Two Phase Distributed Commit Implementation

Objectives:

- To implement algorithm for two phase distributed commit

Environmental Setup:

- We'll use Python's `sockets` library to simulate communication between processes.

Theory:

The Two-Phase Commit algorithm allows multi-partition atomic updates. It ensures that either all nodes commit or all nodes abort. This pattern may be used by any applications, but it's most widely discussed in the context of databases. The Two-Phase Commit protocol is the most straightforward and because of that most widely used when atomic transactions are required. This algorithm has two concepts:

- coordinator (sometimes called transaction manager) — node the transaction originates
- subordinates (sometimes called participants) — nodes the sub-transactions are executed on

Advantages of Two-Phase Distributed Commit:

- Data Consistency:
  The primary advantage of the Two-Phase Commit (2PC) protocol is its ability to maintain data consistency across distributed systems. By ensuring that all participants either commit or abort the transaction together, 2PC prevents partial updates and keeps the system's state synchronized.

- Controlled Transaction Management:
  2PC provides a controlled approach to transaction management. The coordinator has the authority to decide whether to commit or abort based on the responses from all participants, allowing for centralized control of distributed transactions.

Disadvantages of Two-Phase Distributed Commit:

- Communication Overhead:
  2PC involves multiple rounds of communication between the coordinator and participants. This can introduce significant latency, especially in systems with a large number of participants or in environments with high network latency.

- Blocking Problem:
  If the coordinator fails after sending the Prepare request but before sending the Commit or Abort message, participants may remain in a blocked state, waiting indefinitely for a decision. This can cause resource locking and delays in the system.

- Lack of Fault Tolerance:
  The basic 2PC protocol does not inherently handle failures well, especially if the coordinator fails. Without additional mechanisms like logging, the system may be left in an inconsistent state, leading to potential data integrity issues.

Here, implementing a 2 phase distributed commit protocol for an online food delivery system named "**Momo's Dumpling delivery service**".

Momo's delivery service only accepts/places an order if the dumpling is available in storage and a delivery agent is available to deliver. Here, a process handling the orders is the **coordinator** and the process handling dumpling storage and delivery agents are **participants**.

Two Phase Commits:

Here the entire flow of ordering a dumpling is split into 2 phases

- Prepare - Reserve
- Commit - Book

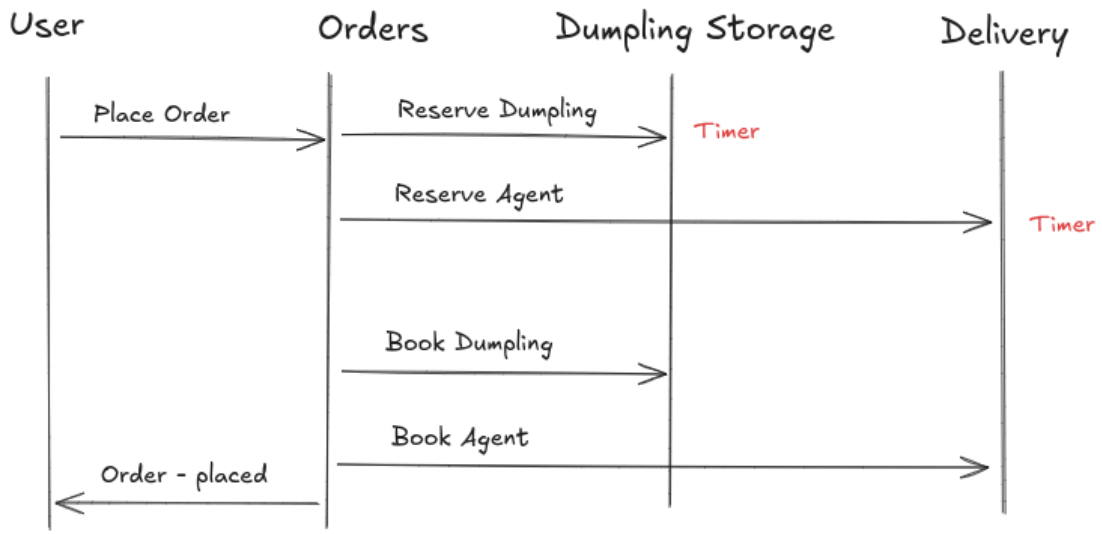


Fig: Two phases reserve and book

Where, the **Reserve Dumpling** reserves the first available dumpling packet
The **Reserve Agent** reserved the first available agent
The **Book Dumpling** assigns the first reserved food we find
The **Book Agent** assigns the first reserved agent we find

