CISC 5597 Distributed Systems

Lab 3: 2 Phase Commit Protocol

Name: Swoichha Adhikari

GitHub code repository: https://github.com/swoichha/CISC-5597-DistributedSystems/tree/master/Lab-3

Introduction

In this lab, I have implemented the 2 Phase Commit (2PC) Protocol using XML-RPC for distributed environment. 2PC protocol is a distributed algorithm that manages the coordination of all processes involved in a distributed atomic transaction, determining whether to commit the transaction or roll it back (abort) based on the votes/participant.

Objectives

- Implement the 2 Phase Commit (2PC) Protocol using a distributed approach

- Create a client-server architecture where client request to perform a transaction which is handle by server(coordinator) with participant nodes using 2PC and XML-RPC

- Ensure that coordinator counts votes from participant before making on decision to commit or abort. Making sure that participant nodes can accept the transaction commit, reject the commit, and maintain the consistency

- Handle potential failures and restarts of nodes, ensuring that transaction consistency is still reached

System Architecture:

The system consists of two participant nodes, a coordinator node, and a client node running on different IPs. The client initiates the transaction by using the above-mentioned command, which the coordinator receives. The coordinator then performs this transaction based on the scenario number by using 2PC through participants A and B. Each participant is responsible for handling transaction requests and responses and committing values.

The communication between the client, coordinator and the participants are done using XML-RPC. The protocol's flow involves broadcasting ready to commit to the participant, handling responses, and committing agreed-upon values for each transaction on account.txt file based on the node in the system.

Components:

1. Client:

- The client interacts with the coordinator and sends commands to execute different transactions, which trigger different phases of the 2PC protocol

- It supports commands like:

    - scenario <scenario_number>: <first_transaction_number> <second_transaction_number> => Perform different scenarios as per the lab questions and the two transactions mentioned in the lab
    - restart: Resets all nodes to their initial states.

2. Coordinator:

- Handles transaction initiation, communication, and commit or abort transactions.
- Coordinates actions between participating nodes.

3. Node A and Node B

- Check if they can commit to the transaction based on the scenario number and the transaction number
- Simulate crash using time sleep
- Validate if the account.txt file exist or not and if the balance is enough to perform the transaction or not

Logic and Workflow:

The client enters a command to execute transactions on Node A and B through. The following steps is for the 2PC Protocol implemented between coordinator and participant nodes:

1. Commit-request phase (or voting phase):
    - The coordinator asks all participants whether they can carry out a transaction. If the participant nodes can perform the transaction, then they will vote ‘Yes’ by returning *True*, or else they will vote ‘No’ by returning *False*.
    - On the participant side, the node checks the balance in the account by reading the account_A.txt and account_B.txt files, which exist on participant A and B ends, respectively. Based on the scenario number and transaction number, these participants will agree to commit if the balances are greater than or equal to the minimum value required to perform the transaction.

i. Node A

```python
def initialize_account(self, scenario_number):
    """
    Set the initial value of the account based on the scenario number.
    """
    self.scenario_number = scenario_number      You, 2 days ago • Added log file for each node
    if scenario_number == 2:
        self.balance = 90.0
    else:
        self.balance = 200.0
    try:
        write_account(self.account_file, self.balance)
        logging.info(colored(f"Account initialized with {self.balance} for Scenario {scenario_number}.", 'blue'))
        return True
    except Exception as e:
        logging.error(colored(f"Error initializing account: {e}", 'red'))
        return False
```

ii. Node B

```python
def initialize_account(self, scenario_number):
    """
    Set the initial value of the account based on the scenario number.
    """
    print("****** scenario number",scenario_number)
    self.scenario_number = scenario_number
    if scenario_number == 2:
        self.balance = 50.0
    else:
        self.balance = 300.0
        if scenario_number == 3:
            self.crash_before = True
        elif scenario_number == 4:
            self.crash_after = True
    try:
        write_account(self.account_file, self.balance)
        logging.info(colored(f"Account initialized with {self.balance} for Scenario {scenario_number}.", 'blue'))
        return True
    except Exception as e:
        logging.error(colored(f"Error initializing account: {e}", 'red'))
        return False
```

