Ira A. Fulton Schools of Engineering
Arizona State University

CSE 531: Distributed and Multiprocessor Operating Systems

gRPC Written Report

Problem Statement

In distributed banking systems, customers access their accounts through multiple bank branches, each representing a node in a networked environment. A fundamental challenge in such systems is ensuring data consistency across branches, as updates to shared accounts, such as deposits and withdrawals, need to be reflected promptly and accurately throughout the network. Without robust synchronization, discrepancies in account balances can arise, leading to potential confusion and financial risk for customers.

This project simulates a distributed banking system where branches and customers communicate via gRPC (Google Remote Procedure Call) to maintain synchronized account balances. Each branch holds a replica of a shared account balance and must consistently reflect updates from other branches. Multiple customers, each interacting with a designated branch, initiate transactions such as deposits, withdrawals, and balance inquiries, which are then propagated across branches to ensure uniformity in account information.

Using Python and gRPC, this system is designed to handle transactions sequentially, synchronizing balance updates across all branches after each transaction. The project reads transaction events from an input file, processes them in order, and ensures that each branch maintains an up-to-date balance after each operation. This approach mitigates risks associated with concurrent modifications and enhances the system's reliability. By addressing these challenges, the project demonstrates the practical application of distributed systems principles in ensuring data consistency within a multi-node banking environment.

Goal

The goal of this project is to develop a distributed banking system that allows multiple branches and customers to interact with a shared bank account, ensuring consistent account information across all branches. This system will employ gRPC in Python to facilitate reliable remote communication between customer and branch processes, leveraging techniques such as sequential transaction processing and state propagation across branch replicas to maintain data consistency.

Specifically, the project will implement key interfaces—Deposit, Withdraw, and Query—to manage customer requests, along with Propagate_Deposit and Propagate_Withdraw interfaces for synchronizing balances between branches. Each branch will handle customer transactions individually while reflecting updates in real time across other branches to prevent data discrepancies. This design will process events from a predefined input file, ensuring each transaction is correctly applied and balance information remains coherent across the network.

Beyond achieving a functional distributed banking model, this project aims to deepen our understanding of fundamental distributed systems principles, inter-process communication, and the challenges of maintaining data integrity in multi-node environments. The project also provides practical experience with tools and concepts essential to distributed system design, including defining services in .proto files, using the protocol buffer compiler, and implementing a gRPC-based client-server architecture. By completing this project, we gain hands-on experience in building and managing distributed applications, applying synchronization techniques, and using modern development tools for reliable, scalable communication.

Setup

I. Integrated Development Environment (IDE)

For this project, I will use Visual Studio Code (VS Code), a free, open-source IDE known for its versatility and robust support for multiple programming languages. VS Code features built-in terminals, easy installation of extensions and plugins, and efficient library management, making it suitable for projects of this scale and complexity. To download and install VS Code, refer to its official website [1].

II. Python Installation

The Python programming language is required to implement this project, so it is crucial to ensure it is installed correctly. For this project, I will be using a Mac. While some Macs come with Python pre-installed, the included version may be outdated or default to python3. You may have difficulty to run some required “python” commands (e.g., “python server.py input.json” and “python client.py input.json”). If you encounter a “command not found” error in the terminal, you can refer to the troubleshooting guide provided by Daniel Kehoe on his website [2].

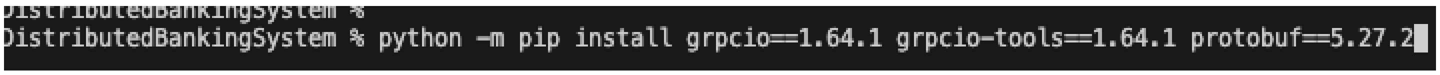
As recommended by Daniel Kehoe [3], it is unnecessary to update or remove the system Python that comes pre-installed with macOS or the Xcode Command Line Tools. Instead, we can install the latest version of Python directly from its official website [4]. Additionally, there is no need to worry about manually installing pip, as it is included by default in Python installers for versions 3.4 and above. Once Python is installed, ensure its path is correctly added to your system’s bash configuration file.

This step ensures that Python scripts can be executed without errors, and the python command functions as required by the project.

III. Project Setup

After successfully installing Python and VS Code, the next step is to set up the project environment. Begin by creating a directory named DistributedBankingSystem, which will serve as the workspace for the project files. Within this directory, copy the provided input.json file along with the template files for customer.py and branch.py.

Once the files are in place, install the required Python packages to enable gRPC communication and protocol buffer functionality. These packages include grpcio, grpcio-tools, and protobuf, all of which are specified in the project requirements. The command to run in the terminal is shown below.



```
DistributedBankingSystem %
DistributedBankingSystem % python -m pip install grpcio==1.64.1 grpcio-tools==1.64.1 protobuf==5.27.2
```

Implementation Processes

I. System End-to-End Workflow

With the directory organized and dependencies installed, the implementation of the distributed banking system can proceed. A clear understanding of the system architecture and the relationships between the various files is essential before implementation. Different interpretations of the project requirements may result in varying designs and implementations. Based on my interpretation of the requirements, I will outline the system's end-to-end functionality and the details of its design and implementation.

To begin, the server processes must be initialized by running the command “python server.py input.json.” Upon execution, server.py reads the input.json file and identifies elements with the "type" attribute set to "branch." For each branch element, the server creates a corresponding process, using branch.py to initialize these branch processes based on their unique identifiers (id) specified in the input file.

Next, the client processes are launched using the command “python client.py input.json.” This step also reads the input.json file and extracts elements where the "type" is "customer." Each customer element corresponds to a unique customer identified by an id, with an associated "events" list that contains the transactions to be performed by that customer. The createStub() and executeEvents() methods in customer.py are called to establish connections between the customer processes and their corresponding branch processes. The method customer.executeEvents() will sequentially process the transactions listed in each customer's "events."

The executeEvents() method in customer.py handles three types of transactions: deposit, withdraw, and query, as specified in the "interface" attribute of each event. To process these transactions, it calls the MsgDelivery() method in branch.py. Acting as a central hub, branch.MsgDelivery() receives the transaction requests from customer.executeEvents() and delegates them to the appropriate methods for further processing. For instance, deposit transactions invoke branch.Deposit(), withdrawals call branch.Withdraw(), and balance inquiries are handled by branch.Query().

The Deposit() and Withdraw() methods not only update the balance for the respective branch process but also propagate these changes to other branch processes to ensure consistency. This propagation is handled through the Propagate_Deposit() and Propagate_Withdraw() methods, which are called within MsgDelivery().

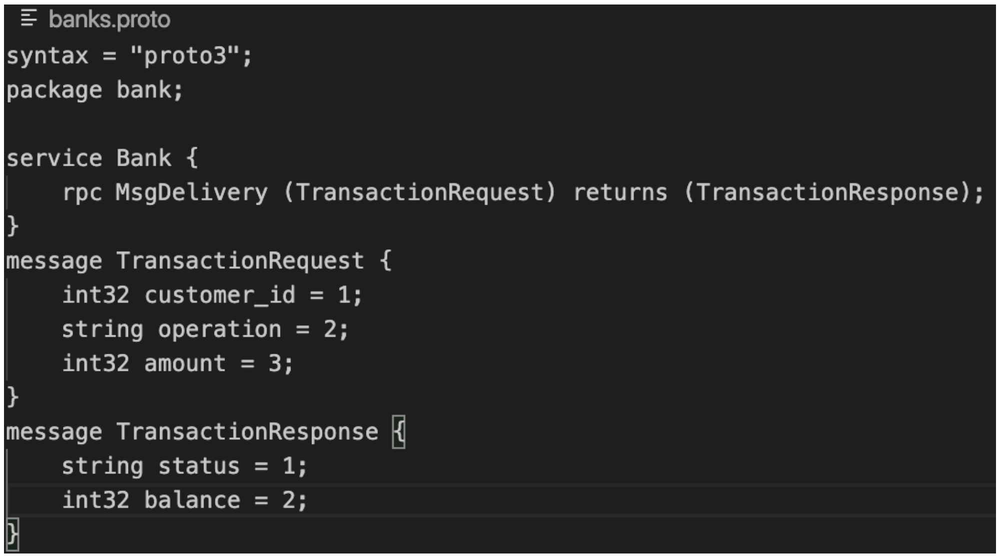
After processing the transactions, customer.executeEvents() receives responses from the branch processes. These responses are formatted into a structured JSON output and returned to client.py. The client.py consolidates these responses and generates the output.json file, which contains the results of all transaction events.

