message)
            requested_filepath = self.get_filepath()
            if requested_filepath is not None:
                self.has_requested_data(requested_filepath)
            if not self.want_formats and requested_filepath is None:
                self.wants_and_does_not_have_data()
            if self.convert_formats:
                self.can_convert_data()
            if self.message_type == Constants.SENT_DATA:
                filename = self.message.get_file_data()[0].get_file_name()
                return Wormhole.receive(self.connection, self.user.get_request_message(), self.message.get_content(),
                                        filename, self.sender_id)
        return None

    def has_requested_data(self, filepath: str) -> None:
        if self.message_type == Constants.REQUEST_DATA:
            Wormhole.send(self.connection, self.user_id, self.message, filepath)

    def wants_and_does_not_have_data(self) -> None:
        if self.message_type == Constants.ANNOUNCE_MESSAGE:
            self.want_data(False)

    def can_convert_data(self) -> None:
        if self.message_type == Constants.ANNOUNCE_MESSAGE:
            self.convert_data_announcement()
```

Figure 20: ProcessMessage.py

```
    def want_data(self, for_convert: bool) -> None:
        request_want_formats = self.want_formats
        data = self.message.get_file_data()
        request_message_id = self.message_id
        origin_sender_id = self.sender_id

        for file_data in data:
            filename = file_data.get_file_name()
            file_format = filename.split(".")[1]

            already_requested = filename in self.user.get_files_requested(
                origin_sender_id) if self.user.get_files_requested(origin_sender_id) is not None else False

            if file_format in request_want_formats and not already_requested:
                request_message = Message(self.user_id, Constants.REQUEST_DATA)
                request_message.add_request_formats(request_want_formats)
                request_message.request_file(file_data)
                request_message.add_origin_message_id(request_message_id)
                request_message.add_source_user_id(origin_sender_id)
                request_message.add_content(f"Requesting file '{filename}'")
                self.connection.direct(request_message, origin_sender_id)
                self.user.add_file_request(origin_sender_id, filename)
                self.user.add_request_message(request_message)
                break

    def convert_data_announcement(self) -> None:
        announced_data = self.message.get_file_data()
        request_data = []
        convertable_formats = {}

        for file_data in announced_data:
            file_format = file_data.get_file_name().split(".")[1]
            if file_format in self.convert_formats:
                request_data.append(file_data)
                convertable_formats[file_format] = self.convert_formats[file_format]

        request_message = Message(self.user_id, Constants.CAN_TRANSLATE)

        for file_data in request_data:
            request_message.request_file(file_data)

        for original_format, dest_formats in convertable_formats.items():
            for dest_format in dest_formats:
                request_message.add_convert_format(original_format, dest_format)

        request_message.add_origin_message_id(self.message_id)
        request_message.add_source_user_id(self.sender_id)
        request_message.add_content(f"I can convert the data from {convertable_formats}")
        self.connection.announce(request_message)

    def get_filepath(self) -> Union[str, None]:
        for path in self.filepaths:
            filename = os.path.basename(path)
            for file_data in self.message.get_file_data():
                if file_data.get_file_name() == filename:
                    return path
        return None
```

Figure 21: ProcessMessage.py (Part 2)

Log

The Log.py file in this directory is used for logging events and errors in the system through the usage of Python's logging module. This module provides a simple interface for setting up and using a system logger. This is useful for debugging and tracking the system's activities. Both the shell application and RabbitPy.py can use this logging functionality. The set_output method is used to set up the logger. It takes two arguments: output_type and log_level. The output_type determines whether the logs will be outputted to the console or to a file. The log_level sets the level of the logger. The method creates a handler based on the output_type and sets its level to the log_level. It then adds the handler to the logger. The log_message method is used to log a message. It takes two required arguments: message and level, and an optional argument: method_name. Depending on the level, it logs the message as an info, warning, or debug message. If the level is "warning", it also includes the method_name in the log message. The get_log_file_name method is used to get the name of the log file. It creates a directory named "output-logs" in the current working directory if it doesn't exist, and then returns the path of the log file in this directory. The name of the log file is the current date and time.

