

ASSIGNMENT 1

Group 3

Trading Application

PBT205: Project Based Learning Studio: Technology

July 13, 2025

Overview

This report talks about how we built a basic trading system using Python and RabbitMQ. The goal was to simulate a really simple exchange (only for one stock — XYZ Corp) where people can place buy and sell orders, kind of like a minimal version of what actual trading platforms do. Honestly, setting up RabbitMQ at the beginning felt kind of tricky and the Docker setup took a few attempts before we figured out the ports and how to access the management UI. But once that was sorted, building and testing the messaging-based system became much more manageable.

Technology Used

- Language: Python 3 (chosen for its simplicity and readability)
- Middleware: RabbitMQ (run using Docker, with management interface on port 15672)
- Data Format: JSON for sending messages between scripts
- Libraries:
 - pika – for communicating with RabbitMQ
 - json – for encoding/decoding messages
 - sys – for handling command-line argument
 - datetime – for timestamps in orders and trades

Architecture

The system has three main scripts:

- `send_order.py`: This is a simple command-line tool that creates and submits an order.
- `exchange.py`: This acts like the core exchange server. It receives all submitted orders and tries to match them.
- `Trading_GUI`:

We decided to use separate queues (orders and trades) to keep everything modular. This way, future components like GUIs or loggers could just subscribe to the trades queue without needing to touch the core logic.

Components Description

1. Send_order.py:

This script is meant to simulate the trader's side. It takes input from the command line: username, port, side (BUY/SELL), quantity, and price. It puts all that into a Python dictionary, adds a timestamp, and then sends it to RabbitMQ as a JSON string.

Key steps in the code:

- Checks that all 5 arguments are provided (no fancy validation beyond that)
- Uses pika to connect to RabbitMQ on the given port
- Declares the orders queue to make sure it exists
- Sends the message using `basic_publish()`
- Closes the connection after sending — it doesn't wait for a reply or confirmation

2. Exchange.py:

This is the core of the system. It runs continuously, listening to the orders queue for incoming messages. When a new order arrives, it attempts to match it against existing unmatched orders in memory.

We maintain two lists:

- `buy_orders`: pending BUY orders
- `sell_orders`: pending SELL orders

Matching Logic:

- A BUY order looks for a matching SELL order at the same or lower price
- A SELL order looks for a matching BUY order at the same or higher price
- If a match is found, a trade is executed:
- The trade is printed to the terminal
- A message is published to the trades queue

Writing the matching logic took a few iterations, particularly around making sure trades were matched correctly and removing the filled orders from their respective lists.

3. GUI:

This is the system's user-facing component. It has a Tkinter-based graphical user interface that shows the most recent trade price for each stock as well as the recent trade history while listening to the trades queue.

Important parts of the code:

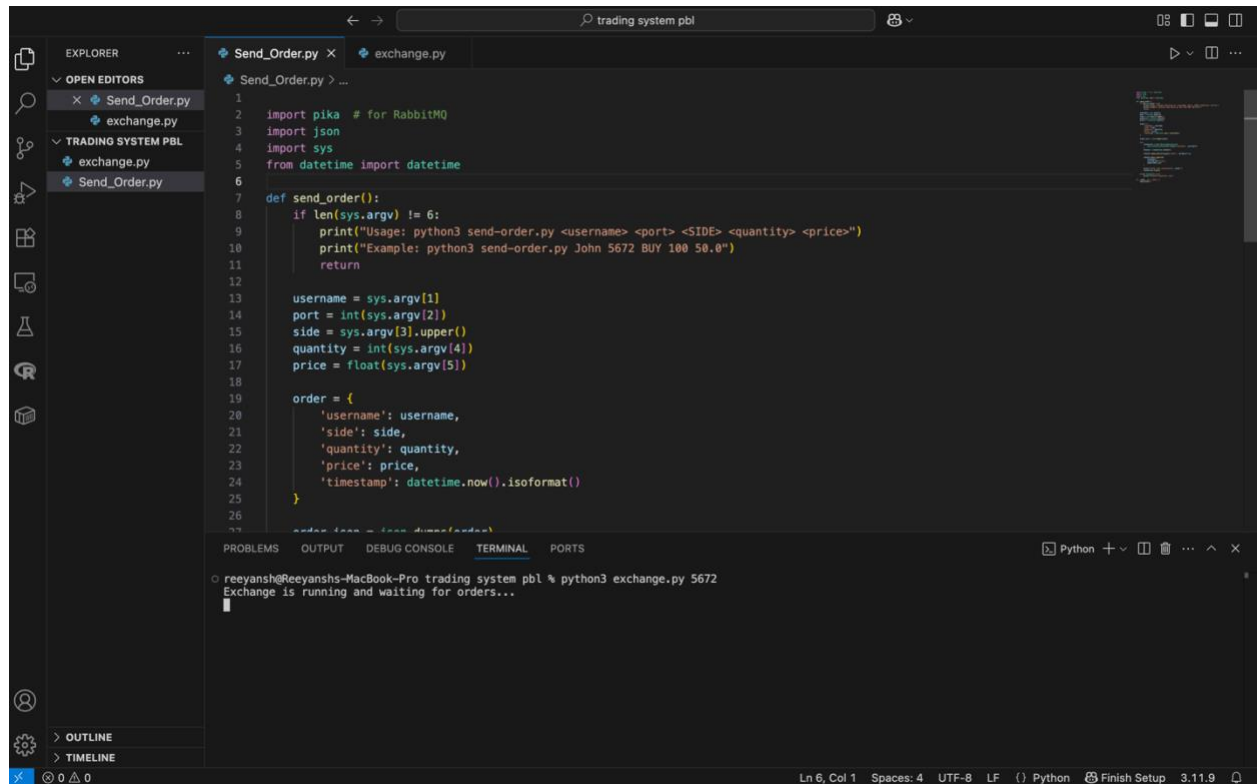
- Subscribes to the trades queue by using pika in a background thread.
- Every time a trade message is received, the most current price per stock is stored by updating a dictionary called latest_prices. The trade is added to the trade_history in-memory list and the GUI's displayed trade history and stock price is modified.
- Features a drop-down menu for changing between stock symbols (such as ABC and XYZ).
- Shows the chosen stock's most recent price in huge font.
- Displays, in a scrolling text window, the last ten trades for the chosen stock.

Obstacles in the development process:

- Ensuring that the Tkinter UI may be updated by the background RabbitMQ listener without resulting in thread problems. After() callbacks were used to handle this.
- Ensuring that the dropdown's stock list is updated each time a new stock symbol appears in a trade.

Testing and Validation

1. Test 1: Successful Trade Execution



The screenshot shows a Visual Studio Code editor window with the file `Send_Order.py` open. The file contains a Python script for sending orders. The terminal window at the bottom shows the command `python3 exchange.py 5672` being executed, and the output is `Exchange is running and waiting for orders...`.

```
1 import pika # for RabbitMQ
2 import json
3 import sys
4 from datetime import datetime
5
6
7 def send_order():
8     if len(sys.argv) != 6:
9         print("Usage: python3 send-order.py <username> <port> <SIDE> <quantity> <price>")
10        print("Example: python3 send-order.py John 5672 BUY 100 50.0")
11        return
12
13    username = sys.argv[1]
14    port = int(sys.argv[2])
15    side = sys.argv[3].upper()
16    quantity = int(sys.argv[4])
17    price = float(sys.argv[5])
18
19    order = {
20        'username': username,
21        'side': side,
22        'quantity': quantity,
23        'price': price,
24        'timestamp': datetime.now().isoformat()
25    }
26
27    order_json = json.dumps(order)
```

reeyansh@Reeyanshs-MacBook-Pro trading system pbl % python3 exchange.py 5672
Exchange is running and waiting for orders...

```
exchange.py > main
64 def main():
81     def callback(ch, method, properties, body):
85         trades = match_order(order)
86         for trade in trades:
87             trade_json = json.dumps(trade)
88             channel.basic_publish(
89                 exchange='',
90                 routing_key='trades',
91                 body=trade_json
92             )
93             print_trade_executed(trade)
```

reeyansh@Reeyanshs-MacBook-Pro trading system pbl % python3 exchange.py 5672
Exchange is running and waiting for orders...

