2-Phase Distributed Commit(2PC)

-[USING JAVA RPC]

I have never given nor received unauthorized assistance on this work.

**Sign: Date:** 01/11/2022

Durga Pramodh Kumar Jajala (1001960015)

And

Kumar, Subham (1002021023)

# Introduction

In this project , we have implemented an 2-node distributed system that implements a 2-phase distributed commit(2PC). The distributed system uses a logical clock to timestamp messages sent/received among the nodes.

**2-Phase Distributed Commit(2PC):**

The two phase commit (2PC) protocol is an atomic commitment protocol which is a specialized type of consensus protocol. The goal of the protocol is to atomically commit / store a new value at all of the nodes in a distributed system that store that value so that all nodes are synced to the new value, even if there is node failure or network failure during the commitment process. **How does 2PC work :**

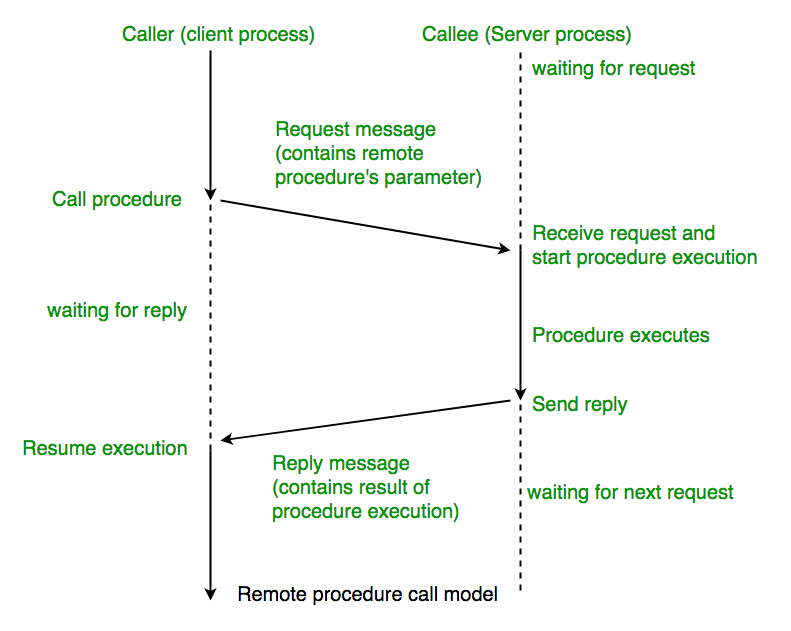
* The global coordinator prompts the commit point site to commit and the action is performed.
* The commit point site records its commitment and sends a response back to the global coordinator, informing that it has successfully committed.
* The global and local coordinators instruct all other nodes to commit to the transaction.
* Each node's database releases its locks and commits its local portion of the distributed transaction.
* Each node's database registers an additional redo entry in its local log to show that is has committed the transaction.
* All participating nodes alert the global coordinator to the status of their successful commitment.

**Implementation:**

In this project we have chosen to use java’s S RMI API to implement the vector clock and subsequent operations. To simulate a client-server architecture, we have created two java projects one acts a client and other acts as a server that listen on a port in the localhost .

In server side, we have implemented an interface that supports. All the services on the server side are registered with java RMI registry and exported on a specific known port that listens to the incoming requests.

In the client side, we have implemented 2 participants that make requests to the server. We have also handled concurrency issues on the server side for transactions using java’s multi-threading capabilities like synchronized methods to avoid simultaneous requestion accessing the same resource causing the data inconsistency.



RPC Communication through Java RMI API.

**2-Phase Distributed Commit(2PC):**

We have implemented a multi-threaded file server that implements 2PC. Java’s executor service is used to create a thread pool so that each request can be run on a separate thread that communicates with the server and gets the result and then terminates. This approach is extremely useful so that independent operations can run without waiting on the thread to complete the entire life cycle.

# Computation Server Implementation Details

We have created three process registered this interface with java RMI registry and listens to the client requests on the specified port. For this server we have used both synchronous client which waits the response from the server before proceeds with executing rest of the operation and asynchronous client which do not for the response from the server and notifies the main thread when we get the response from the server.

**Observations:**

1. Java provide package to implement RPC communication
2. Every request must be handled with proper timeouts and error scenarios.
3. Client-server communication is complex to implement with RPC protocol
4. Synchronization must be handled properly for implementing transactions especially when multiple threads are trying to access the same resource.

**Issues Faced:**

1. Transaction synchronization was a little hard to achieve as the multiple events were triggered for any operation and should be handled carefully
2. Java RMI registry can act as a single point of failure as we are register all the services to the registry and client finds the server using this stub in registry.
3. While multiple threads performs the operation, concurrency must be handled properly.

**Contribution:**

**Durga Pramodh Kumar, Jajala (1001960015):**

1. Implemented the computation server and both synchronous and asynchronous client.
2. Implemented multi-threaded file server that creates a new thread for every request
3. Refactored the code and handled exceptions for negative scenarios
4. Performed and an extensive testing on all operations.

**Subham, Kumar(1002021023)**

1. Implemented vector clock operations.
2. Worked on Participant and coordinator code synchronization
3. Performed testing on the RPC communication.
4. Researched on the java RMI implementation.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*The End\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***