```python
import logging
import os
from datetime import datetime

class Log:
    LOG_CLASS = __name__
    logger = logging.getLogger(LOG_CLASS)

    @staticmethod
    def set_output(output_type, log_level):
        level = getattr(logging, log_level.upper(), logging.NOTSET)
        Log.logger.setLevel(level)
        handler = logging.StreamHandler() if output_type.lower() == "console" else logging.FileHandler(
            Log.get_log_file_name())
        handler.setLevel(level)
        Log.logger.addHandler(handler)
        Log.logger.propagate = False

    @staticmethod
    def log_message(message, level, method_name=None):
        if level == "info":
            Log.logger.info(message)
        elif level == "warning":
            Log.logger.warning(f"{Log.LOG_CLASS}.{method_name} - {message}")
        elif level == "debug":
            Log.logger.debug(message)

    @staticmethod
    def get_log_file_name():
        log_dir = os.path.join(os.getcwd(), "output-logs")
        os.makedirs(log_dir, exist_ok=True)
        return os.path.join(log_dir, datetime.now().strftime("%Y-%m-%d;%H:%M:%S.log"))
```

Figure 22: Log.py

RabbitMQ

The RabbitMQConnection class handles connection and messaging operations in a RabbitMQ server. The RabbitMQConnection class has several attributes including user, connection, channel, and queue_name. These attributes represent the user of the connection, the connection to the RabbitMQ server, the channel of the connection, and the name of the queue, respectively. The class provides several methods for messaging operations. For example, the announce method sends a message to the ”announce” routing key, and the direct method sends a message to a specified user's routing key. The get_channel and

get_queue_name methods are used to retrieve the channel of the connection and the name of the queue, respectively.

```python
import pika
from constants import Constants
from message import Message
from user import User


class RabbitMQConnection:
    EXCHANGE_NAME = "research"
    EXCHANGE_TYPE = "direct"
    ANNOUNCE_ROUTING_KEY = "announce"

    def __init__(self, user: User, uri: str) -> None:
        self.user = user
        self.connection = pika.BlockingConnection(pika.URLParameters(uri))
        self.channel = self.connection.channel()
        self.queue_name = self.channel.queue_declare(queue="", exclusive=True).method.queue

        self.channel.exchange_declare(exchange=self.EXCHANGE_NAME, exchange_type=self.EXCHANGE_TYPE)
        self.channel.queue_bind(exchange=self.EXCHANGE_NAME, queue=self.queue_name,
                                routing_key=self.ANNOUNCE_ROUTING_KEY)
        self.channel.queue_bind(exchange=self.EXCHANGE_NAME, queue=self.queue_name,
                                routing_key=self.user.get_user_id())

    def announce(self, message: Message) -> None:
        self.channel.basic_publish(exchange=self.EXCHANGE_NAME, routing_key=self.ANNOUNCE_ROUTING_KEY,
                                   body=message.to_json())

    def direct(self, message: Message, user_id: str) -> None:
        self.channel.basic_publish(exchange=self.EXCHANGE_NAME, routing_key=user_id, body=message.to_json())

    def get_channel(self) -> pika.channel:
        return self.channel

    def get_queue_name(self) -> str:
        return self.queue_name

    def list_queues(self):
        channel = self.connection.channel()
        return [queue.method.queue for queue in channel.queue_declare(queue='', passive=True)]

