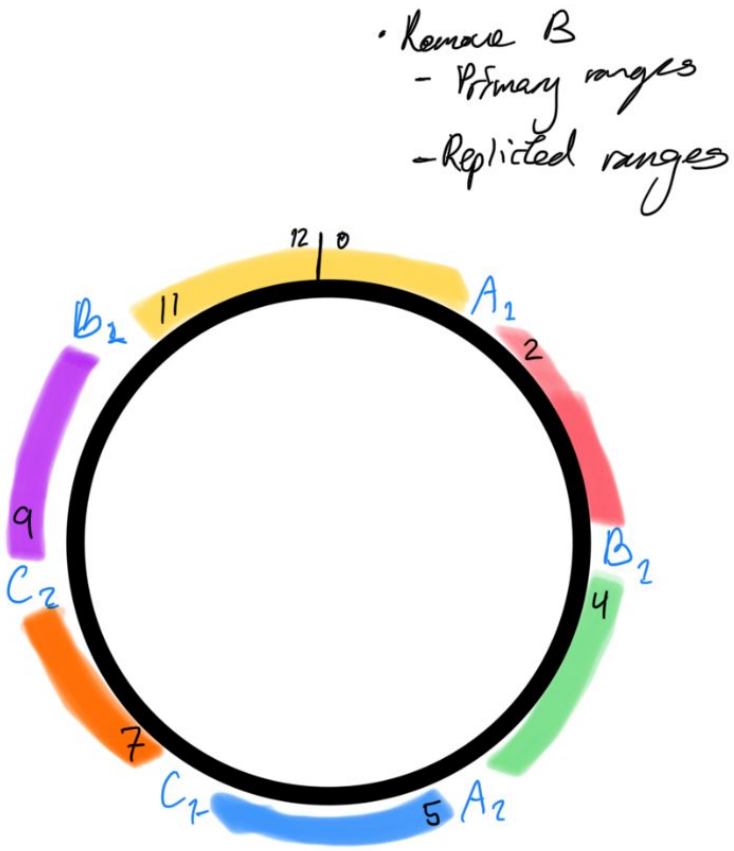
Requirements

1) Ensure causal consistency
    a) Define causal context
        i) Will be represented by both a vector clock and a timestamp (for breaking ties)
        ii) Vector clock would need to be the same size as the repl factor, we also need to reset the clocks when a view change occurs
2) Replication / Durability
    a) Preference list idea (linear search for edge cases)
    b) Shard id's pertain to tokens (talk/ask on slack)
3) Highly Available / Fault tolerant
    a) Timers will be used when there are partitions

Replication

- ➔ Sharding implemented with consistent hashing
  - ◆ Each shard is replicated at R - 1 previous unique tokens