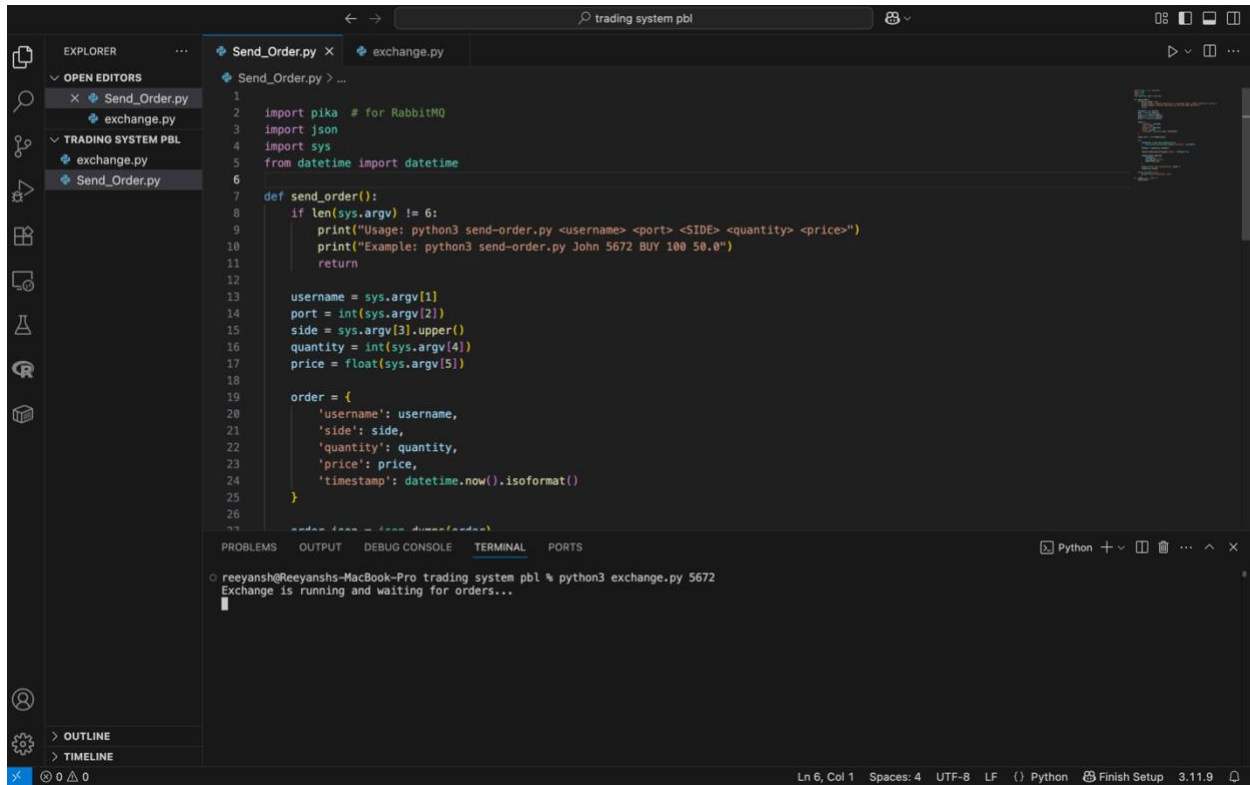
--- New Order Received ---
User: Alice
Side: BUY
Quantity: 100
Price: \$50.00
Timestamp: 2025-07-10T19:06:00.663585

--- New Order Received ---
User: Bob
Side: SELL
Quantity: 100
Price: \$50.00
Timestamp: 2025-07-10T19:06:16.037541

=== TRADE EXECUTED ===
Alice buys 100 shares from Bob at \$50.00
Timestamp: 2025-07-10T19:06:16.050975

This test simulates a successful trade between two users. Alice submits a BUY order for 100 shares at \$50.00, followed by Bob who submits a SELL order for 100 shares at \$50.00. The exchange matches the orders based on price and executes a trade at the seller's price of \$50.00. This confirms that the core matching engine is working correctly.