    def list_exchanges(self):
        channel = self.connection.channel()
        return [exchange.exchange for exchange in channel.exchange_declare(exchange='', passive=True)]
```

Figure 23: RabbitMQ.py

Shell Application

This directory contains various files related to the shell application, including File_Function.py for file conversions and user_list.py for storing user credentials. These files are used by RabbitPy.py to manage the user's interactions with the system, such as converting files and managing user accounts. The first version of this framework, a rough demo of the GUI-driven application was developed in this directory and boasted a few rudimentary functions such as user registration and login and provided a basis for the structure of further versions. In the current version of the project, the most relevant classes in this directory are File_Function and user_list. File_Function contains multiple methods for file conversion between various common formats such as the aforementioned csv, pdf, and txt files. user_list contains all of the current registered users and their hashed passwords.

User

The User class represents a user the various attributes and methods that are needed for performing messaging and file conversion operations.Such attributes include userID, want, convert, filepaths, receivedMessages, filesRequested, translationsRequested, and requestMessage. These attributes represent the user's ID, the formats the user wants, the formats the user can convert, the file paths of the user's files, the messages received by the user, the files requested by the user, the translations requested by the user, and the current request message of the user, respectively.

The class provides several methods for manipulating and retrieving these attributes. For example, the add_want_format method adds a format to the user's want formats, and the has_want_format method checks if a format is in the user's want formats. The get_destination_formats method retrieves the destination formats for a given original format from the user's convert formats.The get_files_requested method retrieves the files requested by a given source user from the user's files requested. The get_translation_format_requests method retrieves the

translation format requests for a given filename from the user's translations requested.

```python
from dataclasses import dataclass, field
from pathlib import Path
from typing import Dict, List
from uuid import uuid4
from message import Message

@dataclass
class User:
    userID: str = field(default_factory=lambda: str(uuid4()))
    _want_formats: List[str] = field(default_factory=list)
    _convert_formats: List[str] = field(default_factory=list)
    filepaths: List[Path] = field(default_factory=list)
    receivedMessages: Dict[str, Message] = field(default_factory=dict)
    filesRequested: Dict[str, List[str]] = field(default_factory=dict)
    translationsRequested: Dict[str, List[str]] = field(default_factory=dict)
    requestMessage: Message = None
    ALLOWED_FORMATS = ['.pdf', '.csv', '.txt', '.json','.jpg', '.png', '.jpeg', '.gif', '.bmp', '.tiff', '.svg']
    FORMAT_CONVERSIONS = [
        'csv_to_pdf','pdf_to_csv','csv_to_json','text_to_csv','json_to_csv','csv_to_text','pdf_to_text','text_to_pdf']

    def __init__(self, user_id):
        self.user_id = user_id
        self._want_formats = []
        self._convert_formats = []

    def add_want_format(self, format):
        if format not in self.ALLOWED_FORMATS:
            raise ValueError(f"Invalid format. Allowed formats are {', '.join(self.ALLOWED_FORMATS)}")
        self._want_formats.append(format)

    def remove_want_format(self, format):
        if format in self._want_formats:    self._want_formats.remove(format)
        else:   raise ValueError(f"Format {format} is not in the want_formats list.")

    def add_convert_format(self, source_format, destination_format):
        if source_format not in self.ALLOWED_FORMATS or destination_format not in self.ALLOWED_FORMATS:
            raise ValueError(
                f"Invalid conversion. Formats must consist of the following:{', '.join(self.ALLOWED_FORMATS)}")
        self._convert_formats.append(f"{source_format} to {destination_format}")

