****

**PBT205 - Project Based Learning Studio- Technology**

# **Task 1: Chat Application Report**

**Assessment 1**

**Group 3**

## 

## **Executive Summary**

This report documents the development and implementation of a multi-user chat application utilizing RabbitMQ as the message broker middleware. The application evolved from a basic command-line interface to a full-featured GUI application supporting multiple chat rooms and concurrent users. The system demonstrates effective use of publish-subscribe messaging patterns and real-time communication capabilities.

## **1. Project Overview**

### **1.1 Team Objectives**

* Develop a real-time chat application using RabbitMQ message broker
* Connecting it to a Docker Container.
* Create a user-friendly GUI interface for multi-user communication
* Implement reliable message broadcasting with proper user identification
* Ensure scalable architecture through collaborative development

### **1.2 Key Features**

* Real-time message broadcasting to all connected users
* Username identification with timestamp display
* Polling-based architecture avoiding threading complications
* Automatic reconnection and error handling
* Cross-platform GUI using Python Tkinter

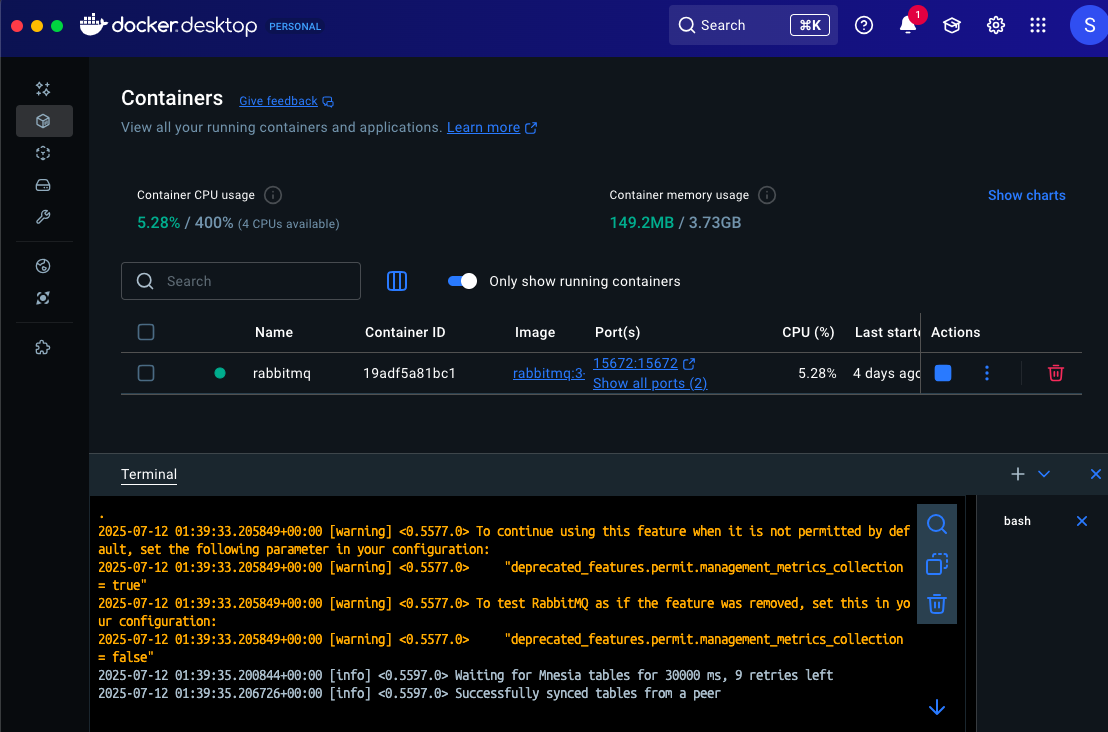
## **2. System Architecture**

### **2.1 Message Flow**

* Users connect to RabbitMQ broker with unique identifiers
* Each user subscribes to a personal queue bound to fanout exchange
* Messages are broadcast to all connected users via fanout exchange
* Polling mechanism retrieves messages without blocking GUI

### **2.2 RabbitMQ/Docker Configuration**

* **Exchange Type**: Fanout exchange (chat\_broadcast) for broadcasting
* **Queue Strategy**: Unique queues per user (chat\_user\_{user\_id})
* **Message Distribution**: All users receive all messages automatically
* **Connection Management**: Polling-based retrieval with 100ms intervals