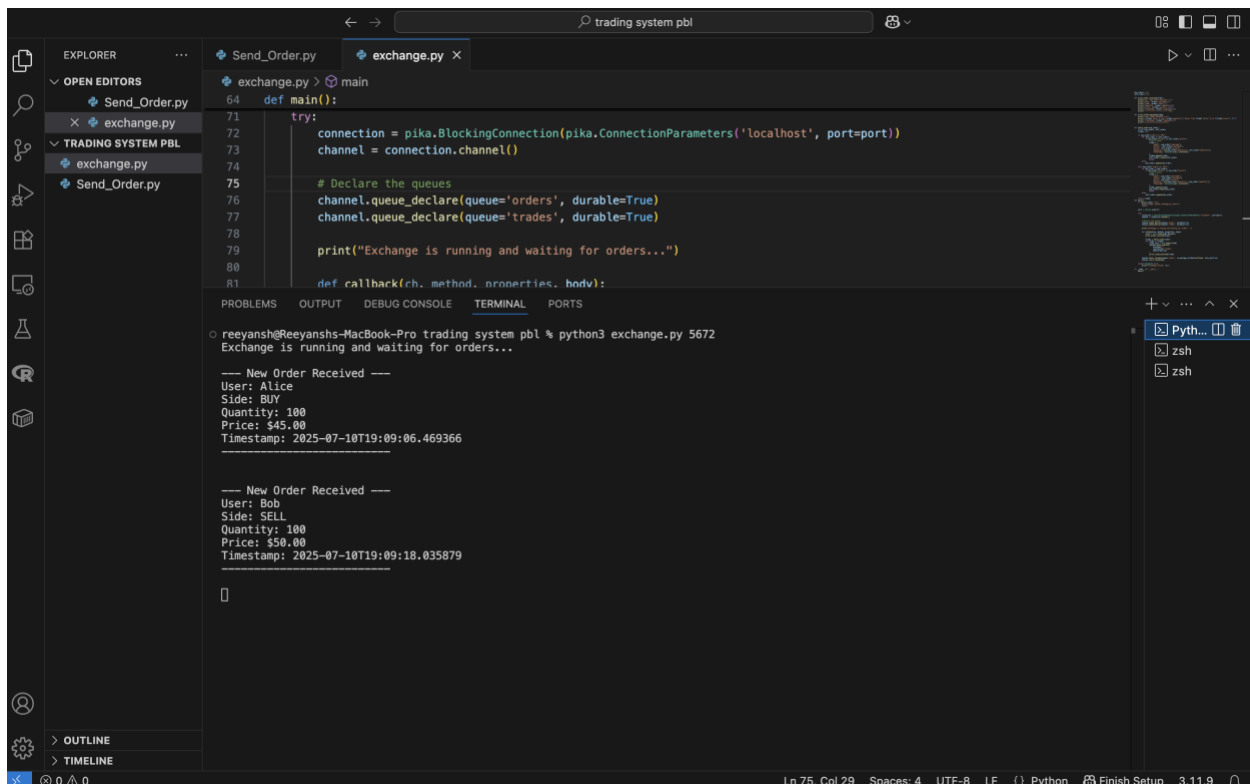
2. Test 2: No Match – Order Stay in Memory



This screenshot shows the VS Code editor with the file `Send_Order.py` open. The code defines a `send_order()` function that takes command-line arguments for username, port, side, quantity, and price, and constructs a JSON order object. The terminal window at the bottom shows the command `python3 exchange.py 5672` being executed, with the output "Exchange is running and waiting for orders..."

```
1
2 import pika # for RabbitMQ
3 import json
4 import sys
5 from datetime import datetime
6
7 def send_order():
8     if len(sys.argv) != 6:
9         print("Usage: python3 send-order.py <username> <port> <SIDE> <quantity> <price>")
10        print("Example: python3 send-order.py John 5672 BUY 100 50.0")
11        return
12
13    username = sys.argv[1]
14    port = int(sys.argv[2])
15    side = sys.argv[3].upper()
16    quantity = int(sys.argv[4])
17    price = float(sys.argv[5])
18
19    order = {
20        'username': username,
21        'side': side,
22        'quantity': quantity,
23        'price': price,
24        'timestamp': datetime.now().isoformat()
25    }
26
27    order_json = json.dumps(order)
```

reeyansh@Reeyansh-MacBook-Pro trading system pbl % python3 exchange.py 5672
Exchange is running and waiting for orders...



This screenshot shows the VS Code editor with the file `exchange.py` open. The code sets up a RabbitMQ connection and declares two queues: `orders` and `trades`. The terminal window at the bottom shows the command `python3 exchange.py 5672` being executed, with the output "Exchange is running and waiting for orders...". Below this, two "New Order Received" messages are shown, one for Alice (BUY) and one for Bob (SELL).

```
64 def main():
65     try:
66         connection = pika.BlockingConnection(pika.ConnectionParameters('localhost', port=port))
67         channel = connection.channel()
68
69         # Declare the queues
70         channel.queue_declare(queue='orders', durable=True)
71         channel.queue_declare(queue='trades', durable=True)
72
73         print("Exchange is running and waiting for orders...")
74
75     except KeyboardInterrupt:
76         sys.exit()
77
78     def callback(ch, method, properties, body):
79         pass
80
81     channel.basic_consume(queue='orders', on_message_callback=callback, auto_ack=True)
```

reeyansh@Reeyansh-MacBook-Pro trading system pbl % python3 exchange.py 5672
Exchange is running and waiting for orders...