Phase 1: Reservation:

- If both fails, transaction fails, we **abort**
- If only one succeeds, we cancel the reservation and **abort**
- If both succeeds, we move ahead to the **commit phase**

Phase 2: Commit:

- If both succeeds, **order is placed**
- If only one fails, we cancel the reservation and **abort**

Initial Setup

1. Roles
   Orders: Acts as the coordinator.
   Dumpling Storage: Reserves the dumpling. [participant]
   Delivery: Reserves a delivery agent. [participant]

2. Sockets
   Each process will run on a different port.
   The coordinator (Orders) will communicate with the participants (Dumpling Storage and Delivery) over TCP sockets.

3. Two-Phase Commit Protocol
   Phase 1 (Prepare-Reserve): The coordinator sends a "reserve" request to Dumpling Storage and Delivery. Each participant replies with a "RESERVED" (success) or "NOT RESERVED" (failure).

   Phase 2 (Commit-Book): If both participants respond with "COMMITTED," the coordinator sends a "book" request. If either participant responds with "NOT COMMITTED," the coordinator sends an "abort" message.

Python Implementation

1. Coordinator Process:

```
# Orders: Acts as the coordinator

import socket

def orders_process():

    # Defining ports for Dumping Storage and Delivery
    port_storage = 5001
    port_delivery = 5002

    # Prepare phase - reserve
```

```
    print("Orders: Starting the prepare phase - Reserve")

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
storage_sock, \
    socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
delivery_sock:

        # Connecting to Dumping Storage
        storage_sock.connect(('localhost', port_storage))
        storage_sock.sendall(b'RESERVE DUMPLING')
        data = storage_sock.recv(1024)
        storage_response = data.decode()

        # Connecting to Delivery
        delivery_sock.connect(('localhost', port_delivery))
        delivery_sock.sendall(b'RESERVE AGENT')
        data = delivery_sock.recv(1024)
        delivery_response = data.decode()

        # If both the responses are positive, then commit
        if storage_response == "RESERVED" and delivery_response ==
"RESERVED":
            print("Orders: Reserve phase successful")
            print("Orders: Starting the commit phase")

            commit_status = commit(port_storage, port_delivery)
            if commit_status:
                print("Orders: Commit phase successful")
                print("Orders: Order placed successfully")
            else:
                print("Orders: Order failed during commit phase")
                abort(port_storage, port_delivery)
        else:
            print("Orders: Reservation failed. Aborting the
transaction.")
            abort(port_storage, port_delivery)


def commit(storage_port, delivery_port):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
storage_sock, \
    socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
delivery_sock:

        # Connecting to Dumping Storage to commit
        storage_sock.connect(('localhost', storage_port))
```

```
        storage_sock.sendall(b'BOOK DUMPLING')
        data = storage_sock.recv(1024)
        storage_response = data.decode()

        # Connecting to Delivery
        delivery_sock.connect(('localhost', delivery_port))
        delivery_sock.sendall(b'BOOK AGENT')
        data = delivery_sock.recv(1024)
        delivery_response = data.decode()

        # If both the responses are positive, then commit
        if storage_response == "COMMITTED" and delivery_response ==
"COMMITTED":
            return True
        else:
            return False

def abort(storage_port, delivery_port):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
storage_sock, \
    socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
delivery_sock:

        # Connecting to Dumping Storage to abort
        storage_sock.connect(('localhost', storage_port))
        storage_sock.sendall(b'ABORT DUMPLING')


        # Connecting to Delivery to abort
        delivery_sock.connect(('localhost', delivery_port))
        delivery_sock.sendall(b'ABORT DELIVERY AGENT')

        print("Orders: Transaction aborted")

if __name__ == "__main__":
    orders_process()
```