II. Implementation

In summary, the system operates as follows: the server initializes branch processes, and the client initiates transaction requests that cascade through the system, invoking methods at each layer. Responses travel back up the layers, culminating in the generation of a comprehensive output file. To facilitate seamless communication across these layers, gRPC is employed. A protos directory is created within the project directory, containing the banks.proto file, which defines the gRPC service. The proto file specifies a single RPC method, MsgDelivery, fulfilling the requirement for branch.MsgDelivery() to serve as the central mechanism for routing all transaction requests to their corresponding methods.

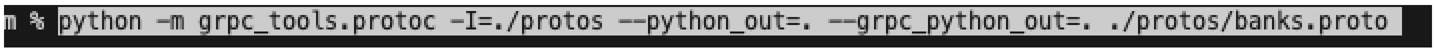
The detailed implementation of banks.proto is below. The third attribute “amount” in TransactionRequest is optional for deposit, withdraw, propagate_deposit, and propagate_withdraw transactions. The second attribute “balance” in TransactionResponse is optional for query transactions.

(Continued on the next page)



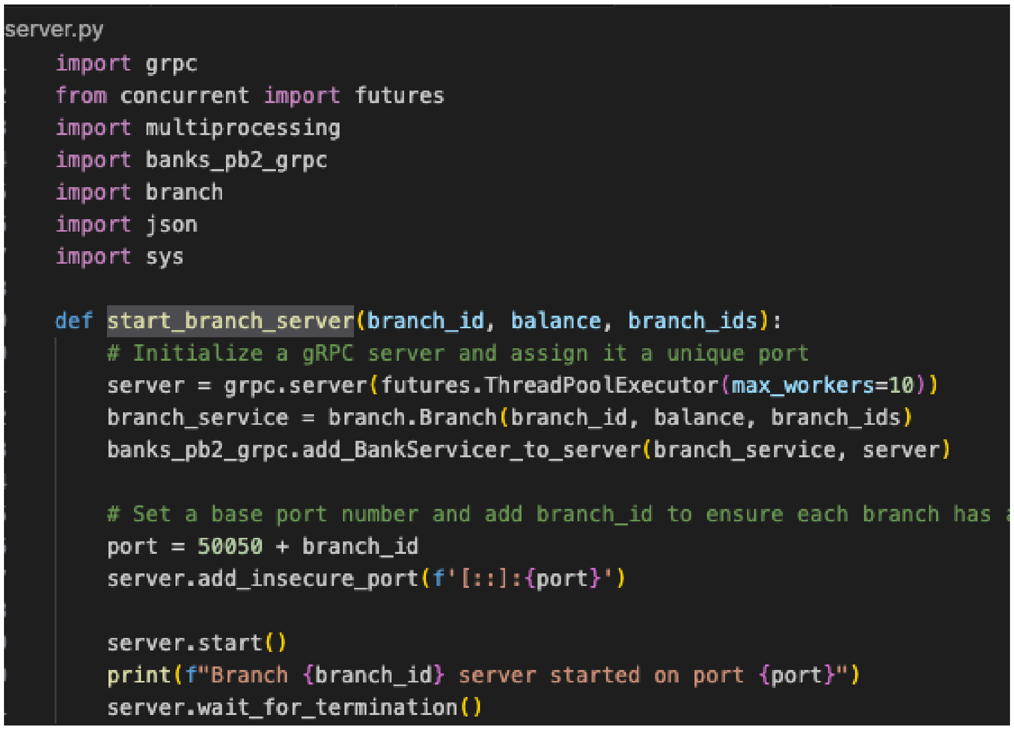
```
banks.proto
syntax = "proto3";
package bank;

service Bank {
    rpc MsgDelivery (TransactionRequest) returns (TransactionResponse);
}
message TransactionRequest {
    int32 customer_id = 1;
    string operation = 2;
    int32 amount = 3;
}
message TransactionResponse {
    string status = 1;
    int32 balance = 2;
}
```

After implementing the proto file, run the following command to generate banks_pb2_grpc.py and banks_pb2.py. These generated language-specific files contain service stubs, eliminating the need to manually write code for data serialization, deserialization, and RPC communication.

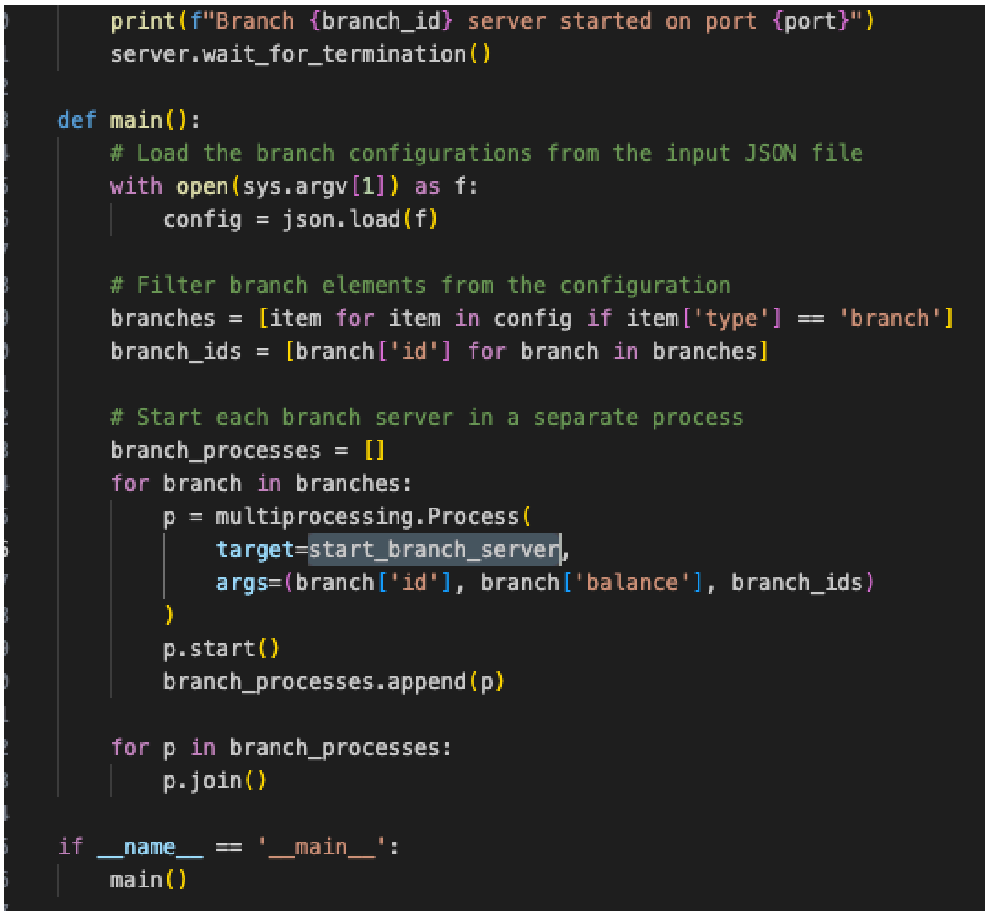


```
m % python -m grpc_tools.protoc -I=./protos --python_out=. --grpc_python_out=. ./protos/banks.proto
```

Next, create a file named server.py to handle two primary tasks: processing the data from input.json and initializing server processes by invoking branch.py. The detailed implementation is shown below.

