DISYS exam 2022 - Jacob Grum

I hereby declare that this submission was created in its entirety by me and only me.

I re-used my own code from assignment 3-5, and altered it to fit this exam. Assignment 3-4 (Chitty chat and mutual exclusion) helped with peer-to-peer architecture and locks. Assignment 5 (Auction house), helped with passive replication.

Questions about implementation

Description of the architecture of the system

Features:
Data structure: Map – For clients, stored responses and for the distributed dictionary.
Concurrency: Go-routines – For heartbeats and leader election, while using the system
Communication - simple gRPC functions, meaning no streaming is used

Architecture:
The architectural model is Peer to Peer – With an "all-in-one approach", as a peer functions as both a front-end with a command line, but also every peer is a replica. Thus, allowing us to use 2 nodes, to simulate the whole system.

Assumptions of our system model:
Network: Reliable. Messages is received if they are sent and may be re-ordered.
Nodes: Crash-stop. After a node crashes, it stops executing forever.
Timing: Partially synchronous. Our lack of complete asynchronously is due to heartbeats. We have timebound in the form of the heartbeat, as we expect the heartbeats to happen within at most 5 seconds. A heartbeat is sent every 300 milliseconds. The rest of the system does not use any timings. That means we potentially could have an algorithm run for 1 second or 1 hour, as long as the heartbeats checked on a different go-routine are within our timebound.

a) How does your system satisfy points 1-7 (Why read/add) is correct.

Add: Due to the map data structure, it will automatically overwrite the old value. I utilize 2 maps; One for storing the unique request id, generated on every new request, along with the response. The second is for the distributed dictionary. There is also a lock, preventing race-conditions, such as multiple clients adding the same time. If the request has already been processes before, by checking our handled requests map, we will resend the answer and return, as replicas don't need to be updated. If we haven't processed the request, we will add the word and definition to our distributed dictionary, along with adding the unique id of the request and the response, to our other map. This setup with at-least-once semantics, enables us to keep retrying a request, if we don't receive a response, thus giving us liveness of our requests, and ensures uniqueness of the response. This is smart, if a client sends a request, just as the primary crashes, the request will instead be forwarded to the new primary after election.

Read: As this operation is read-only, I implemented it with at-most-once semantics. When calling read(), the same lock is grabbed, ensuring that the dictionary is not updated in between calls, and just returns to the client the entry of the map. Since there is no re-sending, a client would have to manually request a read operation again, if they were to send it to a crashed primary. Since no data

is updated and this operation is idempotent, and there is no danger re-sending the same read operation.

b) Argue why your system satisfies the requirement in point 8

Same lock between add() and read(), and all requests are handled by a single node, the primary, meaning there will be no race-conditions between add and read.

When a call to Add(word, definition) is made, our client will continue in a loop, to check for the response to have been added to the requests handled map, and will afterwards release the lock. Therefore if we attempt to Read, while this add operation is ongoing, we will have to wait for the lock to released. Therefore Read(word), always returns the latest value, as when reading the lock is taken, blocking any concurrent calls to Add(), therefore always giving the latest value of the last processed call to Add(). Thus, any subsequent read will always give the latest value, until a new Add(word, definition) has been made.

c) Argue why the system is tolerant to one crash-failure

A primary will keep sending heartbeats through a go-routine to all the backups. The backups keep checking the heartbeats through their own go-routine. If the last heartbeat is over 5 seconds ago, we assume the primary has crashed, and a new election takes place. The election is a simplified bully process. All peers, has access to other peers id, and if the peer is the highest Id, it will take place as primary, otherwise it will take place as backup. Thus it doesn't matter if we crash a backup or a primary, the system will still continue to function. To summarize, with N peers, we can tolerate N-1 crash-failures, and since we have 2 Nodes, our system is tolerant of one crash-failure.

Leader election – Bully like:
Safety: Every process chose a leader, and also selects the same leader. As the Ids are unique, only a single leader can have the highest Id and thus only 1 process can send coordinator to all other processes declaring itself as the winner and rest as backups, and shows that the leader algorithm is safe.
Liveness: Lets assume that the leader crashed right before it announces itself with coordinator(). Then there would be no leader, and thus none of the backups would receive any heartbeats, and therefore after a timeout, they will start a new election process and removes the old leader from their Map of alive clients. During this new election process, since the old leader has been removed, another unique Id will now take place as the highest and become leader.

<u>Difference between my bully and a real bully implementation:</u>
In a real implementation, we send Election/Answer to other processes, meaning they inform each other about an election going on. In mine, i just re-use the logic for heartbeat, meaning the processes find out about the election themselves, by not receiving a heartbeat by a leader. But I still send Coordinator(), meaning even if a process is unaware of an election, they will still receive message when a victor has been selected, and thus know about any new leaders, without knowing an election has taken place. My implementation works only because, every peer knows every peer. Whereas a real bully would work for example if not every peer knew every peer. Such as P1 knew P2, P3 and P3 knows about P4 , P5. In this scenario my bully would fail, as P1 would select P3 as leader, but P3 would select P5.

d) Argue whether your system satisfies linearisability and/or causal consistency.

In the second passive replication phase, coordination, our primary handles each request atomically in the order the primary receives it. Thus we are not guaranteed that if P1 sends a request before P2, that it will be handled in that order. All we can ensure is that if P1 sends 3 requests, that they will be handled in the correct order of that individual process. This might be interleaved with requests of other peers. Resulting in an interleaved ordering which might be
P1_1 < P1_2 < P2_1 < P1_3 < P2_2 < P2_3. But this interleaving meets the specification of a single object performing these actions, which in our case is always the primary, thus always ending up in a state, where all primaries and backups eventually have the same data. Therefore, we satisfy the properties of **sequential consistency.**

If we instead were to satisfy the real-time ordering, we would have linearizability.

The queue of the messages are bound by the lock, first to access it gets it. To abide causal consistency, giving a partial order, we would have to ensure a happened-before relationship on the messages, such as using a lamport timestamp for the queue, processing the lowest timestamp first.