2. Participant Process (Dumpling Storage):

```
# Dumpling Storage: Reserves the dumpling.

import socket
import random
```

```
def dumpling_store_process():
    port = 5001
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.bind(('localhost', port))
        sock.listen()

        print("Dumplingstore: Waiting for the orders...")

        while True:
            conn, addr = sock.accept()
            with conn:
                data = conn.recv(1024).decode()

                if data == "RESERVE DUMPLING":
                    print("\n Dumpling Storage: Reserving
dumpling...")

                    temp = random.randint(0, 1)
                    if temp == 0:
                        conn.sendall(b"RESERVED")
                    else:
                        conn.sendall(b"NOT RESERVED")

                elif data == "BOOK DUMPLING":
                    print("Dumpling Storage: Booking dumpling...")

                    temp = random.randint(0, 1)
                    if temp == 0:
                        conn.sendall(b"COMMITTED")
                    else:
                        conn.sendall(b"NOT COMMITTED")


                elif data == "ABORT DUMPLING":
                    print("Dumpling Storage: Aborting dumpling
reservation...")

if __name__ == "__main__":
    dumpling_store_process()
```

3. Participant Process (Delivery):

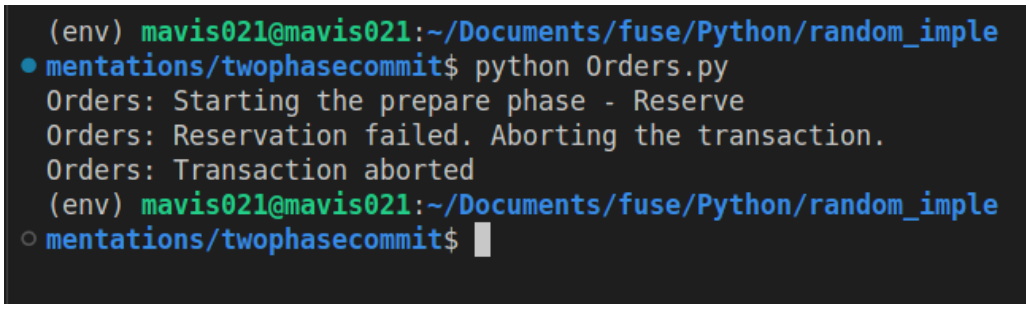
```
# Delivery: Reserves a delivery agent
```

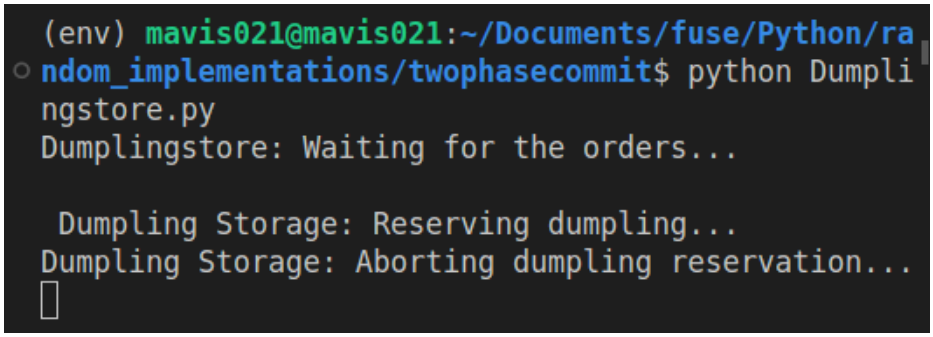
```
import socket
import random

def delivery_process():
    port = 5002
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.bind(('localhost', port))
        sock.listen()

        print("Delivery: Waiting for the orders...")

        while True:
            conn, addr = sock.accept()
            with conn:
                data = conn.recv(1024).decode()

                if data == "RESERVE AGENT":
                    print("\n Delivery: Reserving delivery agent...")
                    # Simulate reservation success

                    temp = random.randint(0, 1)
                    if temp == 0:
                        conn.sendall(b"RESERVED")
                    else:
                        conn.sendall(b"NOT RESERVED")

                elif data == "BOOK AGENT":
                    print("Delivery: Booking delivery agent...")

                    temp = random.randint(0, 1)
                    if temp == 0:
                        conn.sendall(b"COMMITTED")
                    else:
                        conn.sendall(b"NOT COMMITTED")

                elif data == "ABORT DELIVERY AGENT":
                    print("Delivery: Aborting delivery agent
reservation...")

if __name__ == "__main__":
    delivery_process()
```