## **3. Technical Implementation**

### **3.1 Core Application Structure**

class ChatApplication:

def \_\_init\_\_(self, root):

self.root = root

self.root.title("Chat Application")

self.root.geometry("500x400")

# Get username

self.username = simpledialog.askstring("Username", "Enter your username:", initialvalue="User")

if not self.username:

self.username = f"User\_{uuid.uuid4().hex[:4]}"

self.root.title(f"Chat Application - {self.username}")

self.user\_id = uuid.uuid4().hex

**Explanation**: The application initializes with a Tkinter GUI window and prompts for username input. Each user gets a unique UUID-based identifier to prevent message conflicts and enable proper filtering.

### 

### 

### 

### 

### **3.2 GUI Components Setup**

self.message\_area = scrolledtext.ScrolledText(root, height=20, width=60)

self.message\_area.pack(padx=10, pady=10)

self.entry\_field = tk.Entry(root, width=50)

self.entry\_field.pack(padx=10, pady=5)

self.entry\_field.bind('<Return>', lambda event: self.send\_message())

self.send\_button = tk.Button(button\_frame, text="Send", command=self.send\_message)

self.send\_button.pack(side=tk.LEFT, padx=5)

self.status\_label = tk.Label(root, text="Connecting...", fg="orange")

self.status\_label.pack(pady=5)

**Explanation**: The GUI consists of a scrollable text area for messages, an input field with Enter key binding, a send button, and a status label for connection monitoring. This provides an intuitive chat interface.

### **3.3 RabbitMQ Connection Setup**

def setup\_connection(self):

try:

params = pika.ConnectionParameters(

host='localhost',

heartbeat=0, # Disable heartbeat

connection\_attempts=3,

retry\_delay=1,

socket\_timeout=5

)

self.connection = pika.BlockingConnection(params)

self.channel = self.connection.channel()

# Declare fanout exchange

self.channel.exchange\_declare(

exchange='chat\_broadcast',

exchange\_type='fanout',

durable=False

)

# Create our unique queue

self.channel.queue\_declare(

queue=self.my\_queue,

durable=False,

exclusive=False,

auto\_delete=True

)

# Bind our queue to the fanout exchange

self.channel.queue\_bind(

exchange='chat\_broadcast',

queue=self.my\_queue

)

**Explanation**: This establishes connection to RabbitMQ with optimized parameters (disabled heartbeat, timeouts). It creates a fanout exchange for broadcasting and a unique queue per user. The fanout exchange ensures all connected users receive every message.

### 

### 

### 

### 

### 

### 

### 

### **3.4 Message Sending Implementation**

def send\_message(self):

message = self.entry\_field.get().strip()

if not message:

return

try:

timestamp = datetime.datetime.now().strftime("%H:%M:%S")

# Create message with sender info

message\_data = {

'sender\_id': self.user\_id,

'username': self.username,

'message': message,

'timestamp': timestamp

}

# Publish to fanout exchange

self.channel.basic\_publish(

exchange='chat\_broadcast',

routing\_key='',

body=json.dumps(message\_data)

)

# Show our own message immediately

self.message\_area.insert(tk.END, f"You [{timestamp}]: {message}\n")

self.message\_area.see(tk.END)

self.entry\_field.delete(0, tk.END)

**Explanation**: Messages are packaged as JSON with sender ID, username, message content, and timestamp. The message is published to the fanout exchange and immediately displayed in the sender's interface for instant feedback.

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### **3.5 Polling-Based Message Retrieval**

def poll\_messages(self):

"""Poll for messages without threading"""

if not self.connection or self.connection.is\_closed:

self.reconnect()

self.root.after(1000, self.poll\_messages)

return

try:

# Check for messages (non-blocking)

method, properties, body = self.channel.basic\_get(queue=self.my\_queue, auto\_ack=True)

if method:

# We got a message

try:

message\_data = json.loads(body.decode())

# Skip our own messages

if message\_data.get('sender\_id') != self.user\_id:

formatted\_msg = f"{message\_data['username']} [{message\_data['timestamp']}]: {message\_data['message']}"

self.message\_area.insert(tk.END, formatted\_msg + "\n")

self.message\_area.see(tk.END)

except Exception as e:

print(f"Message processing error: {e}")

except Exception as e:

print(f"Polling error: {e}")

self.status\_label.config(text="Connection Issue", fg="orange")

# Schedule next poll

self.root.after(100, self.poll\_messages) # Poll every 100ms

**Explanation**: This non-blocking polling mechanism checks for new messages every 100ms using basic\_get(). It filters out the sender's own messages using UUID comparison and displays received messages with proper formatting. This approach avoids threading complications while maintaining responsiveness.