    def has_want_format(self, format):  return format in self._want_formats
    def want_formats(self) -> List[str]:    return self._want_formats
    def want_formats(self, value):  self._want_formats = value
    def convert_formats(self) -> List[str]: return self._convert_formats
    def convert_formats(self, value):   self._convert_formats = value
    def get_destination_formats(self, original_format) -> List[str]:
        return self.convert_formats.get(original_format, [])
    def all_filepaths(self) -> List[Path]:  return self.filepaths
    def all_messages(self) -> Dict[str, Message]:   return self.receivedMessages
    def get_message(self, message_id) -> Message:   return self.receivedMessages.get(message_id)
    def user_id(self) -> str:   return self.userID
    def user_id(self, value):
        self.userID = value
    def get_files_requested(self, source_user_id) -> List[str]:
        return self.filesRequested.get(source_user_id, [])
    def current_request_message(self) -> Message:   return self.requestMessage
    def get_translation_format_requests(self, filename) -> List[str]:
        return self.translationsRequested.get(filename, [])
```

Figure 24: User.py

RabbitPy

The RabbitPy.py file is the center point for the entire framework and utilizes all of the other files and directories in order to provide a user interface through the Python console. In addition to importing and calling methods from the other classes in the project, RabbitPy also defines various commands and actions a user is able to select in order to perform their requested operations such as building a user profile, adding their preferred and available formats, and even performing basic file type conversions through the application. Each command has its own function and helper method which are called in the main function, mirroring the previous Java-based implementation. A detailed example of interacting with the RabbitPy file is detailed in the following Framework section.

CHAPTER IV:
FRAMEWORK

Changes from ResearchAPI

This research differs from previous implementations due to providing a physical application for the data-sharing framework through the usage of RabbitPy and the corresponding shell application. The shell application utilizes all of the same code as RabbitPy, but creates a more visually comprehensive and easy-to-use GUI driven experience with Python's tkinter library. The documentation for these files is listed above in the implementation section. As mentioned previously in this paper, the reason for there being two distinct methods of access for this framework is to address discrepancies amongst the user base and its familiarity with command line arguments.

CHAPTER V:
FRAMEWORK EXAMPLE

Documentation

The implementation of the data-sharing framework is visualized through the various code snippets and figures found throughout this project as well as the entire framework being available on the previously mentioned public Github repository. A fair bit of setup is required before utilizing this framework, as a user is expected to have downloaded RabbitMQ / Erlang onto their computer and is aware of the link to the CloudAMQP server, which would be provided by the administrator in a professional setting. The previous Java implementation's API also includes a Java Web formatted description of the methods and classes used. Documents such as a comprehensive user's guide and web formatted API description page would be highly recommended implements for future iterations of this project. All of the setup materials are easy to find and search for any user due to their popularity and open-source nature.

Usage

This section will explore user interaction with the framework in its console-based iteration found in the RabbitPy.py file. All of the available commands will be demonstrated.

1. User registration and login

When RabbitPy.py is first run, the user is presented with the register, login, and close_connection commands. The register command has them set a username and password for their account and stores their credentials in user_credentials.json. This json file persists across multiple sessions and would be located on a host machine in an implementation of the framework. After registration, the user is allowed in and a personal queue is created for them under their username. After registration, users are immediately able to access the other commands.

```
Successfully connected to the RabbitMQ server.
Welcome to the RabbitPy Data-Sharing Framework!
Please select an interface to begin.

Enter command:
register     login   close_connection
register
Please create a user_id: testing_user
Please create a password: eBFK7tjw
User testing_user registered successfully.
Message sent: User testing_user has registered the system.
```

Figure 25: User registration with RabbitPy

```
Enter command:
register     login   close_connection
login
Enter user_id: ecooper
Enter password: 1234
Successfully connected to the RabbitMQ server.
Message sent: User ecooper has has logged in the system.
```

Figure 26: User Login to RabbitPy

2. Managing Formats

Through the usage of add_want_format and add_convert_format, users are able to add their preferences and available conversion types to a JSON file similar to user_credentials. Users can also check both their own and another user's formats as easily readable lists using the check$_{u}ser_{f}ormatsmethodbyinputtingtherelevantuserID.Ourcurrentlyallowedformatsandconversiontyp$

```
add-want-format
Enter format: .pdf
Format .pdf has been added for user testing_user.
```

Figure 27: Adding a want format

```
add-convert-format
Select a conversion format:
1. csv_to_pdf
2. pdf_to_csv
3. csv_to_json
4. text_to_csv
5. json_to_csv
6. csv_to_text
7. pdf_to_text
8. text_to_pdf
Enter the number of your choice: 1
```

Figure 28: Adding a convert format