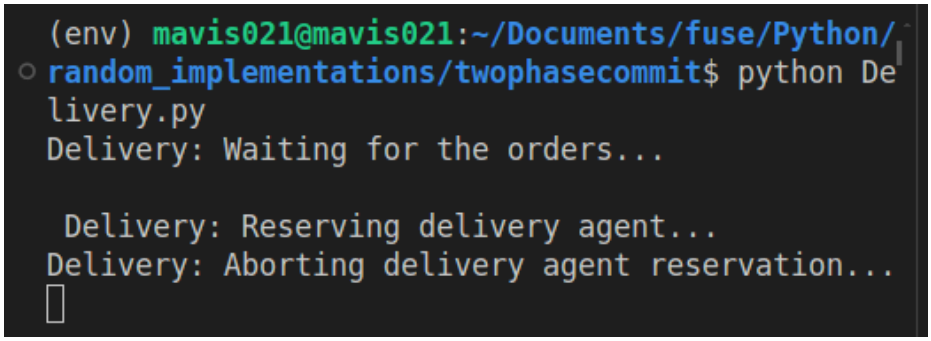
Here the **random** is used to provide the randomness in reservation or committing process by the participants.

Outputs

When a reservation phase fails







When a committing phase fails

```
(env) mavis021@mavis021:~/Documents/fuse/Python/random_imple
mentations/twophasecommit$ python Orders.py
Orders: Starting the prepare phase - Reserve
Orders: Reserve phase successful
Orders: Starting the commit phase
Orders: Order failed during commit phase
Orders: Transaction aborted
```

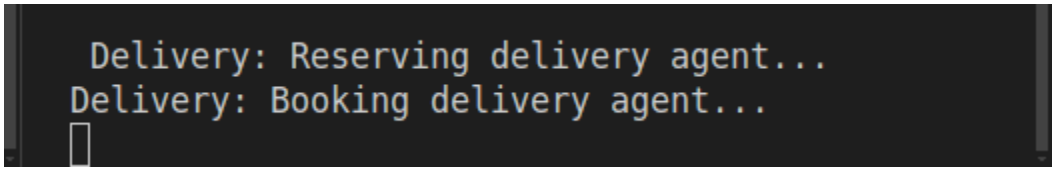
```
Dumpling Storage: Reserving dumpling...
Dumpling Storage: Booking dumpling...
Dumpling Storage: Aborting dumpling reservation...
```

```
Delivery: Reserving delivery agent...
Delivery: Booking delivery agent...
Delivery: Aborting delivery agent reservation...
```

Transaction success

```
(env) mavis021@mavis021:~/Documents/fuse/Python/random_imple
mentations/twophasecommit$ python Orders.py
Orders: Starting the prepare phase - Reserve
Orders: Reserve phase successful
Orders: Starting the commit phase
Orders: Commit phase successful
Orders: Order placed successfully
```

```
Dumpling Storage: Reserving dumpling...
Dumpling Storage: Booking dumpling...
```



Analysis and Discussion

The lab exercise required the implementation of a Two-Phase Commit (2PC) protocol in a distributed system. To fulfill this requirement, a scenario was developed involving a hypothetical distributed application -Momo's Dumpling delivery service. The 2PC protocol was applied to coordinate the reservation and booking processes between Dumpling Storage and Delivery services, ensuring that both either committed or aborted the transaction in unison, thus preserving consistency.

Challenges were identified during the implementation, including communication overhead due to the multiple messaging rounds between the coordinator and participants, which increased latency. Additionally, the basic 2PC protocol lacked fault tolerance, which posed a risk of leaving the system in an uncertain state if the coordinator failed. These challenges highlighted the potential need for improvements, such as timeout mechanisms, recovery logging, and concurrency control, to enhance the system's robustness.

Conclusion

In the context of this lab exercise, the Two-Phase Commit (2PC) protocol was successfully implemented to manage distributed transactions for Momo's Dumpling delivery service. While the protocol ensured consistency across the system, the exercise revealed challenges related to communication overhead and fault tolerance. These findings suggest that while 2PC is effective for maintaining consistency, additional mechanisms would be beneficial for handling failures and improving scalability in more complex systems.