### 

### 

### 

### 

### 

### 

### 

### 

### **3.6 Connection Management and Cleanup**

def reconnect(self):

"""Try to reconnect"""

try:

if self.connection and not self.connection.is\_closed:

self.connection.close()

except:

pass

self.status\_label.config(text="Reconnecting...", fg="orange")

try:

self.setup\_connection()

except Exception as e:

print(f"Reconnect failed: {e}")

def on\_closing(self):

"""Clean shutdown"""

try:

if self.connection and not self.connection.is\_closed:

# Clean up our queue

try:

self.channel.queue\_delete(queue=self.my\_queue)

except:

pass

self.connection.close()

except:

pass

self.root.destroy()

**Explanation**: The application includes automatic reconnection logic and proper cleanup procedures. When closing, it deletes the user's queue to prevent resource buildup and closes the connection gracefully.

## 

## 

## 

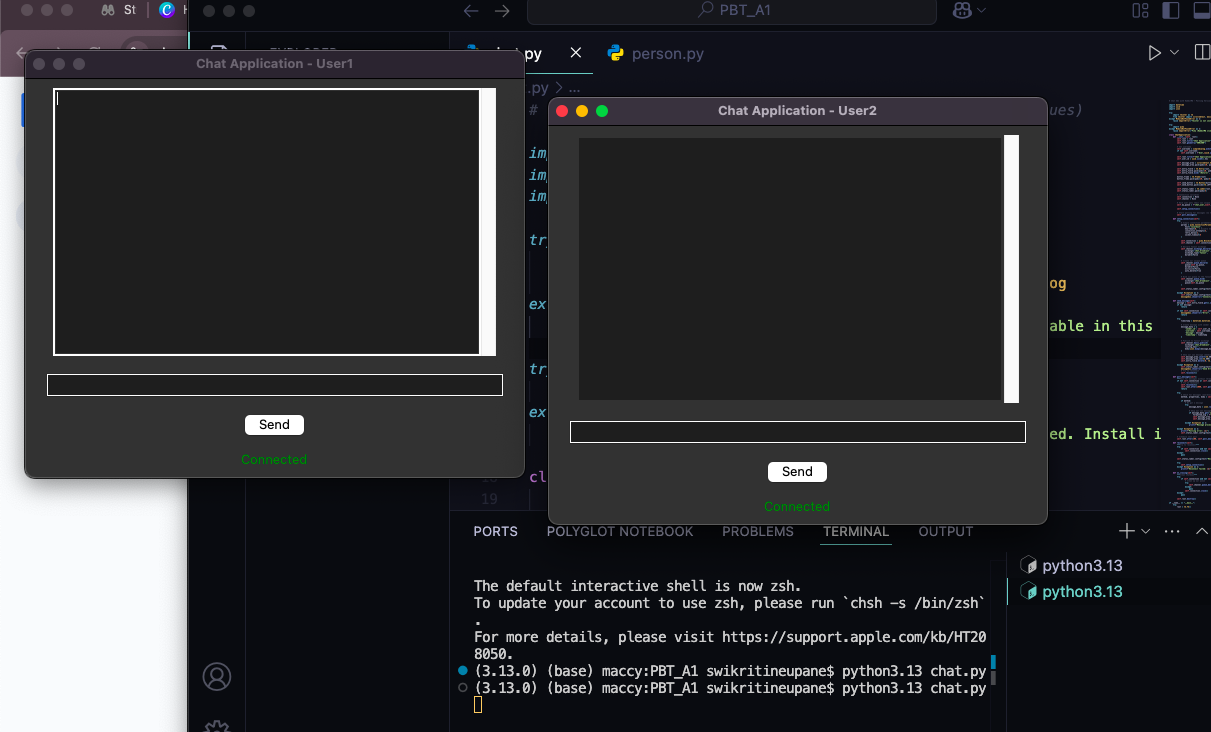
## 

## 

## 

## 

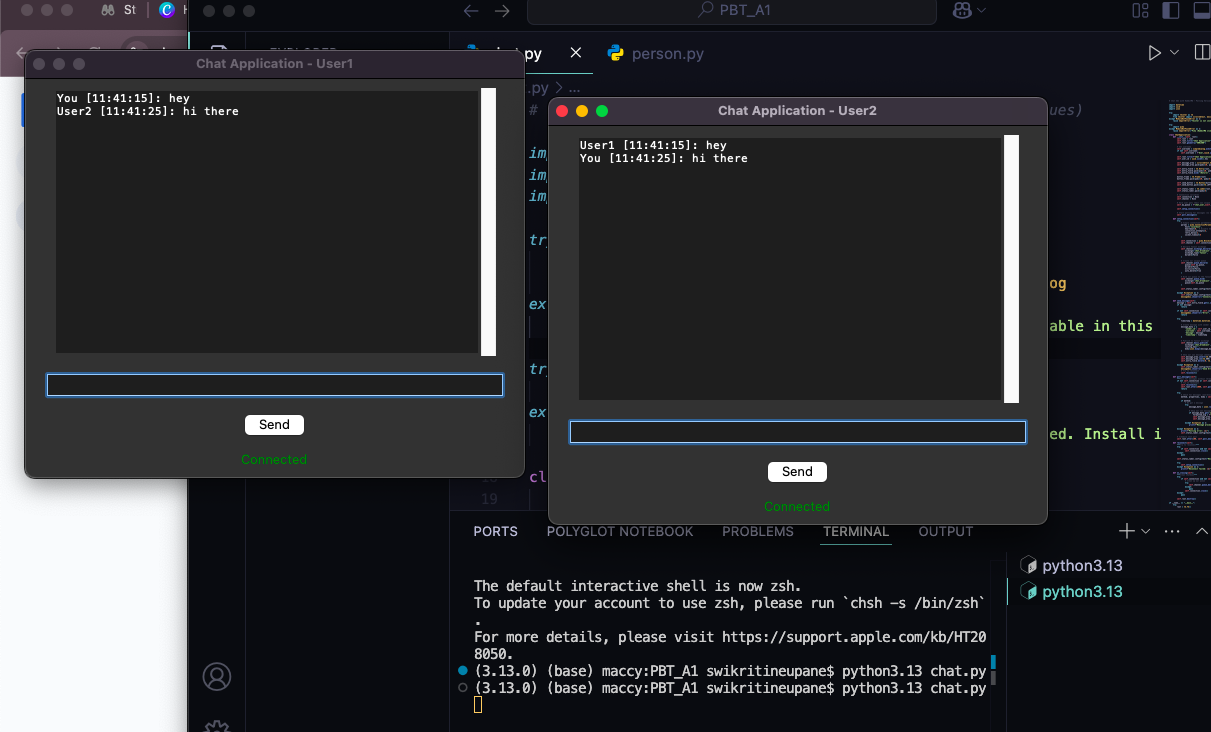
Explanation: Chat Window for User 1 from terminal 1