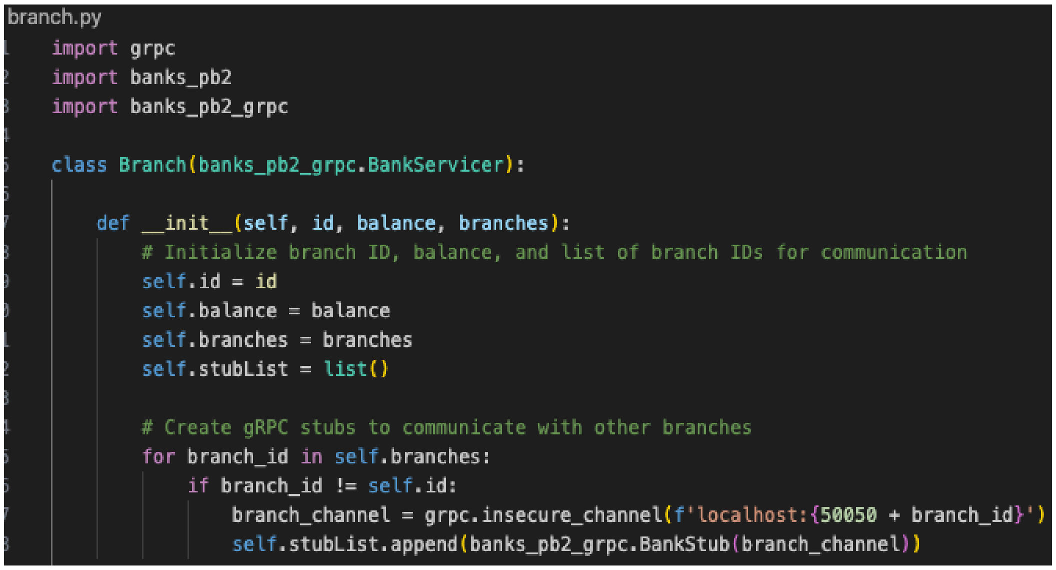


```
server.py
import grpc
from concurrent import futures
import multiprocessing
import banks_pb2_grpc
import branch
import json
import sys

def start_branch_server(branch_id, balance, branch_ids):
    # Initialize a gRPC server and assign it a unique port
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    branch_service = branch.Branch(branch_id, balance, branch_ids)
    banks_pb2_grpc.add_BankServicer_to_server(branch_service, server)

    # Set a base port number and add branch_id to ensure each branch has
    port = 50050 + branch_id
    server.add_insecure_port(f'[::]:{port}')

    server.start()
    print(f"Branch {branch_id} server started on port {port}")
    server.wait_for_termination()
```



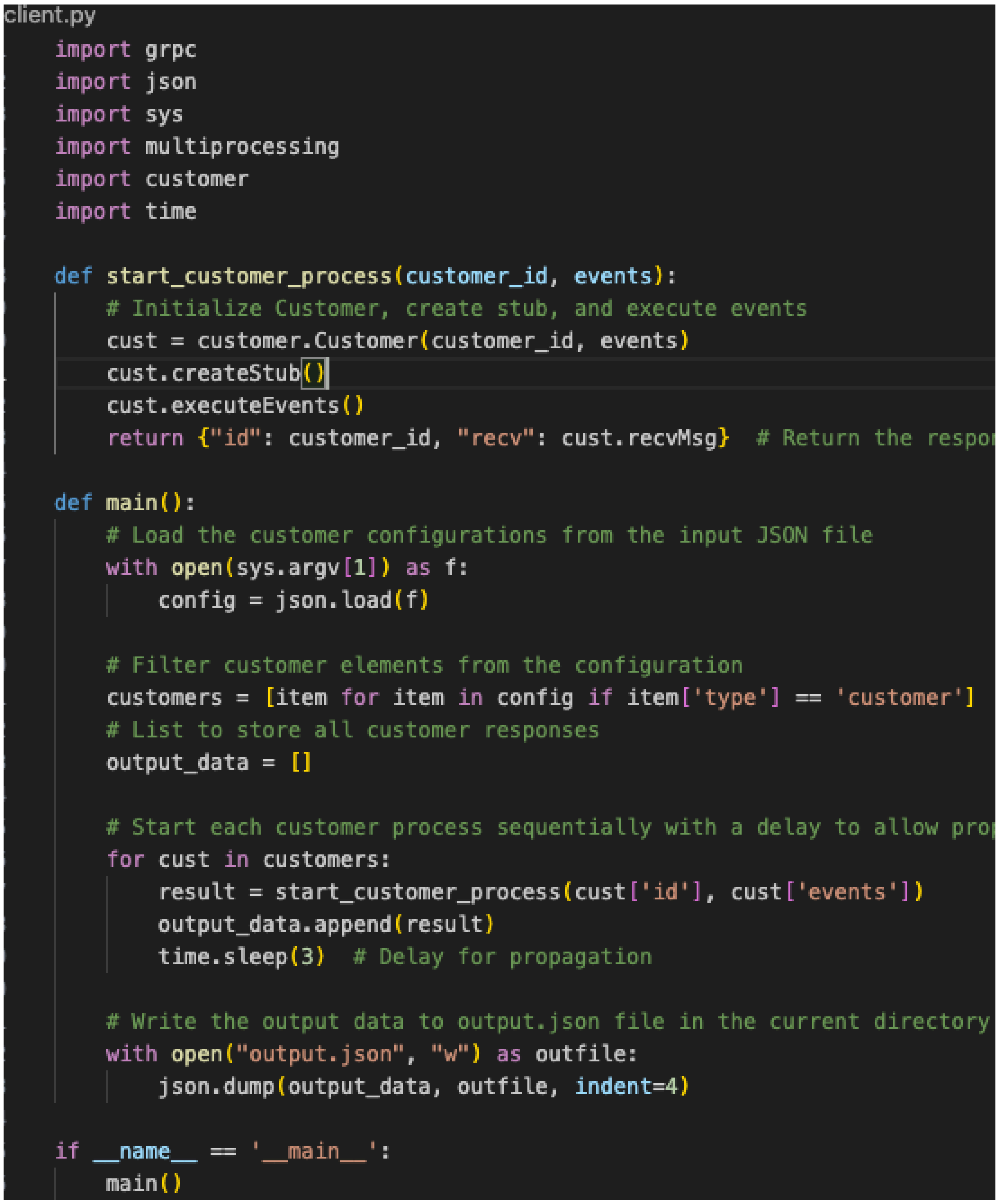
```python
        print(f"Branch {branch_id} server started on port {port}")
        server.wait_for_termination()

def main():
    # Load the branch configurations from the input JSON file
    with open(sys.argv[1]) as f:
        config = json.load(f)

    # Filter branch elements from the configuration
    branches = [item for item in config if item['type'] == 'branch']
    branch_ids = [branch['id'] for branch in branches]

    # Start each branch server in a separate process
    branch_processes = []
    for branch in branches:
        p = multiprocessing.Process(
            target=start_branch_server,
            args=(branch['id'], branch['balance'], branch_ids)
        )
        p.start()
        branch_processes.append(p)

    for p in branch_processes:
        p.join()

if __name__ == '__main__':
    main()
```

Since server.py needs to invoke branch.py to create branch processes, the next step is to implement the initializer in branch.py. As the project simulates a distributed banking system, it will run locally. To ensure that each branch process operates on a unique port, the port number will be assigned as {50050 + branch_id}.



```python
branch.py
import grpc
import banks_pb2
import banks_pb2_grpc

class Branch(banks_pb2_grpc.BankServicer):

    def __init__(self, id, balance, branches):
        # Initialize branch ID, balance, and list of branch IDs for communication
        self.id = id
        self.balance = balance
        self.branches = branches
        self.stubList = list()

        # Create gRPC stubs to communicate with other branches
        for branch_id in self.branches:
            if branch_id != self.id:
                branch_channel = grpc.insecure_channel(f'localhost:{50050 + branch_id}')
                self.stubList.append(banks_pb2_grpc.BankStub(branch_channel))
```