--- New Order Received ---
User: Alice
Side: BUY
Quantity: 100
Price: \$45.00
Timestamp: 2025-07-10T19:09:06.469366

--- New Order Received ---
User: Bob
Side: SELL
Quantity: 100
Price: \$50.00
Timestamp: 2025-07-10T19:09:18.035879

This test confirms that the exchange correctly holds unmatched orders in memory without executing trades. Since Alice's BUY price (\$45.00) is below Bob's SELL price (\$50.00), the system does not perform any trade, which reflects accurate market behavior where buy prices must meet or exceed sell prices for execution.

GUI TESTING:

Test Case 1: Exact Match — BUY and SELL at Same Price

On sending a BUY and SELL order for 100 XYZ shares at \$50 (as seen above in the exchange testing), the gui.py shows:

- Latest price for XYZ: \$50
- The trade list in the trade history

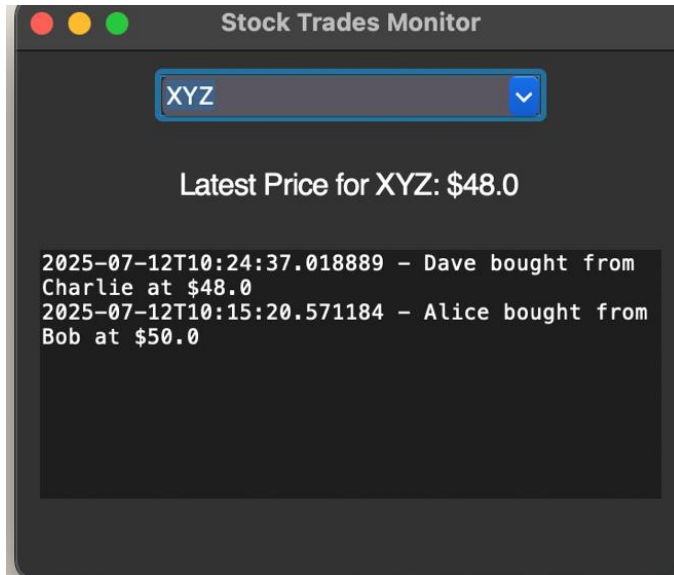


Test Case 2: Partial Match — SELL 150, BUY 100

On sending a SELL order for 150 XYZ shares at \$48 and a BUY order for 100 XYZ shares at \$49, the exchange.py prints a trade execution: Dave buys 100 shares from Charlie at \$48. 50 shares of Charlie's sell order remain in the order book. The gui.py shows:

- Latest Price for XYZ: \$48.0

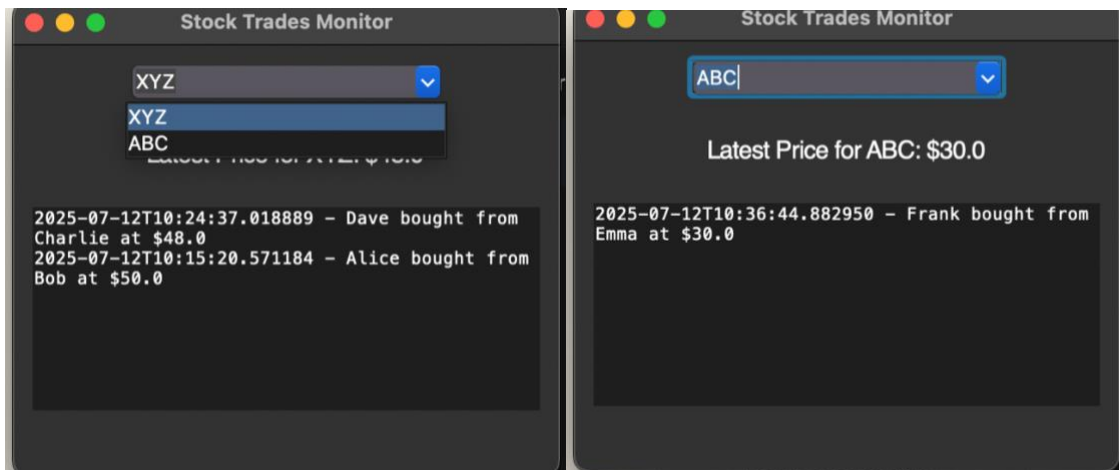
- The trade in the history, showing Dave and Charlie at \$48.