2. Commit Phase:
    - After receiving a response from the commit-request phase, the coordinator checks if all participants agreed to commit or not. If all agree, then it proceeds to perform the transaction and asks the participant nodes to perform the transaction.
    - If any one or more node votes ‘No’ then the coordinator sends abort request to the participants and the whole process is aborted. Log files will be updated for it.
3. **Handling Failures:**
    - The participants are designed to restart and clear their state (account.txt file reset) on command “restart”, simulating a clean scenario to run and test different cases without closing the terminal and rerunning the code.
    - The participant checks logs after recovering from a crash using *get_last_command* and based on the previous log it will continue the transaction or will abort the transaction.
    - If *abort( )* is called upon at any point, then a flag *revert is checked.* If it is true then the account balance is reverted back to what it was after the last transaction.
    - If an unexpected error occurs, then an error message is thrown, and the log can be seen in the terminal

Key Functions and Code Walkthrough:

1. **Coordinator side:**

a. execute_transaction( ): Executes the entire Two-Phase Commit (2PC) protocol for a given scenario and transaction.

```python
def execute_transaction(self, scenario_number,transaction_number):
    """Execute the distributed transaction."""
    self.transaction_number = transaction_number
    self.scenario_number = scenario_number
    try:
        # Initialize accounts only if this scenario hasn't been initialized
        if scenario_number not in self.initialized_scenarios:
            initialize_val_A = self.participant_1.initialize_account(self.scenario_number)
            initialize_val_B = self.participant_2.initialize_account(self.scenario_number)

            if initialize_val_A and initialize_val_B:
                self.initialized_scenarios.add(scenario_number)
                self.log_action("Accounts initialized for scenario", self.scenario_number, self.transaction_number)
                logging.info(colored(f"Accounts initialized for scenario {scenario_number}.", 'green'))
            else:
                self.log_action("Failed to initialize account for scenario", self.scenario_number, self.transaction_number)
                logging.error(colored(f"Failed to initialize accounts for scenario {scenario_number}.", 'red'))
                return f"Error during initializing values for scenario {scenario_number}.", False
        else:
            logging.info(colored(f"Scenario {scenario_number} already initialized. Skipping initialization.", 'yellow'))
      You, last week • refactor initialization for A n B along with re…

    except Exception as e:
        logging.error(colored(f"Error during initializing value for scenario {self.scenario_number} before starting 2PC: {str(e)}", 'red'))
        return f"Error during initializing value for scenario {self.scenario_number} before starting 2PC: {str(e)}",False

    try:
        # Prepare Phase
        logging.info(colored("Prepare Phase Initiated", 'green'))
        canCommit = self.preparePhase(self.transaction_number)  # Timeout set to 10 seconds

        # Commit Phase
        if canCommit:
            self.log_action("Prepare Phase Successfully Completed", self.scenario_number, self.transaction_number)
            logging.info(colored("Prepare Phase Successfully Completed ", 'green'))
            logging.info(colored("Commit Phase Initiated", 'green'))
            doCommitStatus = self.commitPhase( self.transaction_number)

            if doCommitStatus:
                logging.info(colored("Transaction Committed Successfully to node A and B", 'green'))
                return f"Transaction Committed Successfully", True
            else:
                return "Transaction Failed", False
        else:
            logging.error(colored("Transaction Aborted Initiated on node A and B", 'red'))
            self.abort_transaction()
            return "Transaction Aborted", False

    except Exception as e:
        self.abort_transaction()
        logging.error(colored(f"Transaction Failed: {str(e)}", 'red'))
        return "Transaction Aborted", False
```

b. log_action( ): Logs the actions performed by the coordinator into a file named "coordinator_log.txt" for tracking purposes.