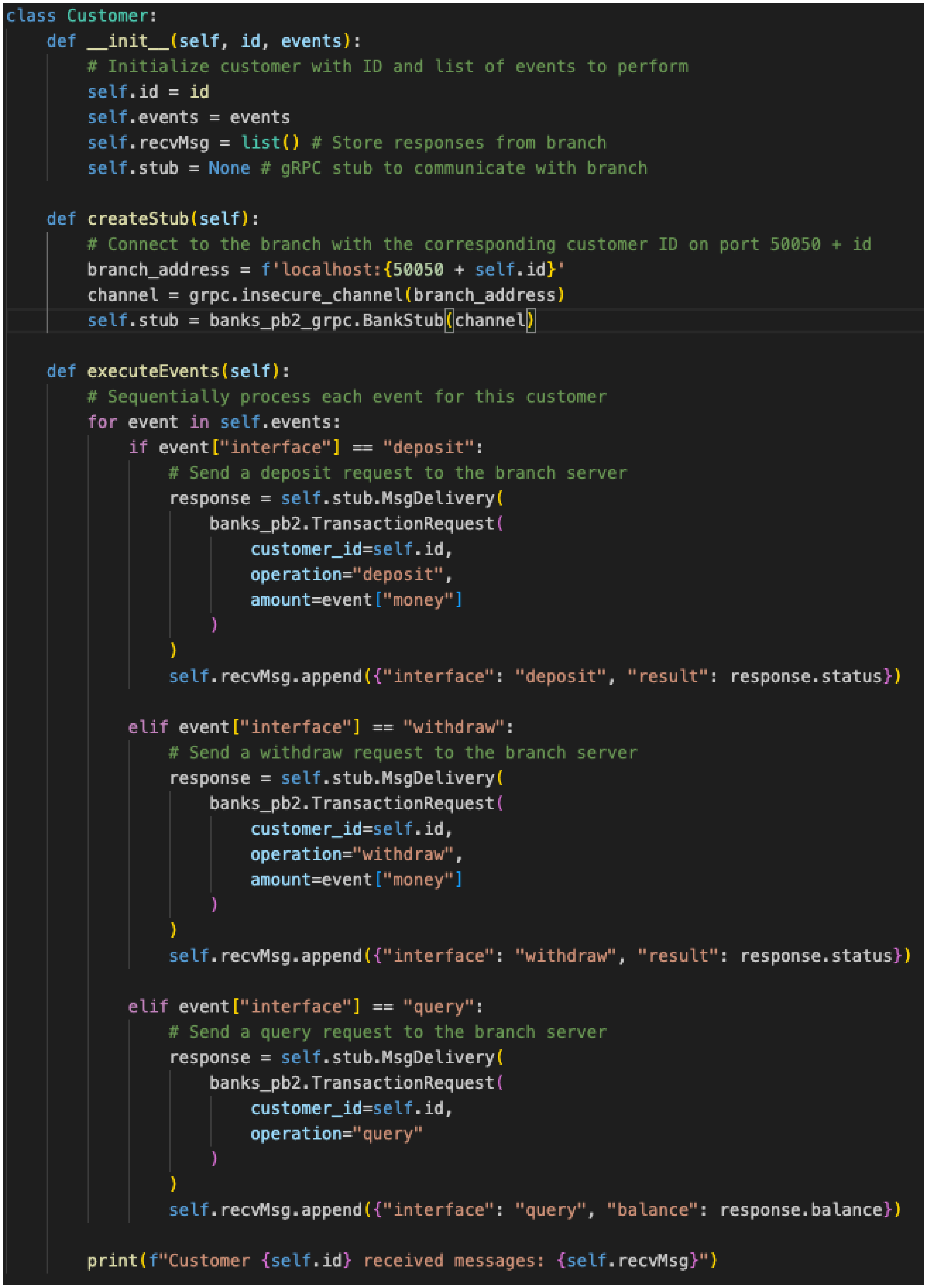
With the server implementation complete, we can proceed to create client.py. The client.py performs three primary tasks: first, it processes data from input.json; second, it invokes the createStub() and executeEvents() methods in customer.py to sequentially handle each customer's events list; and finally, it generates the output.json file containing the received responses. The detailed implementation is shown in the screenshot below.



client.py

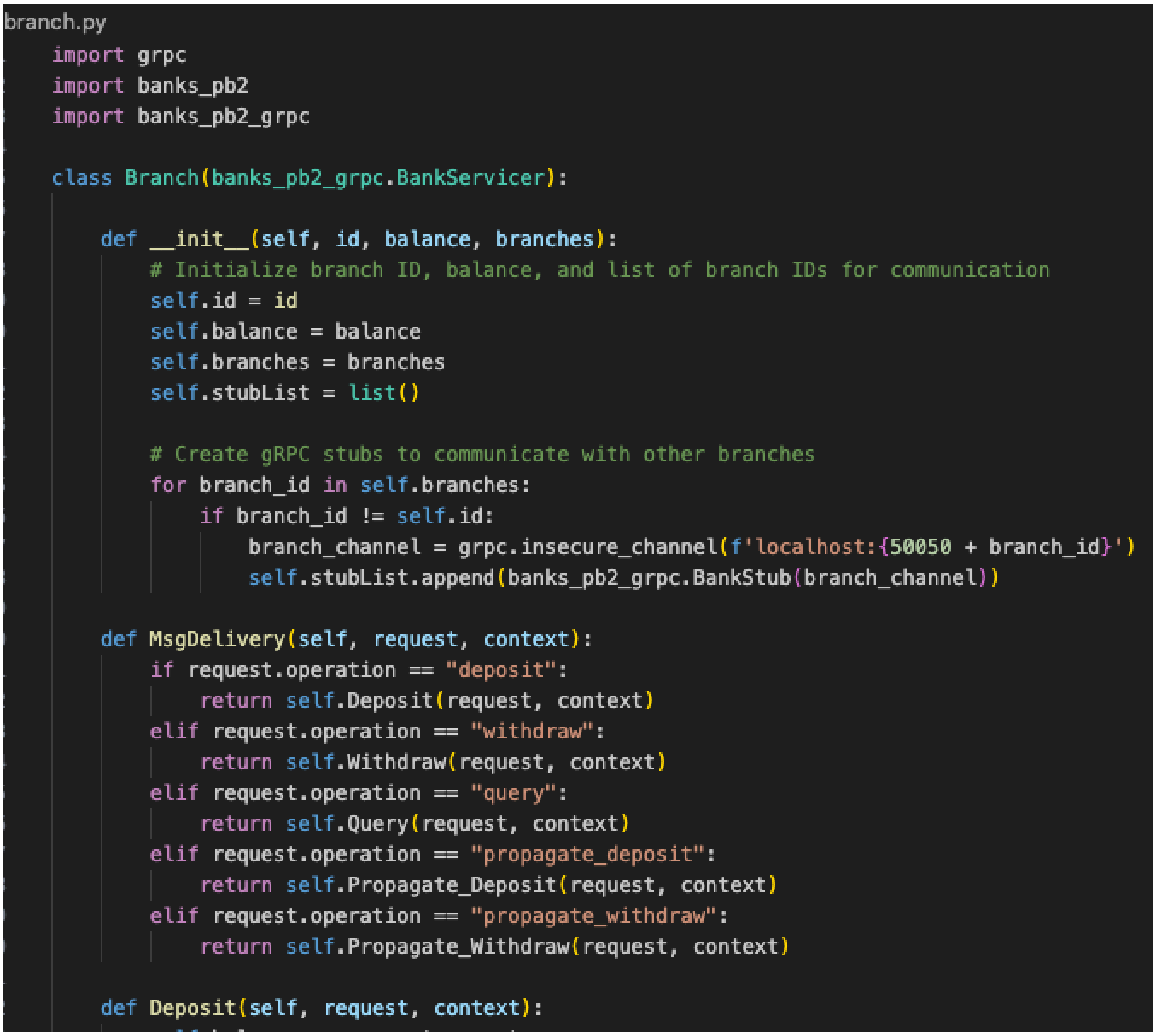
```python
import grpc
import json
import sys
import multiprocessing
import customer
import time

def start_customer_process(customer_id, events):
    # Initialize Customer, create stub, and execute events
    cust = customer.Customer(customer_id, events)
    cust.createStub()
    cust.executeEvents()
    return {"id": customer_id, "recv": cust.recvMsg}  # Return the respon

def main():
    # Load the customer configurations from the input JSON file
    with open(sys.argv[1]) as f:
        config = json.load(f)

    # Filter customer elements from the configuration
    customers = [item for item in config if item['type'] == 'customer']
    # List to store all customer responses
    output_data = []

    # Start each customer process sequentially with a delay to allow prop
    for cust in customers:
        result = start_customer_process(cust['id'], cust['events'])
        output_data.append(result)
        time.sleep(3)  # Delay for propagation

    # Write the output data to output.json file in the current directory
    with open("output.json", "w") as outfile:
        json.dump(output_data, outfile, indent=4)

if __name__ == '__main__':
    main()
```

