Two-Phase Commit Design

1. Message Protocol Design

1.1 Prepare Message

- String txnId: Transaction ID
- byte[] imgBytes: data of the target college image
- public List<String> requestedImages;

1.2 VoteMessage

- String txnId: Transaction ID
- boolean vote: true for YES (can commit), false for NO (cannot commit)

1.3 CommitMessage

- String txnId: Transaction ID (matching the prepare message)

1.4 AbortMessage

- String txnId: Transaction ID

1.5 AckMessage

- String txnId: Transaction ID

Since the server knows the status of each transaction, the Ack message (Commit or Abort) is not needed in the message protocol in order to reduce message size.

2. Transaction Data Structure Design

2.1 Server

Server uses a concurrent hashmap to keep track of all active transactions:

- private static ConcurrentHashMap<String, CoordinatorTransaction> transactions

The CoordinatorTransaction object stores this information:

- private State state: Current state of the transaction (INIT, PREPARING, COMMITTING, ABORTING, COMMITTED, ABORTED)
- private String id: The unique ID of the transaction
- private String filename: The filename of the collage photo to be created
- private byte[] imageData: The image data of the collage photo
- private Map<String, List<String>> participantImages: Maps each participant (UserNode) to the list of images requested from it
- private Set<String> voteReceived: Set of participants that have sent their votes
- private Set<String> ackRemained: Set of participants that still need to acknowledge the final decision

This structure allows the server to maintain complete state for each transaction, track voting progress, and ensure all participants properly acknowledge the final commit or abort decision.

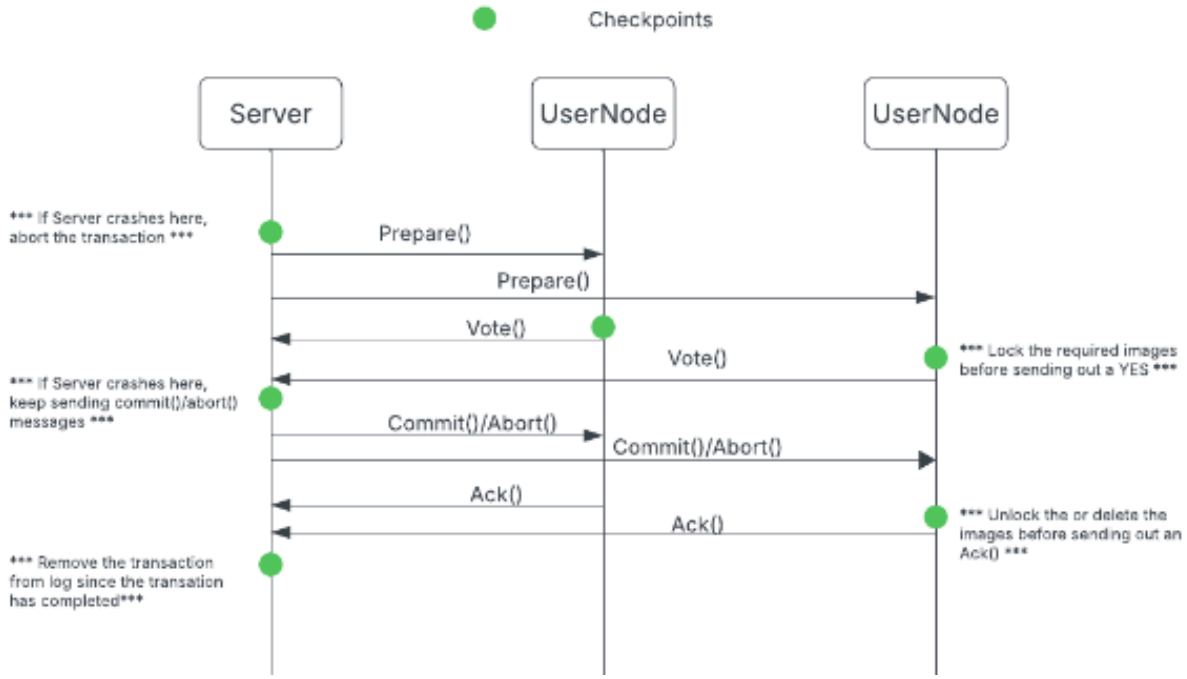
2.2 UserNode

UserNode doesn't need to keep every detail of all transactions, as UserNode only responds to the Server's messages passively. The only information it needs to track is which images are locked by which transactions. This minimalist approach reduces memory usage and simplifies the UserNode implementation while still ensuring correctness of the two-phase commit protocol.

- private Map<String, List<String>> activeTransactions; A list of active transactions and the images they require.
- private Map<String, String> lockedImages: a list of locked images. No one can try to acquire these images

3. Recovery Mechanism

Checkpoints are set up as shown in the graph. The system's state is written into file systems at each checkpoint. While restarting, both the Server and the UserNode would read from the disk and retrieve their previous state. This ensures that the system can recover correctly after crashes and continue the two-phase commit protocol from where it left off.



4. Synchronization

The system implements multi-threading and uses synchronized blocks and ConcurrentHashMap to ensure thread safety and proper synchronization.

4.1 Threads in Server

- Main thread: Initializes the Server, performs state recovery from disk, then handles incoming messages from all UserNodes and processes them according to the message type.
- **Transaction processing threads (N)**: Each incoming commit request spawns a new thread to handle the startCommit function. These threads complete after sending out the Prepare messages to all participants.
- **Transaction timer threads (N)**: For each transaction, a timer thread is created when sending Prepare messages. This thread waits 3 seconds and then checks if all votes have been received. If not, it aborts the transaction.
- **Action completion thread**: This single thread runs continuously, checking every 3 seconds for any transactions in COMMITTING or ABORTING states that are missing acknowledgments from nodes. It resends commit or abort messages to those nodes that haven't acknowledged yet.

4.2 Threads in UserNode

- **Main thread**: Initializes the UserNode, performs state recovery from disk, then enters a sleep loop to keep the process alive.
- **deliverMessage thread**: Handles all incoming messages from the Server, processing PREPARE, COMMIT, and ABORT messages as they arrive.

Proper synchronization ensures that concurrent access to shared transaction data is safe and operations are atomic when needed.