c. PreparePhase( ): Executes the prepare phase by coordinating with participants to determine their readiness for the transaction. Establishes consensus among participants before proceeding to the commit phase.

```python
def preparePhase(self,transaction_number):      You, last week • 2PC: scenario1
    # Prepare Phase
    """Check if a participant can commit."""

    try:
        logging.info(colored(f"Checking if participant A can commit", 'yellow'))
        canAcommit = self.canNodesCommit(self.participant_1,transaction_number)
        # Step 1: Check Node-A
        if not canAcommit:
            self.log_action("Node A preparePhase NO", self.scenario_number, self.transaction_number)
            logging.warning(colored("Node-A cannot commit. Aborting transaction.", 'red'))
            return False

        # Step 2: Check Node-B with timeout
        logging.info(colored(f"Checking if participant B can commit", 'yellow'))
        response_B = [None]  # Use a list to capture response from thread

        def check_node_b():
            response_B[0] = self.canNodesCommit(self.participant_2, transaction_number)

        # Start the thread to wait for Node-B's response
        thread = threading.Thread(target=check_node_b)
        thread.start()
        thread.join(timeout=5)  # Wait for up to 5 seconds

        if response_B[0] is None:  # Timeout occurred
            self.log_action("Node A preparePhase NO", self.scenario_number, self.transaction_number)
            logging.error(colored("Timeout waiting for Node-B's response. Aborting transaction.", 'red'))
            return False  # Or handle it as you wish (e.g., partial commit, abort, etc.)
        elif not response_B[0]:  # Node-B explicitly cannot commit
            self.log_action("Node A preparePhase NO", self.scenario_number, self.transaction_number)
            logging.warning(colored("Node-B cannot commit. Aborting transaction.", 'red'))
            return False

        # If both nodes can commit
        return True

    except Exception as e:
        logging.error(colored(f"Error during prepare phase: {str(e)}", 'red'))
        return False
```

d. commitPhase( ): Handles the commit phase where all participants finalize the transaction. Ensures the distributed transaction is completed, or aborts if any participant fails.

```python
def commitPhase(self,transaction_number):

    # Commit Phase
    """Check if a participant can commit."""
    try:
        logging.info(colored(f"Attempting to commit on Node A", 'blue'))
        A_commit_status, increment  = self.participant_1.doCommit(transaction_number)
        logging.info(colored(f"Attempting to commit on Node B", 'blue'))      You, last week • 2PC: scenari

        # Simulate crash for Node B (after voting to commit, but before actually committing)
        response_B = [None]  # Use a list to capture response from thread

        def commit_node_b():
            # Participant B tries to commit
            response_B[0] = self.participant_2.doCommit(transaction_number, increment)

        # Start the thread to wait for Node-B's response
        thread_B = threading.Thread(target=commit_node_b)
        thread_B.start()
        thread_B.join(timeout=5)  # Wait for up to 5 seconds for Node B to commit


        if response_B[0] is None:  # Timeout occurred (simulating crash of Node B)
            logging.error(colored("Timeout waiting for Node B's commit. Assuming crash.", 'red'))
            # Simulate that Node B crashed, and abort the transaction
            self.abort_transaction(True)
            return False  # Return false indicating the transaction failed

        # If Node B commits successfully
        if A_commit_status and response_B[0]:
            self.log_action("Committed YES", self.scenario_number, self.transaction_number)
            logging.info(colored("Transaction Committed Successfully to Node A and B", 'green'))
            return True
        else:
            self.log_action("Committed NO", self.scenario_number, self.transaction_number)
            logging.error(colored("Error during commit phase.", 'red'))
            return False

    except Exception as e:
        logging.error(colored(f"Error during commit transaction: {str(e)}", 'red'))
        logging.error(colored("Simulated crash for Node B. Aborting transaction.", 'red'))
        self.abort_transaction()  # Abort transaction on both nodes
        return False
```