Since client.py invokes the createStub() and executeEvents() methods from customer.py, the next step is to complete the implementation of these two methods.



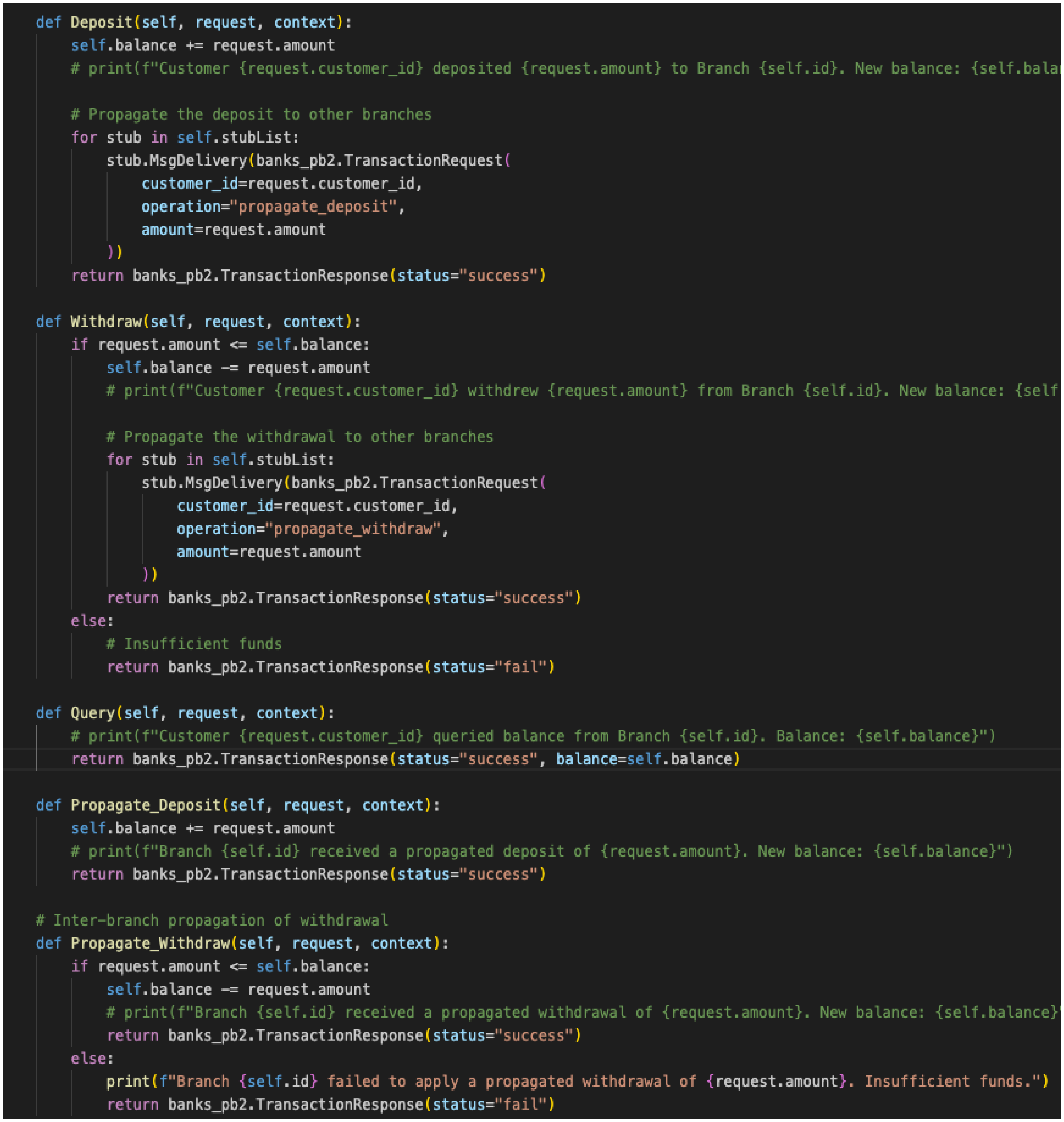
```python
class Customer:
    def __init__(self, id, events):
        # Initialize customer with ID and list of events to perform
        self.id = id
        self.events = events
        self.recvMsg = list() # Store responses from branch
        self.stub = None # gRPC stub to communicate with branch

    def createStub(self):
        # Connect to the branch with the corresponding customer ID on port 50050 + id
        branch_address = f'localhost:{50050 + self.id}'
        channel = grpc.insecure_channel(branch_address)
        self.stub = banks_pb2_grpc.BankStub(channel)

    def executeEvents(self):
        # Sequentially process each event for this customer
        for event in self.events:
            if event["interface"] == "deposit":
                # Send a deposit request to the branch server
                response = self.stub.MsgDelivery(
                    banks_pb2.TransactionRequest(
                        customer_id=self.id,
                        operation="deposit",
                        amount=event["money"]
                    )
                )
                self.recvMsg.append({"interface": "deposit", "result": response.status})

            elif event["interface"] == "withdraw":
                # Send a withdraw request to the branch server
                response = self.stub.MsgDelivery(
                    banks_pb2.TransactionRequest(
                        customer_id=self.id,
                        operation="withdraw",
                        amount=event["money"]
                    )
                )
                self.recvMsg.append({"interface": "withdraw", "result": response.status})

            elif event["interface"] == "query":
                # Send a query request to the branch server
                response = self.stub.MsgDelivery(
                    banks_pb2.TransactionRequest(
                        customer_id=self.id,
                        operation="query"
                    )
                )
                self.recvMsg.append({"interface": "query", "balance": response.balance})

        print(f"Customer {self.id} received messages: {self.recvMsg}")
```

As demonstrated above, the executeEvents() method in customer.py prepares the necessary TransactionRequest() and invokes MsgDelivery() from branch.py to process all transactions, establishing the required communication between the customer and branch interfaces. The branch.MsgDelivery() method subsequently calls its class methods to handle specific transaction types. For deposit and withdrawal transactions, MsgDelivery() is invoked again within these methods to propagate changes to other branches, facilitating the required branch-to-branch communication. The detailed implementation of branch.py is shown below.