Test Case 3: Different Stock — ABC

On sending a SELL order for 100 ABC shares at \$30 and a BUY order for 100 ABC shares at \$32, the exchange.py prints a trade execution: Frank buys 100 shares from Emma at \$30. The gui.py shows:

- a dropdown including ABC
- Latest price for ABC: \$30
- Trade history



Appendix

Send_order.py:

```
# Import necessary modules

import pika          # For RabbitMQ communication
import json          # To format the message as JSON
import sys           # For command-line arguments
from datetime import datetime # To timestamp each order


# Function to print the sent order in a readable format
def print_order_sent(order):
    """Prints the details of the sent order."""
    print("\n--- Order Sent ---")
    print(f"User: {order['username']}")
    print(f"Stock: {order['stock']}")
    print(f"Side: {order['side']}") # BUY or SELL
    print(f"Quantity: {order['quantity']}")
    print(f"Price: ${order['price']:.2f}")
    print(f"Timestamp: {order['timestamp']}")
    print("-----\n")


# Function to construct and send an order message to RabbitMQ
def send_order():
    """
    Builds and sends an order to the RabbitMQ 'orders' queue.
    """
```

Expects: <username> <port> <stock> <BUY/SELL> <quantity> <price>

"""

Validate number of arguments

if len(sys.argv) != 7:

print("Usage: python3 send_order.py <username> <port> <stock> <SIDE> <quantity>
<price>")

print("Example: python3 send_order.py John 5672 XYZ BUY 100 50.0")

return

Extract command-line arguments

username = sys.argv[1]

port = int(sys.argv[2])

stock = sys.argv[3].upper() # Convert to uppercase (e.g., "xyz" -> "XYZ")

side = sys.argv[4].upper() # BUY or SELL

quantity = int(sys.argv[5]) # Number of shares

price = float(sys.argv[6]) # Price per share

Input validation

if side not in ("BUY", "SELL") or quantity <= 0 or price <= 0:

print("Invalid input: Side must be BUY or SELL, and quantity/price must be positive.")

return

Create the order as a dictionary

order = {

'username': username,

'stock': stock,

```
'side': side,  
'quantity': quantity,  
'price': price,  
'timestamp': datetime.now().isoformat() # Generate ISO 8601 timestamp  
}
```

```
# Convert the order to a JSON string
```

```
order_json = json.dumps(order)
```

```
try:
```

```
    # Connect to RabbitMQ at the specified port
```

```
    connection = pika.BlockingConnection(  
        pika.ConnectionParameters(host='localhost', port=port, ssl_options=None)
```

```
    )
```

```
    channel = connection.channel()
```

```
    # Declare the 'orders' queue to ensure it exists
```

```
    channel.queue_declare(queue='orders', durable=True)
```

```
    # Send the order to the 'orders' queue
```

```
    channel.basic_publish(  
        exchange="",
```

```
        routing_key='orders',
```

```
        body=order_json
```

```
    )
```

```
# Print the order confirmation to the terminal
print_order_sent(order)

# Close the connection
connection.close()

except Exception as e:
    # Handle and display connection or sending errors
    print(f"Failed Connection: {e}")

# Run the send_order() function if the script is executed directly
if __name__ == "__main__":
    send_order()
```

exchange.py:

```
import pika          # For RabbitMQ messaging
import json          # To decode order messages
import sys           # For command-line argument handling
from datetime import datetime # For timestamps

# Dictionary to hold order books for each stock
```

```
order_books = {}
```

```
# Helper function to print a nicely formatted incoming order
```

```
def print_order_received(order):
```

```
    """Prints the details of a received order."""
```

```
    print("\n--- New Order Received ---")
```

```
    print(f"User: {order['username']}")
```

```
    print(f"Stock: {order['stock']}")
```

```
    print(f"Side: {order['side']}")
```

```
    print(f"Quantity: {order['quantity']}")
```

```
    print(f"Price: ${order['price']:.2f}")
```

```
    print(f"Timestamp: {order['timestamp']}")
```

```
    print("-----\n")
```

```
# Helper function to print a nicely formatted executed trade
```

```
def print_trade_executed(trade):
```

```
    """Prints the details of an executed trade."""
```

```
    print("\n=== TRADE EXECUTED ===")
```

```
    print(f"{trade['buyer']} buys {trade['quantity']} shares of {trade['stock']} from  
{trade['seller']} at ${trade['price']:.2f}")
```

```
    print(f"Timestamp: {trade['timestamp']}")
```

```
    print("=====\n")
```

```
# Core matching function
```

```
def match_order(new_order):
```

```
    """
```

Tries to match a new incoming order with existing opposite-side orders from the same stock's order book. Supports partial matching and multiple trades.