e. canNodesCommit(): Checks if a participant is ready to commit the transaction.
f. abort_transaction(): Aborts the transaction by instructing all participants to roll back any changes made during the transaction.
    i. Abort

```python
def abort_transaction(self, revert_A =False, revert_B =False):      You, 2 days ago • fixed issues with
    """Abort transaction for all participants."""
    self.log_action("ABORT", self.scenario_number, self.transaction_number)
    logging.warning(colored("Aborting transaction for all participants", 'red'))
    try:
        self.participant_1.abort(revert_A)
    except Exception as e:
        logging.error(colored(f"Error aborting transaction on node A: {str(e)}", 'red'))

    try:
        self.participant_2.abort(revert_B)
    except Exception as e:
        logging.error(colored(f"Error aborting transaction on node B: {str(e)}", 'red'))
```

    ii. Rollback

```python
def get_last_commit_value(self, starts_with):
    try:
        last_command = self.get_last_command()  # Get the last command from the log
        if last_command and last_command.startswith(starts_with):
            # Use regular expression to find the last float number
            match = re.search(r"([+-]?\d+\.\d+)$", last_command)
            if match:
                # Extract and return the float value
                return float(match.group(1))
            else:
                return None  # No float found at the end of the line
        else:
            return None  # Line does not start with the specified prefix

    except Exception as e:      You, 2 days ago • Uncommitted changes
        print(f"Error reading the log file: {e}")
        return None
```

g. restart(): Resets the coordinator and participants to their initial state. Prepares the system for a new transaction cycle or recovers from a failure.

2. Client-side Functions:
    - Upon entering the command in terminal, it calls execute_transaction of the Coordinator through XML-RPC

- logs response received from coordinator

```python
if command.lower() == "exit":
    logging.info(colored("Exiting client.", 'green'))
    break
elif command.startswith("restart"):
    resultMsg, restart_status = coordinator.restart()
    if restart_status:
        logging.info(colored(f"Restart status: {resultMsg} ", 'green'))
    else:
        logging.info(colored(f"Restart status: {resultMsg} ", 'red'))
elif command.startswith("scenario 1") or command.startswith("scenario 2") or command.startswith("scenario 3") or command.startswith(
    try:
        # Validate and parse the command
        if ':' not in command:
            raise ValueError("Invalid format. Use 'scenario <scenario_number>: <first_transaction_number> <second_transaction_numbe

        scenario_part, transactions_part = command.split(":", 1)
        scenario_number = int(scenario_part.split()[1])
        transaction_numbers = transactions_part.strip().split()

        if len(transaction_numbers) != 2:
            raise ValueError("Please provide exactly two transaction numbers.")

        first_transaction_number = int(transaction_numbers[0])
        second_transaction_number = int(transaction_numbers[1])

        logging.info(colored(f"Executing Scenario {scenario_number} with transactions {first_transaction_number} and {second_transac

        # Execute the first transaction
        resultMsg1, first_transaction_status = coordinator.execute_transaction(scenario_number, first_transaction_number)      You,
        if first_transaction_status:
            logging.info(colored(f"Transaction {first_transaction_number} Status: {resultMsg1}", 'green'))
        else:
            logging.info(colored(f"Transaction {first_transaction_number} Status: {resultMsg1}", 'red'))
        # Execute the second transaction
        resultMsg2, second_transaction_status = coordinator.execute_transaction(scenario_number, second_transaction_number)
        if second_transaction_status:
            logging.info(colored(f"Transaction {second_transaction_number} Status: {resultMsg2}", 'green'))
        else:
            logging.info(colored(f"Transaction {second_transaction_number} Status: {resultMsg2}", 'red'))

    except Exception as e:
        logging.error(colored(f"Error in command: {e}", 'red'))

else:
    logging.error(colored("Invalid command.", 'red'))
```

3. Participant- side features and functions:

- Account Management:
  - Implemented account initialization, reading, writing, and balance updates based on the transaction and scenario number.
  - Checks if account exists or not
  - Deduction and increment of account balance based on the transaction and scenario number.
- Transaction Preparation and Commit:
  - canCommit(): Checks if the participant can commit to the transaction being executed based on the current balance.