branch.py

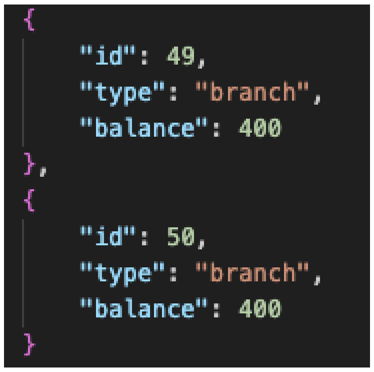
```python
import grpc
import banks_pb2
import banks_pb2_grpc

class Branch(banks_pb2_grpc.BankServicer):

    def __init__(self, id, balance, branches):
        # Initialize branch ID, balance, and list of branch IDs for communication
        self.id = id
        self.balance = balance
        self.branches = branches
        self.stubList = list()

        # Create gRPC stubs to communicate with other branches
        for branch_id in self.branches:
            if branch_id != self.id:
                branch_channel = grpc.insecure_channel(f'localhost:{50050 + branch_id}')
                self.stubList.append(banks_pb2_grpc.BankStub(branch_channel))

    def MsgDelivery(self, request, context):
        if request.operation == "deposit":
            return self.Deposit(request, context)
        elif request.operation == "withdraw":
            return self.Withdraw(request, context)
        elif request.operation == "query":
            return self.Query(request, context)
        elif request.operation == "propagate_deposit":
            return self.Propagate_Deposit(request, context)
        elif request.operation == "propagate_withdraw":
            return self.Propagate_Withdraw(request, context)

    def Deposit(self, request, context):
```


```
def Deposit(self, request, context):
    self.balance += request.amount
    # print(f"Customer {request.customer_id} deposited {request.amount} to Branch {self.id}. New balance: {self.bala

    # Propagate the deposit to other branches
    for stub in self.stubList:
        stub.MsgDelivery(banks_pb2.TransactionRequest(
            customer_id=request.customer_id,
            operation="propagate_deposit",
            amount=request.amount
        ))
    return banks_pb2.TransactionResponse(status="success")

def Withdraw(self, request, context):
    if request.amount <= self.balance:
        self.balance -= request.amount
        # print(f"Customer {request.customer_id} withdrew {request.amount} from Branch {self.id}. New balance: {self

        # Propagate the withdrawal to other branches
        for stub in self.stubList:
            stub.MsgDelivery(banks_pb2.TransactionRequest(
                customer_id=request.customer_id,
                operation="propagate_withdraw",
                amount=request.amount
            ))
        return banks_pb2.TransactionResponse(status="success")
    else:
        # Insufficient funds
        return banks_pb2.TransactionResponse(status="fail")

def Query(self, request, context):
    # print(f"Customer {request.customer_id} queried balance from Branch {self.id}. Balance: {self.balance}")
    return banks_pb2.TransactionResponse(status="success", balance=self.balance)

def Propagate_Deposit(self, request, context):
    self.balance += request.amount
    # print(f"Branch {self.id} received a propagated deposit of {request.amount}. New balance: {self.balance}")
    return banks_pb2.TransactionResponse(status="success")

# Inter-branch propagation of withdrawal
def Propagate_Withdraw(self, request, context):
    if request.amount <= self.balance:
        self.balance -= request.amount
        # print(f"Branch {self.id} received a propagated withdrawal of {request.amount}. New balance: {self.balance}
        return banks_pb2.TransactionResponse(status="success")
    else:
        print(f"Branch {self.id} failed to apply a propagated withdrawal of {request.amount}. Insufficient funds.")
        return banks_pb2.TransactionResponse(status="fail")
```

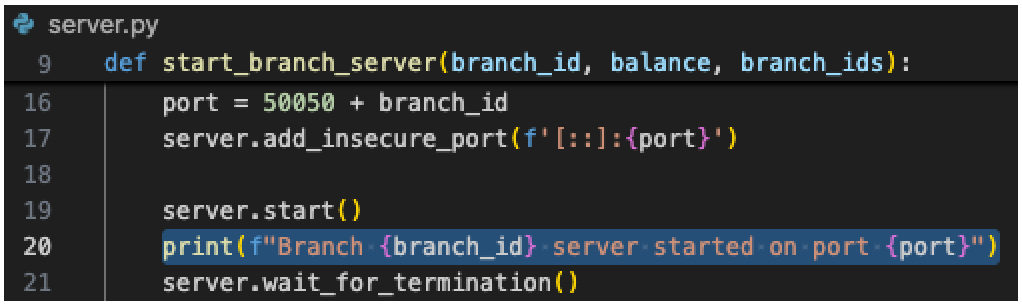
Results

I. Start Server

At this stage, the implementation of all required files is complete, and we can proceed to run the program. Begin by opening a terminal and executing the command “python server.py input.json.” This command processes the "branch" elements from input.json, as shown in the first screenshot below. The corresponding code in server.py (highlighted at line 20 in the second screenshot) helps print out which branch ID is assigned to which port. The terminal logs generated after running the command are displayed in the final image below. From the logs, we can confirm that all 50 "branch" elements are successfully assigned to unique ports.

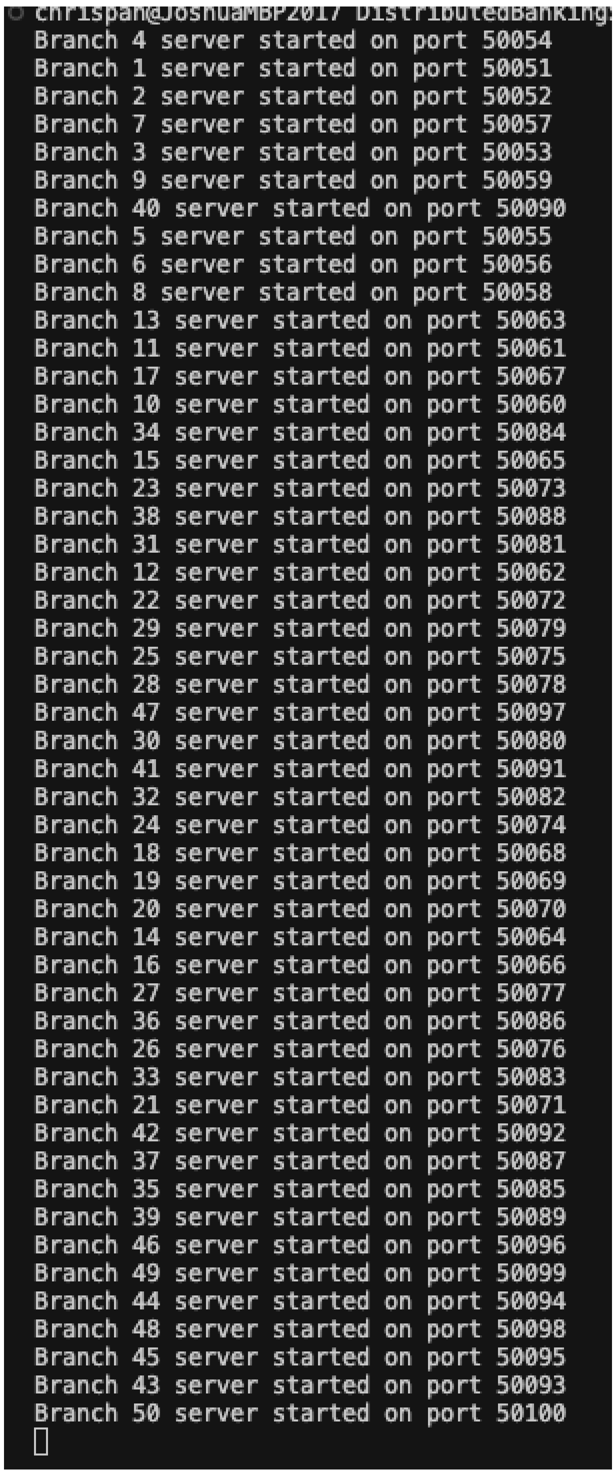


```
{
    "id": 49,
    "type": "branch",
    "balance": 400
},
{
    "id": 50,
    "type": "branch",
    "balance": 400
}
```



```
server.py
 9   def start_branch_server(branch_id, balance, branch_ids):
16       port = 50050 + branch_id
17       server.add_insecure_port(f'[::]:{port}')
18
19       server.start()
20       print(f"Branch {branch_id} server started on port {port}")
21       server.wait_for_termination()
```