```
check_formats
User testing_user wants these formats: ['.pdf', '.csv']
User testing_user can convert these formats: ['.csv to .pdf']
```

Figure 29: Checking a user's want and convert formats

3. Sending a message

The send message command asks a user for the userID of the intended recipient and their message text. This relays a message to the corresponding queue, which is then sent over the active RabbitMQConnection.

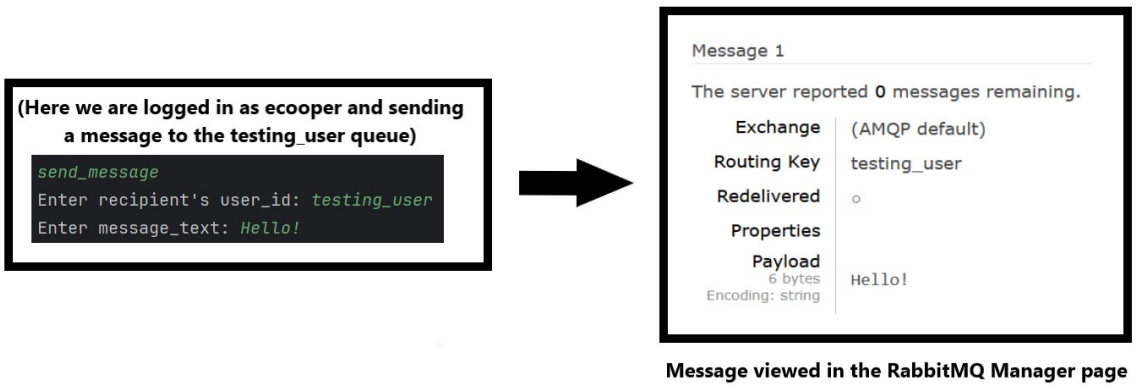

Figure 30: Sending a message to a user

4. Creating a queue

Queue creation is currently limited to only creating a new one whenever a user registers with the framework. This is due to the 100-queue limit that the free tier of the CloudAMQP server provides. This issue can be amended with a more robust cloud platform and some slight modification to the create_user_queue method in RabbitPy. If need be, the system administrator is also able to manually create new queues as well as exchanges inside of the RabbitMQ manager on the CloudAMQP website.

5. Preparing a file for upload to RabbitMQ

For this command, the user provides the file path for the item they wish to share. The size of the item is checked before upload to ensure that it is under RabbitMQ's imposed size limit of 2GB. As long as the file is smaller than 2GB, the file will be uploaded to the logged in user's queue.

```
upload
RABBITMQ HAS A STRICT 2GB SIZE LIMIT, USE MAGICWORMHOLE FOR LARGER FILES!

