Project Report

Shilpa Dhagat

- **Atomicity and Concurrency –**

    - I am using only get() and put() methods of ConcurrentHashMap, which are thread safe.
    - While implementing LocalBank I haven't used synchronized lock anywhere except in currentBalances() method, which is non-atomic operation and iterates through multiple accounts. I have used synchronized block to make sure that I take a correct snapshot of all the bank accounts.
    - Other methods in LocalBank don't use explicit lock as it uses ConcurrentHashMap to get the instance of RemoteAccount against an account number. Rest of the thread safety considerations are delegated to RemoteAccountImpl.

- **Atomicity and thread safety in RemoteAccount (RemoteAccountImpl) -**

  I'll discuss the case of deposit vs withdrawal implementation in RemoteAccountImpl. Both these methods are synchronized. Both these methods first validate and then do the update.

    - Deposit – It checks that deposit amount is positive. Following the validation the program deposits the amount.
    - Withdrawal - It checks that withdrawal amount is positive and withdrawal amount is less than the current available amount. Following the validation the program withdraws the amount.

- **Large number of random operations on a bank account using all three mechanisms**

  I have written a client LoadTestClient to perform this scenario. I have performed these testcases for 100 existing accounts. In each run I am randomly performing 1000 operations for each Buffered, Operation and Remote types of operations.

  The timing for this particular run are as follows:
  Buffered - 3 ms
  Operation - 98 ms
  Remote - 80 ms

  Logically speaking Buffered should take least amount of time as it performs multiple local operations followed by a remote sync call.;
  Remote should take more time as it involves one RMI call with each operation.
  Operation type should take maximum amount of time as it involves double RMI call with each operation.

- **Special Scenario 1 – RemoteAccount**

  - RemoteAccount account = bank.getRemoteAccount(id); - Instance of bank is loaded via the lookup service. BankServer, which is an RMI server serializes Bank instance and TestClient de serializes the instance and loads it.

  - account.deposit(5); - Test client serializes the invocation and sends it to BankServer. Bank server de serializes the invocation, invokes this method and serializes the result to the invoker. Invoker de serializes the response and loads it in JVM.
  - account.withdraw(10); - Test client serializes the invocation, sends it to BankServer. Bank server de serializes the invocation, invokes this method. An exception is thrown because the withdrawal amount exceeds the balance. BankServer serializes the exception to the invoker. Invoker de serializes the Exception and loads it in JVM before it is thrown.

- **Special Scenario 2 – BufferedAccount**

  - BufferedAccount account = bank.getBufferedAccount(id); - Instance of BufferedAccount is created by LocalBank, which is a Remote instance. This statement sends the request via RMI, LocalBank receives this request, creates a new BufferedAccount instance, passing the RemoteAccount instances against the accountId and sends it back via RMI channel.

  - account.deposit(5); - This call doesn't invoke any RMI invocation. It is a local change on the client side. It updates BufferedAccount.

  - account.sync(); - account is an instance of BufferedAccount, which is a regular java object. However, it contains an instance of RemoteAccount, on which it either invokes a deposit or withdraw before calling getBalance. All the calls deposit, withdraw and getBalance are RMI calls to the actual remote server.

  - account.withdraw(10); - This call doesn't invoke any RMI invocation. It is a local change on the client side. It updates BufferedAccount. However during withdraw operation the program finds that the account doesn't have enough balance and it throws the Exception.

- **Special Scenario 3 – Operation case**

  Least performant way of performing an operation. It involves double RMI invocation, which is not needed.

  - bank.requestOperation(id, Operation.deposit(5)); - It first invokes LocalBank implementation via RMI call, which in turn makes another RMI call to the

RemoteAccountImpl to deposit the amount. Since it involves double RMI call, this is least performant.

- bank.requestOperation(id, Operation.withdraw(10)); It first invokes LocalBank implementation via RMI call, which in turn makes another RMI call to the RemoteAccountImpl to withdraw 10 . Since the account doesn’t have enough balance the RemoteAccount throws BankException, which is received by LocalBank followed by the invoker.