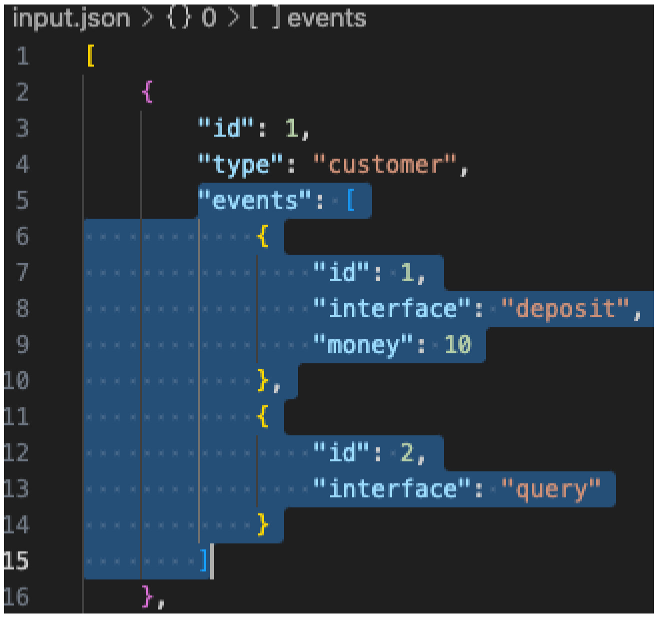
(Continued on the next page)



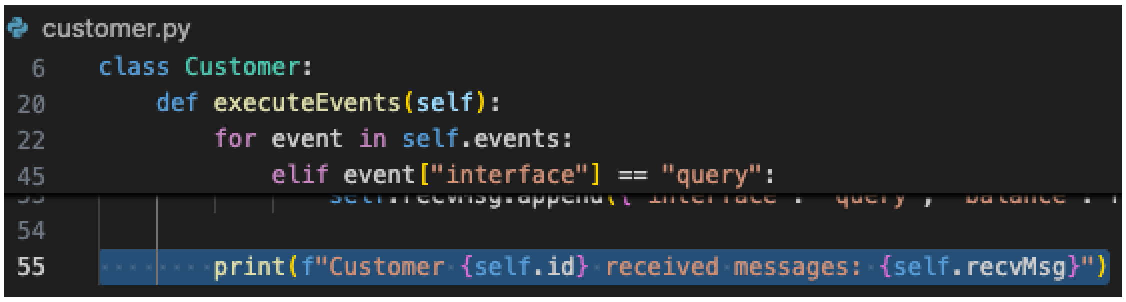
```
chrispan@JoshuaMBP2017 DistributedBanking
  Branch 4 server started on port 50054
  Branch 1 server started on port 50051
  Branch 2 server started on port 50052
  Branch 7 server started on port 50057
  Branch 3 server started on port 50053
  Branch 9 server started on port 50059
  Branch 40 server started on port 50090
  Branch 5 server started on port 50055
  Branch 6 server started on port 50056
  Branch 8 server started on port 50058
  Branch 13 server started on port 50063
  Branch 11 server started on port 50061
  Branch 17 server started on port 50067
  Branch 10 server started on port 50060
  Branch 34 server started on port 50084
  Branch 15 server started on port 50065
  Branch 23 server started on port 50073
  Branch 38 server started on port 50088
  Branch 31 server started on port 50081
  Branch 12 server started on port 50062
  Branch 22 server started on port 50072
  Branch 29 server started on port 50079
  Branch 25 server started on port 50075
  Branch 28 server started on port 50078
  Branch 47 server started on port 50097
  Branch 30 server started on port 50080
  Branch 41 server started on port 50091
  Branch 32 server started on port 50082
  Branch 24 server started on port 50074
  Branch 18 server started on port 50068
  Branch 19 server started on port 50069
  Branch 20 server started on port 50070
  Branch 14 server started on port 50064
  Branch 16 server started on port 50066
  Branch 27 server started on port 50077
  Branch 36 server started on port 50086
  Branch 26 server started on port 50076
  Branch 33 server started on port 50083
  Branch 21 server started on port 50071
  Branch 42 server started on port 50092
  Branch 37 server started on port 50087
  Branch 35 server started on port 50085
  Branch 39 server started on port 50089
  Branch 46 server started on port 50096
  Branch 49 server started on port 50099
  Branch 44 server started on port 50094
  Branch 48 server started on port 50098
  Branch 45 server started on port 50095
  Branch 43 server started on port 50093
  Branch 50 server started on port 50100
```

II. Start Client-Side

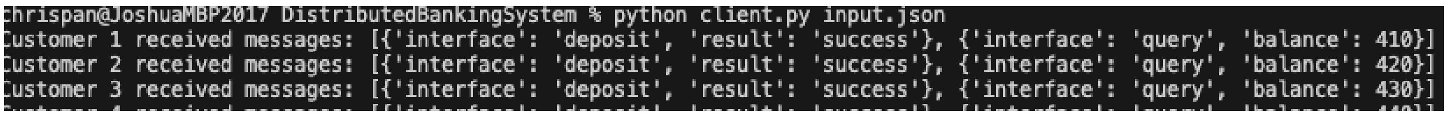
Once all server processes are running, a new terminal can be opened to execute the client file. The JSON elements with the type "customer" (an example is shown in the first screenshot below) will be extracted, and the events within each customer's "events" list will be processed sequentially (highlighted in the same screenshot). The command to execute is "python client.py input.json." After running this command, helpful log messages will be displayed in the terminal, printed by line 55 of customer.py (shown in second image below). These logs include JSON elements that are written into output.json for each processed "events" list. The last three screenshots provide parts of the terminal loggings.


input.json > {} 0 > [ ] events

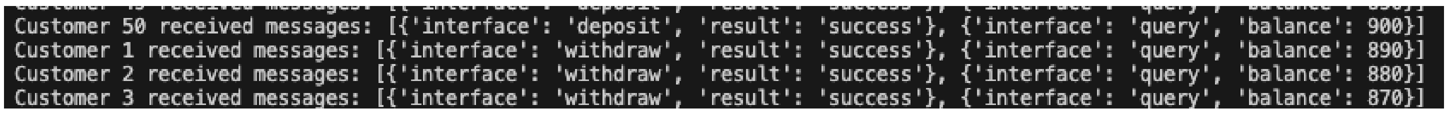
```
[
    {
        "id": 1,
        "type": "customer",
        "events": [
            {
                "id": 1,
                "interface": "deposit",
                "money": 10
            },
            {
                "id": 2,
                "interface": "query"
            }
        ]
    },
```


customer.py

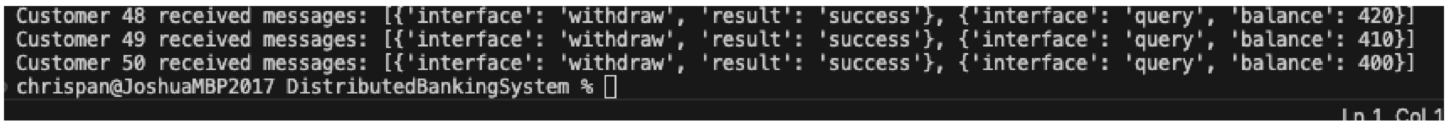
```
6    class Customer:
20       def executeEvents(self):
22           for event in self.events:
45               elif event["interface"] == "query":
54
55           print(f"Customer {self.id} received messages: {self.recvMsg}")
```



```
chrispan@JoshuaMBP2017 DistributedBankingSystem % python client.py input.json
Customer 1 received messages: [{'interface': 'deposit', 'result': 'success'}, {'interface': 'query', 'balance': 410}]
Customer 2 received messages: [{'interface': 'deposit', 'result': 'success'}, {'interface': 'query', 'balance': 420}]
Customer 3 received messages: [{'interface': 'deposit', 'result': 'success'}, {'interface': 'query', 'balance': 430}]
```



