Problem Statement

The problem statement is to implement a State-based Conflict-free replicated data type(CvRDT) system using the set data structure.

Two endpoints to be made available by each node to receive insert and remove items requests from the main_set that is stored locally.

- When an item is inserted to the set using the /add endpoint, the item must reflect in other nodes' set too.
- When an item is removed from the set using the /rem endpoint, the item must be removed from the other nodes' set too.

Proof of Correctness

In the case of implementing a CvRDT, there are few important properties needs to be hold, they are-

Convergence

This property ensures that eventually all replicas or nodes will end up in the same state. In this case, the main_set will be the same across all the replicas.

Proof

This property ensures that eventually all replicas or nodes will end up in the same state. In this case, the main_set will be the same across all replicas.

- Each replica maintains three ordered set data structure variables: main_set, add_set, and rem_set.
    1. main_set stores the final state locally. Insertions or deletions requests from users at the /add and /rem endpoints are applied directly to the main_set.
    2. add_set stores the items being inserted into the main_set. Whenever an item is inserted into the main_set, it’s also inserted into the add_set.
    3. rem_set stores the items being removed from the main_set. Whenever an item is removed from the main_set, it’s inserted into the rem_set.
- The add_set and rem_set are fully broadcasted over UDP using another thread every 1 second, as we’re implementing a state-based CRDT.
- When a replica receives a UDP packet, it first iterates over the received add_set and inserts items from it into the main_set based on two conditions:
    1. The item is not already present in its local add_set (ensuring that the item is not present in the main_set).
    2. The item is not present in the received rem_set over UDP (skipping the extra operation of adding and removing the item later, which is useful for replicas with no initial state, such as newly added replicas).
- If the item is found in the received rem_set and also in the local main_set, it gets removed from the main_set.
- After iterating over the received add_set, it iterates over the received rem_set and removes items from the main_set, if found.

- Whenever an item is inserted or removed from the main_set, the item gets inserted into the local add_set or rem_set accordingly.

In case of the failure of a node or if a node is newly added, they will eventually catch up with the same final state as other replicas by the above implementation.

The above points prove the convergence property, i.e., eventually, the main_set ends up in the same state irrespective of node failure or newly added nodes.

Commutativity

This property ensures that no matter the order in which the set operations (insertion and deletion of integer values) are performed upon receiving by the replicas, it won't affect the final state. For example, inserting two items into the set in any order won't affect the final state of the set.

Proof:

- Let's say there are three nodes: A, B, and C.
- A gets a request to insert element 5 into the main_set. 5 gets inserted into the main_set and also into the add_set.
- At the same time, B gets a request to insert element 8 into the main_set. 8 gets inserted into the main_set and also into the add_set.
- Both A and B broadcast their states.
- A's main_set now contains {5, 8}
- B's main_set now contains {5, 8}
- C's main_set also contains {5, 8}, no matter in what order it received the UDP packets from A and B. Eventually, the insertion of the items into its main_set will end up in the same final state and won't affect it. The implementation as described ensures this.

Associativity

It ensures that rearranging or grouping the operations won't affect the final state. In this implementation, it always holds the associativity property.

Proof:

- Operations on sets are inherently associative.
- For example, applying (insert(a), insert(b)) or insert(a), (insert(b)) results in the same final state {a, b}.
- Similarly, for removal operations: (remove(a), remove(b)) is equivalent to remove(a), then remove(b), resulting in the same state.

Idempotency

This property ensures that performing the same operation multiple times won't affect the final state.

Proof:

- If an item is already present in the main_set, another insertion request for the same item won't affect the state of the set. Upon receiving an insertion request via UDP by the add_set, it won't affect the final state as the implementation first checks if the element to be inserted via the UDP packet is already present in the local add_set or not. Whenever an item gets inserted into the main_set, it also gets inserted into the local add_set.
- Similarly, for the rem_set in the received UDP packet, it gets compared with the local rem_set, and only items not present in the local rem_set are stepped up for removal from the main_set.
- Applying the same operation on the set is inherently idempotent. In our case, we ensure idempotency on the basis of CRDT, i.e., upon receiving multiple UDP packets with no changes at all in the add_set or rem_set, it isn't going to affect the final state of our sets.

Handling Reinsertions

- It is to be noted that, as we're using two different sets to maintain the insertion and deletion of items from the main_set, it won't allow us to insert the same item twice. For example, if an item is added first and then removed from the main_set, we won't be able to insert the same item again. To solve this issue, we're attaching a unique ID with each item in the set.
- Whenever an item is inserted using the /add endpoint, it gets linked with a unique ID and the pair gets inserted into the main_set and the add_set. When removing the item, the rem_set also stores the unique ID along with the item.
- This removes the conflict, ensuring that an item, irrespective of removal, can be inserted again whenever required or requested.