tcla

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

System architecture and assumptions

Assumption 1: The system is setup as intended before crashes occur. This means a node cannot crash before all clients are connected to the leader node and no clients are connected to the backup node before the leader node crashes. Otherwise, errors may occur.

Assumption 2: A healthy node responds within 3 seconds. If a node takes longer than 3 seconds to respond it is declared dead.

The communication between a client and a node is of client server architecture meaning clients can send requests to a node and the node will respond.

The communication between the leader node and the backup node works similarly. The leader sends updates to the backup and the backup responds.

The leader is chosen to begin with by looking at which server has the lowest port number.

Setting up the system

The zip folder contains a README file that explains the setup. You can also see the instructions in appendix.

a) how does your system satisfy points 1-7

1, 2) it provides two operations called 'add' and 'read'. Operation 'add' has two parameters, a word and its definition, and returns a boolean value. Both input parameters are strings.

I made a function for both operations. This can be seen in the files server.go and client.go
The template for these gRPC functions is defined in the template.proto file.

3, 4) operation 'add' updates the dictionary at the requested entry with the new definition. Operation 'add' returns a confirmation of whether it succeeded or not.

When the client calls the add function on the server, the server uses a map from a string to another string to save the words (as key) with their definition (as value). If the server receives an empty string in either of the parameters it returns false and does not update the dictionary. Otherwise it adds the word and definition to its map (after contacting the backup) and returns true.

5, 6, 7) operation 'read' has one parameter (a word) and it returns a string. Operation 'read' accesses the entry specified by the given word. Operation 'read' returns the value associated to the given word (its definition stored in the dictionary)

When the method is called in the server.go file it uses a map as dictionary. It takes the parameter that is a word and uses it as key to retrieve the associated value from the map which is the definition.

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

8) the system is such that if a call add(word,def) is made, with no subsequent calls to add, then a new call read(word) to that node will return def.

Whenever a call to add happens the server will add the key-value pair to the map. It uses a mutex lock to ensure no race conditions occur when updating the map which means the last received call will determine the final value as it overwrites every time an existing word is added. Every read after the last add call will return the value that ended up in the map if it is the same key.

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

If the backup node crashes and fails to respond to the leader node. The leader will simply stop attempting to contact backup and just act as a single node handling requests as usual. If the leader node fails to respond to a client the client code will redirect it and connect to the backup. The backup then acts exactly as if it was a single node like when the leader loses its backup.

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

My system satisfies causal consistency. Every add operation happens in a certain order since the mutex lock prevents them from happening concurrently. However, read operations do not have this restriction. They may happen concurrently and can therefore not be totally ordered.

Appendix

```
# Dsys-exam

## Running the program

Open two terminals for the two nodes and a terminal for each client.

In the server terminals use the command: go run \[path to server.go\] \[number\]

Replace \[number\] with 0 for the leader node and 1 for the backup node.

In the client terminals use the command: go run \[path to client.go\]

Once connection is established you can now enter text lines in the client
terminals.

You can simulate a node crash by using ctrl+C in a node terminal or by entering the
line "end".
```