Enter file_path: C:\Users\Tedio\OneDrive\Pictures\Personal\IMG_20240615_142153.png
```

Figure 31: Uploading a file to RabbitMQ

6. Downloading a message from RabbitMQ

We now specify the queue we wish to download a message from, and the most recent upload is saved in our root directory.

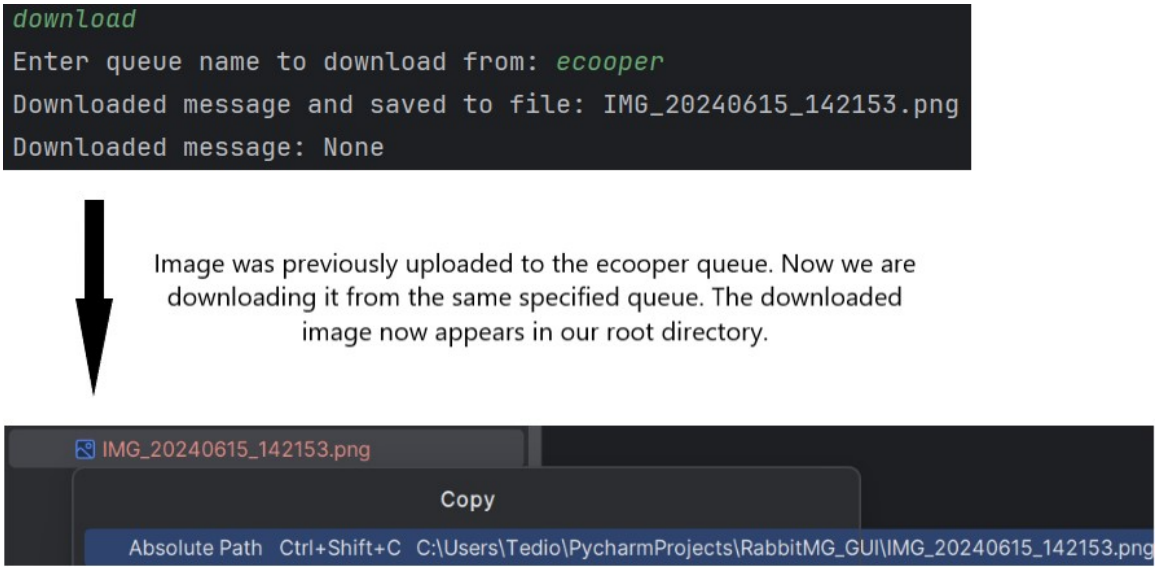

Figure 32: Downloading a file from RabbitMQ to computer

7. Sending a file using MagicWormhole

Due to the size limit imposed by RabbitMQ, larger file transfers are handled by the MagicWormhole Library. Instead of specifying a queue to send to, user IDs are entered directly as well as the file that is to be shared. Both users will need to be running RabbitPy.py in order to have an active connection to each other so that the file can be transferred.

When the sender uses the "magicwormhole" command, they call the handle_magic_wormhole function and pass the path to the file they wish to share as well as the userID of the intended recipient. This function then calls the send method of the Wormhole class, which is then executed. A subprocess runs this command in a new process, which is the only instance of multi-threading in this project.

```
magicwormhole
Enter file_path: C:\Users\Tedio\OneDrive\Pictures\Personal\dog.jpg
Enter user_id to send file to: testing_user
```

Figure 33: Sending with MagicWormhole

When the intended recipient is preparing to receive a file, they will use the ”receive_messages” command, which calls the corresponding handling method. Wormhole’s receive method is then executed in a new subprocess. An important note about this command is that the testing environment (in this case, Jefferson Lab) blocked the CloudAMQP sites with its firewall and they had to be manually added to a whitelist. This is a factor which will mostly likely affect future implementations.

The general flow of this process is described in the figure below:

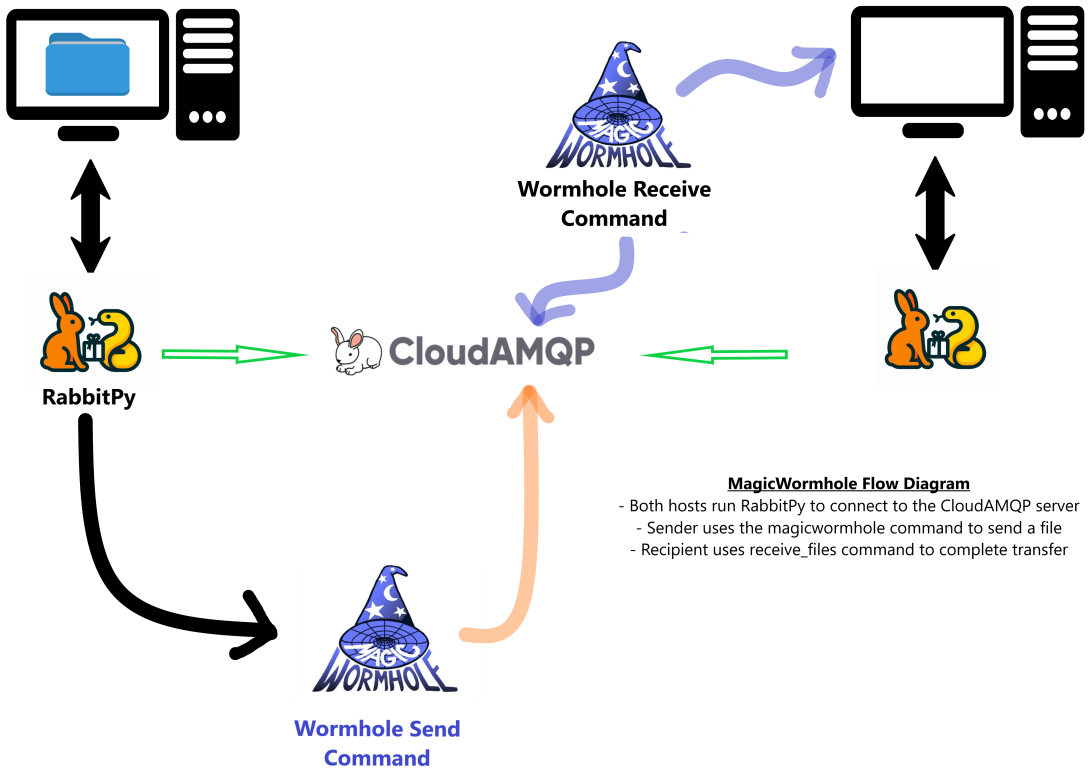

Figure 34: MagicWormhole flow in the RabbitPy program

8. Closing the connection

The close connection command calls the RabbitMQConnection class’s close method and sets the current userID to None, terminating the active connection.