"""

```
stock = new_order.get("stock", "XYZ") # Default stock if not specified
```

```
side = new_order["side"]
```

```
# Initialize order book for stock if this is the first order for it
```

```
if stock not in order_books:
```

```
    order_books[stock] = {"BUY": [], "SELL": []}
```

```
# Determine which side of the order book we're matching against
```

```
opposite_side = "SELL" if side == "BUY" else "BUY"
```

```
matching_orders = order_books[stock][opposite_side]
```

```
# Sort opposite-side orders for best price matching (price priority)
```

```
if opposite_side == "SELL":
```

```
    matching_orders.sort(key=lambda o: o['price']) # Match BUY with lowest sell
```

```
else:
```

```
    matching_orders.sort(key=lambda o: -o['price']) # Match SELL with highest buy
```

```
trades = [] # To collect all resulting trades
```

```
remaining_qty = new_order["quantity"]
```

```
# Try to match with existing orders while quantity remains
```

```
while matching_orders and remaining_qty > 0:
```

```
    existing_order = matching_orders[0]
```

```

# Check if prices match according to trading logic
price_match = (
    (side == "BUY" and new_order["price"] >= existing_order["price"]) or
    (side == "SELL" and new_order["price"] <= existing_order["price"])
)

if not price_match:
    break # Stop if no suitable match is found

# Trade quantity is the minimum of remaining quantities
trade_qty = min(remaining_qty, existing_order["quantity"])

# Create the trade record
trade = {
    "stock": stock,
    "buyer": new_order["username"] if side == "BUY" else existing_order["username"],
    "seller": existing_order["username"] if side == "BUY" else new_order["username"],
    "price": existing_order["price"] if side == "BUY" else new_order["price"],
    "quantity": trade_qty,
    "timestamp": datetime.now().isoformat()
}

trades.append(trade)

# Update quantities
remaining_qty -= trade_qty

```



```

existing_order["quantity"] -= trade_qty

# If matched order is fully filled, remove it from the book
if existing_order["quantity"] == 0:
    matching_orders.pop(0)

# If any quantity of the new order remains unmatched, store it in the book
if remaining_qty > 0:
    new_order["quantity"] = remaining_qty
    order_books[stock][side].append(new_order)

return trades # Return list of trades executed (could be empty)

# Entry point for the exchange system
def main():
    """Main function to connect to RabbitMQ and start listening for orders."""
    if len(sys.argv) != 2:
        print("Usage: python exchange.py <port>")
        return

    port = int(sys.argv[1]) # RabbitMQ port passed as argument

    try:
        # Connect to RabbitMQ
        connection = pika.BlockingConnection(
            pika.ConnectionParameters('localhost', port=port, ssl_options=None)

```

```
)

channel = connection.channel()

# Declare necessary queues

channel.queue_declare(queue='orders', durable=True)

channel.queue_declare(queue='trades', durable=True)

print("Exchange is running and waiting for orders...")

# Define callback to handle incoming orders
def callback(ch, method, properties, body):

    order = json.loads(body.decode()) # Decode JSON message
    print_order_received(order)      # Log received order

    trades = match_order(order)      # Match and process trades
    for trade in trades:

        trade_json = json.dumps(trade)

        # Publish trade result to 'trades' queue
        channel.basic_publish(exchange='', routing_key='trades', body=trade_json)

        print_trade_executed(trade) # Log trade result

# Begin consuming from 'orders' queue

channel.basic_consume(queue='orders', on_message_callback=callback,
auto_ack=True)

channel.start_consuming()
```

```

except KeyboardInterrupt:

    print("Exchange shutting down gracefully.")

except Exception as e:

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


# Launch program

if __name__ == "__main__":

    main()

```

Trading_GUI.py:

```

import pika          # For communicating with RabbitMQ
import json          # For decoding trade messages
import threading      # To run RabbitMQ listener without freezing the GUI
import tkinter as tk # For creating the GUI
from tkinter import ttk # For using the dropdown (Combobox)


# GUI class to visualize live trades from the 'trades' queue
class TradeGUI:

    """

    TradeGUI is a graphical user interface (GUI) application for monitoring stock trades.

    It connects to a RabbitMQ broker, listens for messages from the 'trades' queue,
    and displays the latest prices and trade history per stock in real-time.
    """

