Reflection:

Q1) How does what you've implemented compare to how you might implement similar functionality on top of Bayou? What parts of this would become easier on Bayou (e.g., with custom conflict resolvers), and what might become harder?

Ans1) Bayou at the first sight, looks very similar in terms of functionality of eventual consistency that the Raft algorithm promises. However, for this assignment, implementing the functionality of unstable reads in the form of tickets differs in both implementations in the following ways:

1) Bayou runs on deterministic conflict-detection and merges procedures. Determining what entries from the client would result in conflict would be difficult until calculated in the server replica. In the raft, this work is done during the pre-commit process so would be easier in the homework's implementation rather than in Bayou.

2) With each node getting requests from clients and servers, Bayou checks for consistency is difficult as compared to the Raft implementation as the program has to perform more fixes in logs to undo the effects of tentative execution. In Raft, the only leader would replicate logs and deal with the client request, making the consistency checks easier for the logs.

3) Raft guarantees the consistency of log committed among the nodes if the algorithm followers correctly, implementing these rules in Akka would be easier than that in Bayou. As Bayou is a per-write dependency check system, we have to implement Raft rules for every write (commit) operation and make more rules for defining which write would produce conflict in Bayou.

4) In case of finding inconsistent logs at follower, the leader tries again by sending logs starting from one more previous log entry. This way logs are replicated and eventually become consistent. In Bayou, implementing this conflict resolver would be difficult as have to implement a custom mergeproc procedure to maintain this check.

5) Available tickets might take more time to be accurate in Homework's implementation than in Bayou as Bayou uses Lamport clocks to advance the ordering of writes. Whereas, Raft works on timeouts and elections to advance the writes. With the case of opting for 150ms to 300 ms timeouts, Raft could take some time to produce accurate available tickets.

6) The recovering mechanism in Bayou is much simpler and easier to implement to recover from crashes (internal) as compared to the simple reading from the disk. Bayou uses the timestamp to compare the writes and calculate its consistency, with simply reading from the disk, the program can't fully recover the last state. Checkpoint helps in regaining the last state easily in the case of Bayou. To implement something similar in Raft would be difficult.

Q2) Try to prepare a workload for your system (sequence of operations for each frontend to issue) that will maximize the observed error between the committed ticket count and observable (with unstable reads) ticket count. How big a discrepancy can you observe in your code between an unstable read and a consistent read that is issued by the same client immediately after the unstable read (after doing whatever you think will lead to significant divergence)? Please do submit the workload/configuration that would do this experiment.

Ans2) With my implementation, the node is printing its commit consistent state, and the clients print the response for the unstable read from the respective nodes. I have started with a small number of tickets i.e 20, then tried out with the number of tickets to be 50. With more than 50, the client takes more time to send the final log entry. With the number of tickets to be 50, and log size corresponding to this number, and the clients asking for the unstable read every 2000ms, the discrepancy seen is about 6-8 tickets. This is the peak discrepancy after a few more iterations, this discrepancy reduces, and eventually, the state becomes consistent. The current program runs through this same client configuration so, results can be generated by simply running the given program.

The workload for the client:

[-1, -2, -1, -3, -5, -6, -2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1] = -45

Left over tickets = 50 - 45 = 5

Q3) Look at the pathological cases in the Raft paper, such as the example showing why leaders only commit entries from their own current term. Is there a theoretical limit on how far off an inconsistent read can be from a consistent one in your implementation? What factors that arise in practice (e.g., hardware, request rates, etc.) might effectively impose some limitation on how far apart those can get?

Ans3) Two of the main reasons for bigger discrepancies are getting enough votes from the followers before committing the log, and the client sending msgs fast enough to append in the leader's log. This occurs when running the algorithm on the same machine. Only after the leader's entries are committed, and increased commit length, then the followers would be able to append and apply for their entries, till the gap between stable and unstable read would increases.

Also, if we have a leader for a considerably long time, the gap decreases as logs are starting to match, and with a new log entry, the commit would be faster. The more elections program would have, the gap increases as the leader would not have enough run-time to replicate its logs onto other nodes.

If Client msgs are getting dropped more frequently, then the nodes would have entries in their logs in the incorrect order, resulting in an incorrect state. For the unstable read, the gap would be even more. Likewise, the gap would increase if leaders’ msgs are getting dropped but some are reaching before the followers’ timeout so taking more time to achieve consistency. If this leader with a network delay stays leader for a long time, followers’ log would not be committed due to the commit check and stays in the unstable state for a long time.