```
Enter command:
    convert     add-want-format     add-convert-format  check_formats   upload  download    receive_messages    send_message
    magicwormhole
    close_connection
close_connection
Connection to the RabbitMQ server has been closed.

Process finished with exit code 0
```

Figure 35: Closing connection to RabbitMQ

CHAPTER VI:
CONCLUSION

Future Research

This research was intended to elaborate on the previously established Java API framework, offering further streamlining and elaboration as well as a comprehensive application-driven implementation. Through the usage of this framework, scientific users are expected to be able to upload, convert, download, and transfer files remotely and asynchronously amongst themselves. A key concern of the previous research was accessibility, which has been addressed by changing from the Java programming language to primarily Python due to its greater readability and less setup work with interacting with RabbitMQ and MagicWormhole. Another key concern is security, which saw the basic implementation of usernames and passwords for the server users. In an actual implementation of this framework, this information would be hashed and then stored on the remote server where it is verified upon user login. This is a change that will need to be addressed in future versions. In addition, a more comprehensive and professional documentation of this library would also be of great benefit to potential users.

A primary concern that will be noted is scalability. Our testing environment was kept to a small RabbitMQ server with 20 queues. Should this framework be implemented, we can expect much larger user bases and much larger file transfers, so the architecture must be capable of handling the load of those aspects. Funding would need to be allocated to renting or developing an architecture if one does not already exist on the server host's side.

LITERATURE CITED

[1]A. Nguyen, "A service-oriented framework using rabbitmq for physics collaboration," MA thesis (Christopher Newport University, 2023).

[2]T. G.-J. G. M. Roy, "Introduction to pika," Pivotal Software (2023).

[3]J. Williams, *Rabbitmq in action: distributed messaging for everyone* (Manning, 2012).

[4]G. M. R. J. Titcumb, *Rabbitmq in depth* (Manning, 2017).

[5]C. Villarreal, "Proof of concept of a data translation and messaging system using rabbiymq," MA thesis (Christopher Newport University, 2022).

[6]RabbitMQ, "Rabbitmq tutorials - rabbitmq," RabbitMQ (Accessed 9 November 2023).

[7]Baeldung, "Introduction to rabbitmq," Baeldung (2022).

[8]M. Lutz, *Python pocket reference* (O'Reilly Media Inc, 2014).