- Node A

```python
def canCommit(self, transaction_number):
    self.transcation_number = transaction_number
    try:
        logging.info(colored(f"CAN COMMIT:", 'yellow'))
        if not os.path.exists(self.account_file):
            self.prepared = False
            self.log_action("Vote NO", self.scenario_number, transaction_number)
            logging.info(colored(f"Account A does not exists", 'red'))
            return self.prepared
        else:
            self.balance = float(read_account(self.account_file))
            if self.transcation_number == 1:
                # Simulate the operation: subtract $100 and add 20% bonus
                if self.balance >= 100:      You, last week • 2PC: scenario1
                    logging.info(colored(f"Account A exists and balance {self.balance} > 100", 'green'))
                    self.prepared = True
                    self.log_action("Vote YES", self.scenario_number, transaction_number)
                else:
                    logging.info(colored(f"Account A exists and balance {self.balance} < 100", 'red'))
                    self.prepared = False
                    self.log_action("Vote NO", self.scenario_number, transaction_number)
                return self.prepared
            elif self.transcation_number == 2:
                if self.balance > 0:
                    logging.info(colored(f"Account A exists and balance {self.balance} > 0", 'green'))
                    self.prepared = True
                    self.log_action("Vote YES", self.scenario_number, transaction_number)
                else:
                    self.log_action("Vote NO", self.scenario_number, transaction_number)
                    logging.info(colored(f"Account A exists and balance {self.balance} <=0", 'red'))
                return self.prepared
            else:
                self.log_action("Vote NO", self.scenario_number, transaction_number)
                logging.info(colored(f"Account A exists but the transaction id is not valid", 'red'))
                return False

    except Exception as e:
        self.log_action("Vote NO", self.scenario_number, transaction_number)
        logging.error(colored(f"Error during canCommit for: {str(e)}", 'red'))
```

- Node B

```python
def canCommit(self, transaction_number):
    self.transcation_number = transaction_number

    if not os.path.exists(self.account_file):
        self.prepared = False
        self.log_action("Vote NO", self.scenario_number, transaction_number)
        logging.info(colored(f"Account B does not exists", 'red'))
    else:
        self.prepared = True
        self.balance = float(read_account(self.account_file))
        self.log_action("Vote YES", self.scenario_number, transaction_number)
        logging.info(colored(f"Account B exists and balance is {self.balance}", 'green'))      You, 2 days ago • Uncommitted changes


    # Simulate a delay/crash for Node-B.
    if self.crash_before:  # Simulate crash for Node-2
        logging.warning(colored(f"Simulating crash for Transaction:{self.transcation_number} simulating a crash (long sleep)...", 'yellow'))
        time.sleep(10)  # Simulate long sleep to represent crash

    try:
        last_command = self.get_last_command()  # Get the last command from the log
        if self.crash_before:
            if last_command.startswith('Vote'):
                self.abort()
            self.crash_before = False
        return self.prepared
    except Exception as e:
        self.log_action("Vote NO", self.scenario_number, transaction_number)
        logging.error(colored(f"Error during canCommit for: {str(e)}", 'red'))
```

- doCommit(): Executes a transaction and updates the balance, logging the changes.

- Node A

```python
def doCommit(self, transaction_number):
    self.transcation_number = transaction_number
    increment = None
    try:
        if self.transcation_number == 1:
            new_balance = self.balance - 100
            write_account(self.account_file, new_balance)
            logging.info(colored(f"Account A updated value from {self.balance} to {new_balance} after Transaction {self.transcation_number}", 'green'))
            self.log_action("COMMITED YES", self.scenario_number, transaction_number, -100.00)
            self.balance = new_balance
            self.commit_status = True


        elif self.transcation_number == 2:
            increment = 0.2 * self.balance
            new_balance = self.balance + increment
            write_account(self.account_file, new_balance)
            self.log_action("COMMITED YES", self.scenario_number, transaction_number, increment)
            self.balance = new_balance
            self.commit_status = True
            logging.info(colored(f"Account A increment value by {increment} after Transaction {self.transcation_number}", 'green'))
            logging.info(colored(f"Account A new balance: {self.balance}", 'green'))

        else:
            self.log_action("COMMITED NO", self.scenario_number, transaction_number)
            logging.info(colored(f"Invlaid transaction {transaction_number} ", 'red'))
        return self.commit_status, increment

    except Exception as e:
        self.log_action("COMMITED NO", self.scenario_number, transaction_number)
        logging.error(colored(f"Error during commit: {str(e)}", 'red'))
```