```

```
"""
```

```
def __init__(self, master, rabbitmq_host='localhost', port=5672):
```

```
    """
```

```
    Initializes the GUI layout and sets up background thread for RabbitMQ listening.
```

```
    """
```

```
    self.master = master
```

```
    master.title("Stock Trades Monitor") # Set window title
```

```
    self.latest_prices = {} # Stores the most recent price per stock (e.g., {'XYZ': 49.0})
```

```
    self.trade_history = [] # Stores full trade history received so far
```

```
    # Dropdown for selecting stock
```

```
    self.stock_var = tk.StringVar()
```

```
    self.stock_dropdown = ttk.Combobox(master, textvariable=self.stock_var)
```

```
    self.stock_dropdown.pack(pady=10)
```

```
    self.stock_dropdown.bind("<<ComboboxSelected>>", self.update_display)
```

```
    # Label to show latest price
```

```
    self.price_label = tk.Label(master, text="Latest Price: N/A", font=("Helvetica", 16))
```

```
    self.price_label.pack(pady=10)
```

```
    # Text box to show trade history (last 10 trades)
```

```
    self.history_text = tk.Text(master, height=10, width=50)
```

```
    self.history_text.pack(padx=10, pady=10)
```

```

# Store RabbitMQ connection parameters

self.connection_params = pika.ConnectionParameters(host=rabbitmq_host, port=port,
ssl_options=None)


# Start RabbitMQ listening in background thread (so GUI stays responsive)

threading.Thread(target=self.start_listening, daemon=True).start()


def start_listening(self):
    """
    Connects to RabbitMQ and listens to the 'trades' queue.
    Each trade is processed and the display is updated accordingly.
    """
    try:
        connection = pika.BlockingConnection(self.connection_params)
        channel = connection.channel()

        # Ensure the queue exists
        channel.queue_declare(queue='trades', durable=True)

        # Callback function that runs every time a new trade is received
        def callback(ch, method, properties, body):
            trade = json.loads(body.decode()) # Convert JSON message to Python dict
            stock = trade.get("stock", "XYZ") # Default stock to XYZ if not provided
            price = trade.get("price")

            # Update latest price and add trade to history

```

```

self.latest_prices[stock] = price
self.trade_history.append(trade)

# Update dropdown values with all known stocks
stocks = list(self.latest_prices.keys())
self.master.after(0, lambda: self.stock_dropdown.configure(values=stocks))

# Auto-refresh display if the selected stock matches the new trade's stock
if self.stock_var.get() == stock:
    self.master.after(0, self.update_display)

# Start listening to the queue with the callback
channel.basic_consume(queue='trades', on_message_callback=callback,
auto_ack=True)

print("GUI listening to 'trades' queue...")
channel.start_consuming()

except Exception as e:
    print(f"RabbitMQ connection error: {e}")

def update_display(self, event=None):
    """
    Updates the price label and trade history display for the selected stock.
    Triggered when the dropdown selection changes or a matching trade is received.
    """
    stock = self.stock_var.get()

```

```

if not stock:

    self.price_label.config(text="Latest Price: N/A")

    return

# Show latest price

price = self.latest_prices.get(stock, "N/A")

self.price_label.config(text=f"Latest Price for {stock}: ${price}")

# Filter and show only trades of selected stock (last 10 entries)

trades = [t for t in self.trade_history if t.get("stock") == stock]

self.history_text.delete('1.0', tk.END)

for trade in reversed(trades[-10:]): # Show most recent first

    self.history_text.insert(tk.END, f"{trade['timestamp']} - {trade['buyer']} bought from  

{trade['seller']} at ${trade['price']}\n")

# Entry point to run the GUI

def main():

    root = tk.Tk()

    app = TradeGUI(root)

    root.mainloop()

# Only run main if the script is executed directly

if __name__ == "__main__":

    main()

```

Conclusion

This project helped us understand how middleware like RabbitMQ can be used to design a decoupled and event-driven system. While RabbitMQ was unfamiliar to most of us, we quickly got comfortable with message publishing and consumption patterns. The system successfully processes buy and sell orders, matches them based on defined rules, and outputs trade confirmations. The remaining work mainly involves improving the system's robustness and finalizing the GUI for user interaction or a potential live demo.