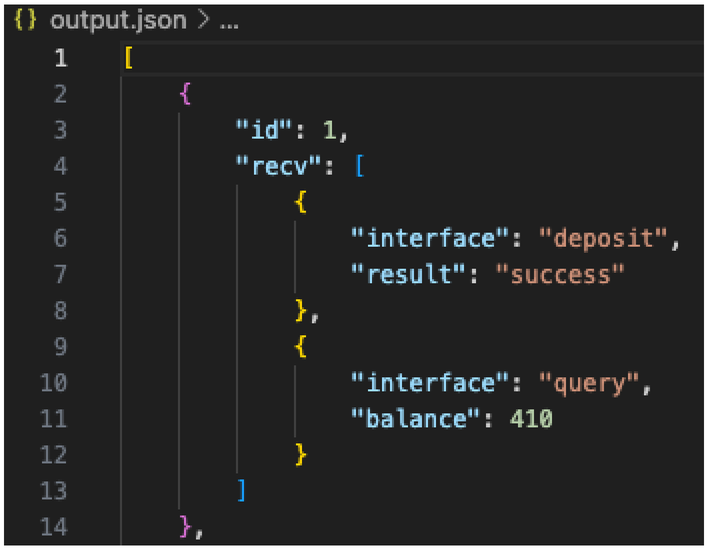
```
Customer 50 received messages: [{'interface': 'deposit', 'result': 'success'}, {'interface': 'query', 'balance': 900}]
Customer 1 received messages: [{'interface': 'withdraw', 'result': 'success'}, {'interface': 'query', 'balance': 890}]
Customer 2 received messages: [{'interface': 'withdraw', 'result': 'success'}, {'interface': 'query', 'balance': 880}]
Customer 3 received messages: [{'interface': 'withdraw', 'result': 'success'}, {'interface': 'query', 'balance': 870}]
```



```
Customer 48 received messages: [{'interface': 'withdraw', 'result': 'success'}, {'interface': 'query', 'balance': 420}]
Customer 49 received messages: [{'interface': 'withdraw', 'result': 'success'}, {'interface': 'query', 'balance': 410}]
Customer 50 received messages: [{'interface': 'withdraw', 'result': 'success'}, {'interface': 'query', 'balance': 400}]
chrispan@JoshuaMBP2017 DistributedBankingSystem % 
```

III. Final Result

After processing all transactions, client.py writes the formatted responses into output.json. Since the full output file is too lengthy, only a small portion is included here for reference. The first screenshot below displays the first JSON element in the output file, indicating that the customer with "id": 1 successfully completed two requests. The first was a deposit transaction, which was processed successfully, followed by a balance query that returned a balance of "410".

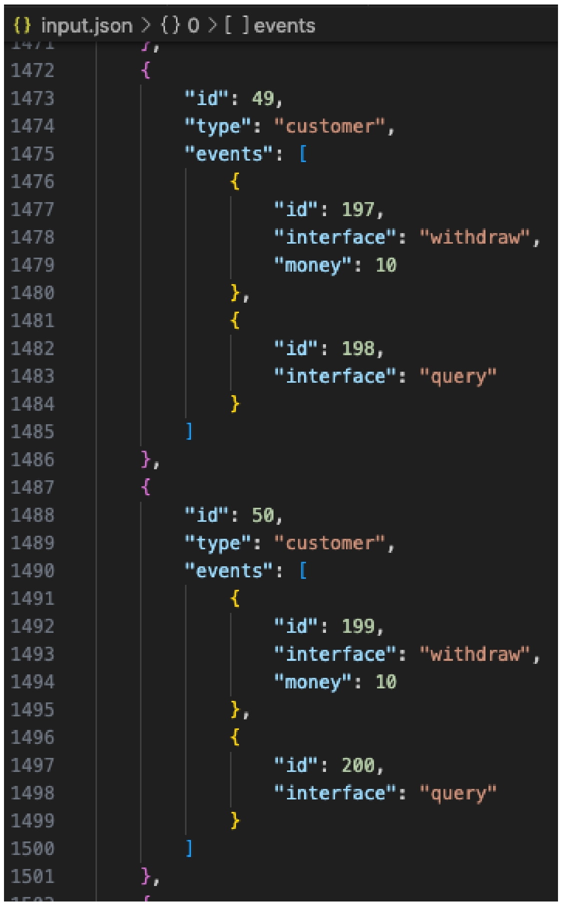


```
[
    {
        "id": 1,
        "recv": [
            {
                "interface": "deposit",
                "result": "success"
            },
            {
                "interface": "query",
                "balance": 410
            }
        ]
    },
```

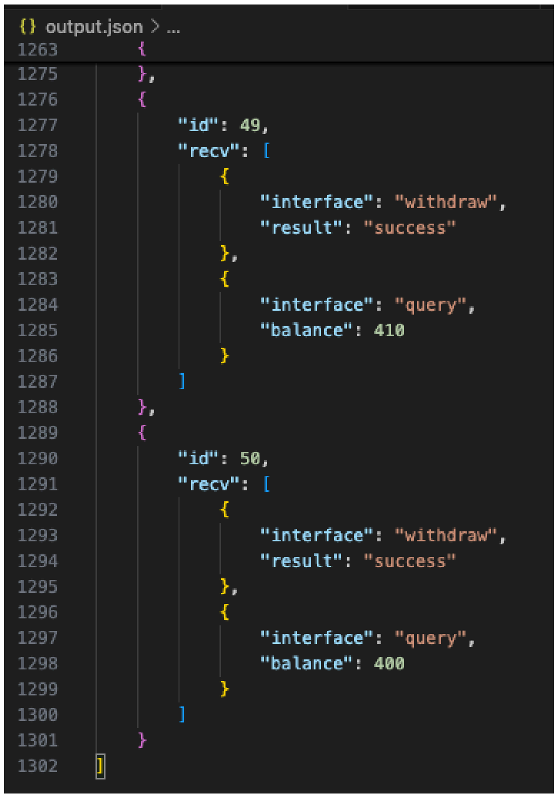
Let us consider another example using both input.json and output.json. The first screenshot below from the input.json file shows the events lists for customers with "id": 49 and "id": 50. Customer 49 performs two transaction requests: a withdrawal of 10 from the balance, followed by a query request. From the output (second screenshot), we see that after the successful withdrawal, the current balance is 410. This balance is consistent across all branch processes, ensuring that subsequent customer requests operate on a unified state with balance = 410.

From the input.json file, we observe that the customer with "id": 50 initiates a withdrawal request for an amount of 10. Since this customer starts with a balance of 410, the next query request should return 400 (calculated as 410 - 10). The corresponding output confirms that the withdrawal is successful, and the final balance queried is indeed 400, validating the correctness of the system's functionality.

(Continued on the next page)



```
        },
        {
            "id": 49,
            "type": "customer",
            "events": [
                {
                    "id": 197,
                    "interface": "withdraw",
                    "money": 10
                },
                {
                    "id": 198,
                    "interface": "query"
                }
            ]
        },
        {
            "id": 50,
            "type": "customer",
            "events": [
                {
                    "id": 199,
                    "interface": "withdraw",
                    "money": 10
                },
                {
                    "id": 200,
                    "interface": "query"
                }
            ]
        },
```



```
    {
    },
    {
        "id": 49,
        "recv": [
            {
                "interface": "withdraw",
                "result": "success"
            },
            {
                "interface": "query",
                "balance": 410
            }
        ]
    },
    {
        "id": 50,
        "recv": [
            {
                "interface": "withdraw",
                "result": "success"
            },
            {
                "interface": "query",
                "balance": 400
            }
        ]
    }
]
```

REFERENCES

[1] Visual Studio Code. “Code faster with AI.” Visual Studio Code. Accessed: Nov. 14, 2024. [Online]. Available: https://code.visualstudio.com/

[2] D. Kehoe. “Command not found: python.” Python. Accessed: Nov. 14, 2024. [Online]. Available: https://mac.install.guide/python/command-not-found-python#:~:text=New%20users%20of%20Python%20on,and%20set%20the%20Mac%20PATH.

[3] D. Kehoe. “Install Python with Rye.” Python. Accessed: Nov. 14, 2024. [Online]. Available: https://mac.install.guide/python/install-rye

[4] Python. “Download the latest version for macOS.” Python. Accessed: Nov. 14, 2024. [Online]. Available: https://www.python.org/downloads/