- Node B

```python
def doCommit(self, transaction_number, increment = None):
    self.transcation_number = transaction_number
    # Simulate a delay/crash for Node-B.
    if self.crash_after:  # Simulate crash for Node-2
        logging.warning(colored(f"Simulating crash for Transaction:{self.transcation_number} simulating a crash (long sleep)...", 'yellow'))
        time.sleep(10)  # Simulate long sleep to represent crash
        # self.crash_after = False

    try:
        last_command = self.get_last_command()
        if last_command.startswith('Vote'):
            if self.crash_after:
                self.abort()
                self.crash_after = False
            else:
                if self.transcation_number == 1:
                    new_balance = self.balance + 100
                    write_account(self.account_file, new_balance)
                    logging.info(colored(f"Account B updated value from {self.balance} to {new_balance} after Transaction {self.transcation_number}", 'green'))
                    self.log_action("COMMITED YES", self.scenario_number, transaction_number, +100.00)
                    self.balance = new_balance
                    self.commit_status = True

                elif self.transcation_number == 2:
                    new_balance = self.balance + increment
                    write_account(self.account_file, new_balance)
                    self.log_action("COMMITED YES", self.scenario_number, transaction_number, increment)
                    self.balance = new_balance
                    self.commit_status = True
                    logging.info(colored(f"Account B increment value by {increment} after Transaction {self.transcation_number}", 'green'))
                    logging.info(colored(f"Account B new balance: {self.balance}", 'green'))      You, 2 days ago • Uncommitted changes
                else:
                    self.log_action("COMMITED NO", self.scenario_number, transaction_number)
                    logging.info(colored(f"Invlaid transaction {transaction_number} ", 'red'))
                return self.commit_status

    except Exception as e:
        self.log_action("COMMITED NO", self.scenario_number, transaction_number)
        logging.error(colored(f"Error during commit: {str(e)}", 'red'))
```

- Logging System:
    - Maintains a log (account_A_log.txt) for all actions, including votes, commits, and aborts.
    - Retrieve the last command or commit value from the log
- Error Handling:
    - handling of exceptions during transactions, commits, and logging operations.
- Abort and Rollback:
    - abort: aborting transactions, optionally reverting to the last commit.
    - If aborted, then based on previous transaction logs will revert the value.

How to execute:

First change the IP and port number in each file.

1. Start the coordinator:

- Run script: python3 coordinator.py.
- The coordinator will start listening for client command

2. Start the participant node:

- Run script: python3 participant_node_A.py.
- Run script: python3 participant_node_B.py.

3. Start the Client:

- Run the client-side script: python3 client.py.
- The client will connect to the coordinator and can run the following available commands to execute the transactions.
- Available Commands (All the commands are case sensitive):
  - scenario <scenario_number>: <first_transaction_number> <second_transaction_number>
    - Example: scenario 1: 1 2
    - This will execute the 1.a scenario where the balance of A is 200 and B is 300. This will perform 1st transaction i.e. transfer 100 from A to B and then perform the 2nd transaction i.e. 20% bonus to A and add the same amount (0.2*A) to B
  - restart: will set all values and account balance to 0 and allows to perform transaction in fresh setup.
  - exit: Disconnects from the coordinator.

Challenges

- Imitating crash on Node B: For 1.c.i and 1.c.ii we needed to show that Node 2 crashes for which I added some sleep to imitate the crash. During the sleep coordinator did not receive response from Node 2 and assumes that Node 2 has crashed. But the challenge was to have the right value of sleep for Node 2 and right waiting time value for coordinator. After multiple trial and error, the perfect value was found.

- Recovery from the crash: Once the sleep is over and Node 2 is back online, then it should abort the transaction as coordinator triggered abort when it could not get a response from Node 2 when it was sleep/crashing. I tried different logics for this the best way was to have a log file for each node and based on the last log the Node will perform the transaction.