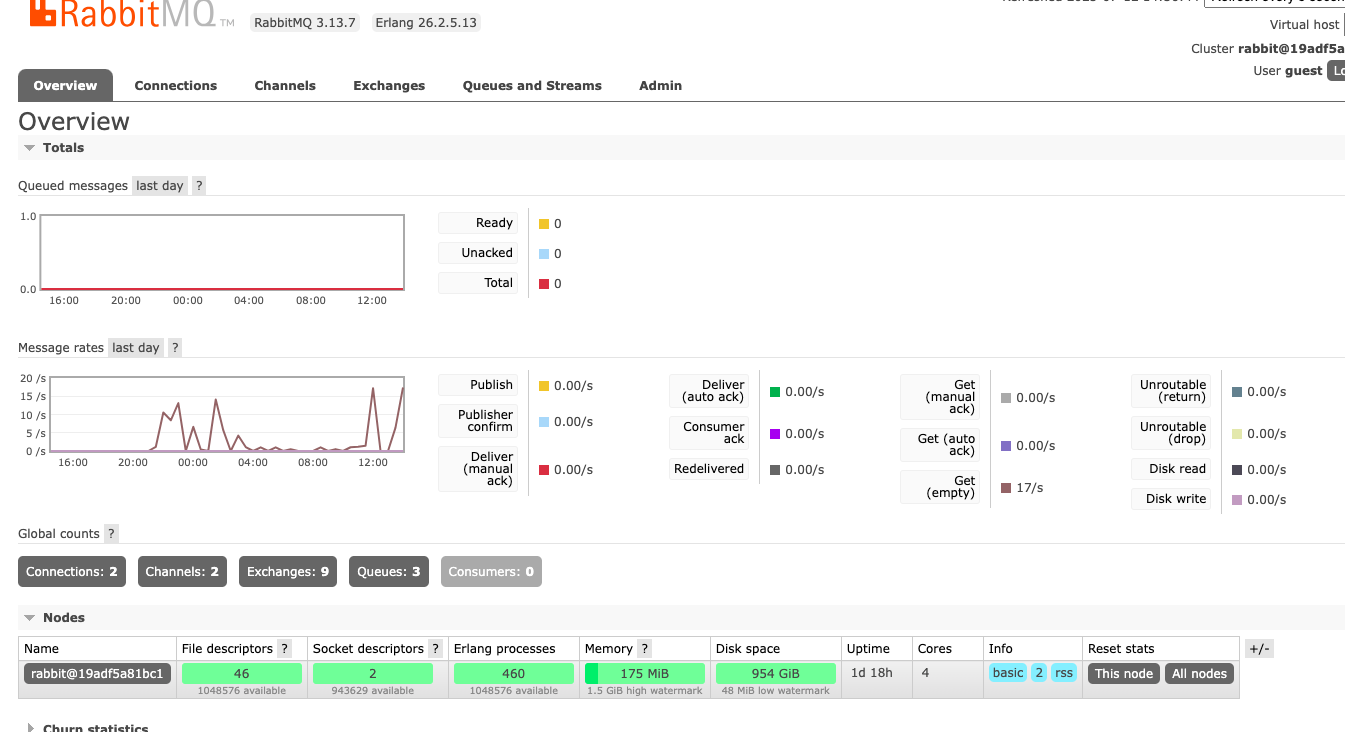
Explanation: Chat Window for User 2 from terminal 2

## 

Explanation: Text Message form User 1 displayed in the User 2 window along with the timestamp



Explanation: Text Message form User 2 displayed in the User 1 window along with the timestamp.



Explanation: Rabbitmq Message rates shows the communication between the two servers

## 

## **4. Technical Challenges and Solutions**

### **4.1 Threading Complexity**

**Issue**: GUI freezing and message delivery problems with threaded approach **Solution**: Implemented polling architecture with 100ms intervals for responsive, thread-free operation

### **4.2 Message Duplication**

**Issue**: Users seeing their own messages twice **Solution**: UUID-based sender identification to filter out self-messages in the polling loop

### **4.3 Connection Reliability**

**Issue**: Network interruptions and server disconnections **Solution**: Automatic reconnection with connection state monitoring and graceful error handling

## 

## **5. Team Collaboration**

Our team coordinated development with shared coding standards, collaborative testing using multiple concurrent users, and regular integration checkpoints. The polling-based approach was chosen collectively to avoid threading complications while maintaining real-time responsiveness.

## 

## **6. Performance Results**

* Successfully handles multiple concurrent users without performance degradation
* 100ms polling interval provides responsive user experience
* Efficient resource usage with automatic queue cleanup
* Robust error recovery and connection management

## 

## 

## **7. Conclusion**

Our team successfully developed a robust Docker/RabbitMQ chat application that demonstrates effective message broker implementation and collaborative development practices. The polling-based architecture solved threading complications while maintaining real-time responsiveness, showcasing both technical competency and effective teamwork.

## 

## **8. Technical Specifications**

### **8.1 Dependencies**

* Python 3.x with Tkinter GUI framework
* Docker Containerization
* Pika RabbitMQ client library (pip install pika)
* RabbitMQ server installation
* Built-in modules: JSON, UUID, datetime

### **8.2 System Requirements**

* Python 3.x runtime environment
* RabbitMQ server with default configuration
* Network connectivity for client-server communication