	Primary	Replicated
A	2-4 5-7	4-5 7-9 9-11
B	11-2 4-5	5-7 2-4
C	9-11 7-9	11-2



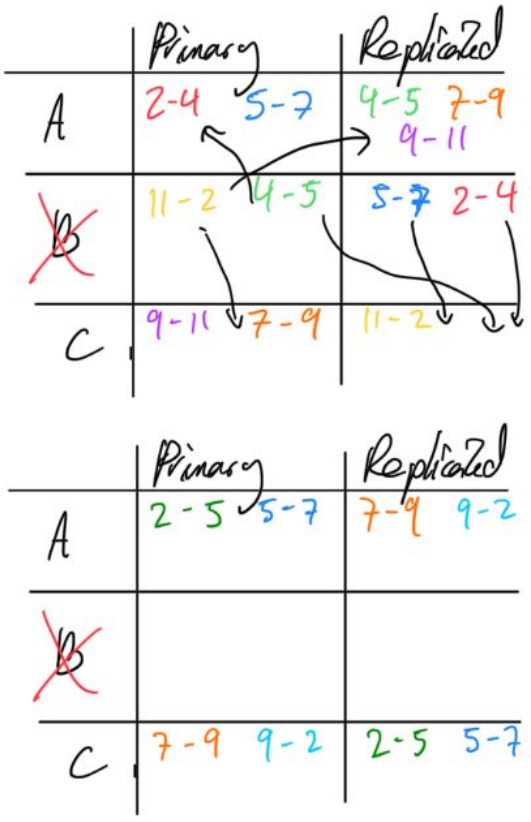
Fault Tolerance

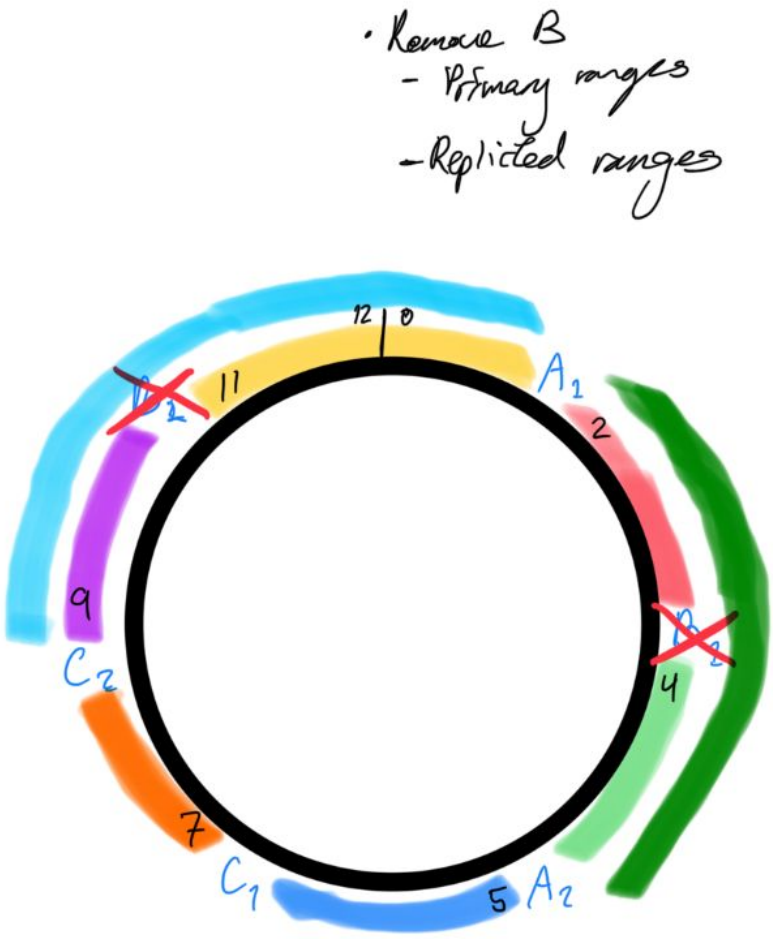
- ➔ Causal context represented with vector clocks with a length of the replication factor as well as time stamps for tie breaking
  - ◆ Each key has a causal context associated with it
- ➔ Key Operations
  - ◆ Put
    - Any node can process put request and will simultaneously write to all replicas of shard
    - Each write increments the vector clock for that node
    - Only one replica must respond correctly for successful put
  - ◆ Get
    - Any node can process get request and will sequentially read from each replica until it gets a valid response
    - Only one replica must respond correctly for successful get
  - ◆ Delete?

Eventual Consistency

- ➔ Every 5 seconds each node iterates through every node in its view
  - ◆ The initiating node sends a list of its tokens
  - ◆ The receiving node finds the shared tokens and returns the causal contexts for each of the shared keys for a shared token, sends values for each key for each token as well
  - ◆ The initial node then compares the contexts and updates values for out of date keys

View Changes





- ➔ Case 1: Removal of a Node (B)
  - ◆ How does merge tokens get affected?
  - ◆ Compute ip addresses affected by token removal
  - ◆ Need added and removed
  - ◆ Gossip and view changes need to be mutexed

Revised Design:

- We need to change number of tokens to be 1 per node
- This means during the setup phase, we would want to divide number of nodes in view by repl factor to determine how many unique shards to create, once these nodes are set up we can go ahead and replicate to the remaining nodes in the view
    - Eg: starting view = [1,2,3,4,5,6], repl_factor = 2
    - One way would be to choose the first len(view) / repl_factor nodes to have unique tokens being created
    - Once we receive confirmation that the tokens are created
- We probably only need to store positions for nodes with unique shards in the hash ring
- The question is how do we deal with view changes?

Case 1: Node removals and repl factor decrease

Prev: [“1”,”2”,”3”,”4”,”5”,”6”], repl factor = 3
Shards: [1: 1,2,3; 2: 4,5,6]
Tokens for 1(37) and 2(79)
Tokens: [37,79]

New: [“1”,”5”,”6”], repl factor = 1
3/1 = 3 - 3 shards
Shards: [1: 1; 2: 5,6] -> [1: 1; 2: 5; 3: 6]
Tokens: [37,79,100]
Changes: [79,100]

Steps:

1) Calculate new number of shards to have and how many nodes pertain to each shard
2) Create a map of new node ip to a boolean, added and removed node arrays will be created, but tweaked once we establish a new shards list
3) Initialize a changed nodes array (empty)

we will be making a copy of the current shards list and then performing updates on that until we get an updated shards list
#part 1 of view change would be to remove or reassign any nodes in the shards list

Case 1: node is no longer in the view, it's the same thing as being removed
Case 2: node is in the view, and is still within the repl_factor constraints for a given shard, it’s the same thing as a node not needing to change unless its predecessor has changed

4) Iterate through each shard
    a) Iterate through shard list of shard

i) If node doesn't exist, we append the node to changed nodes (removed=true)
ii) We don't want to be having all replicas resharding to the same spot, need a check to see if we need to reshard or no

b) If it does exist, perform another check to see whether its ok where it belongs, otherwise we may need to either create a new shard and assign node to that shard, or assign node to an existing shard which has fewer than repl_factor nodes assigned to it

#part two would be looking into the added nodes and seeing where they would fit into the new shards list

5) Compare the old shard list and new shard list, and compute which nodes are changed as a result (node's associated shard still the same, node has been removed and thus must recompute the new home of all the keys/values it stores, node still exists but has been moved to another shard, it will reshard its keys and values and then take on new keys and values)
6) Therefore we can repartition tokens as normal, and replication will occur after repartitioning where we look at the new shards list and initialize nodes with empty kvs with the correct keys and values

Case 2: Repl factor decrease

Prev: ["1","2","3","4","5","6"], repl factor = 6
Shards: [1:1; 2:2; 3:3; 4:4; 5:5; 6:6]
Tokens: [13,29,37,53,71,87]

New:

- Nodes list is tweaked to be only unique ips
- Mapping between id to replicas and replica to id perhaps needed

➔ Adding new shard endpoints
➔ Modifying existing kvs endpoints
  ◆ Include replicas
  ◆ Defining causal context
➔ Gossip
  ◆ Background process in event network partition
  ◆ Kvs operation propagation
➔ View changes
  ◆ How to represent state
  ◆ Modifying token logic

Ultimately we managed to accomplish replication, and consistency for the very first view. We unfortunately ran out of time to implement view changes and causal context.