Take one of the implementations you created for either of the first two design exercises (the chat application) and re-design and re-implement it so that the system is both persistent (it can be stopped and re-started without losing messages that were sent during the time it was running) and 2-fault tolerant in the face of crash/failstop failures. In other words, replicate the back end of the implementation, and make the message store persistent.

The replication can be done in multiple processes on the same machine, but you need to show that the replication also works over multiple machines (at least two). That should be part of the demo. Do not share a persistent store; this would introduce a single point of failure.

As usual, you will demo the system on Demo Day III (March 26). Part of the assignment is figuring out how you will demo both the new features. As in the past, keep an engineering notebook that details the design and implementation decisions that you make while implementing the system. You will need to turn in your engineering notebook and a link to your implementation code. As always, test code and documentation are a must.

Extra Credit: Build your system so that it can add a new server into its set of replicas.

Idea we are using: primary-backup

- Persistent storage: use a sql database
    - Use SQLite3: https://www.geeksforgeeks.org/sql-using-python/
    - Users table (username password etc.)
        - PK - username (string)
        - Password (str)
    - Messages table
        - PK - hashed_id (uuid)
        - FK - Recipient: username (str)
        - FK - Sender: username (str)
        - Messages (str)
        - Timestamp (DateTime)
        - Delivered (bool)
        - Instant (bool)
            - Marks if a message should be instantly delivered or not
- Multiple servers: create 3 copies of server.py, each with its own persistent storage
- Fixed retry list for client (try 5001, 5002, 5003 before giving up, store in env)
- Leader election is also via this fixed retry list, store in env - if leader 5001 goes down, leader becomes 5002 unless 5002 is also down, then is 5003
    - Use “heartbeat signals”? Need to change grpc + proto to have that function
        - If the replica sees the leader dies, elect a new leader.
        - Add gRPC function
    - Need to figure out what gRPC returns if a connection is closed/server dead

- Client talks to leader, leader sends messages to the other 2 servers and waits for all (alive) ones to reply that update has been made (ACK)
    - Do this atomically basically ~~(max_threads = 1)~~ use locks
    - If the request is a PURE READ (ex. list accounts) then the leader can just return without talking to the replicas
- Still todo:
    - When a new client sets up, it asks each port whether or not it's the leader server to decide who to talk to
        - isMaster() proto/grpc whatever needs to be set up
        - When a client gets an error from a grpc function, it should run a function that just goes through the list of potential ports and asks who is the master
    - If a "non-master" server starts being talked to by a client, it just returns an error forever - DONE
    - If a commit attempt errors out when talking to a port, we just assume they died rip
        - Remove them from active_ports - maybe done? Maybe buggy
    - Heartbeat messages/health check let everyone know if someone is dead

Extra Credit: Build your system so that it can add a new server into its set of replicas.

- Future
- Key issue: how can we bring the new server up to date in a timely fashion?
- How to maintain leader election/retry consistency: have a .txt with a lock that has all live servers so that everyone knows who's alive (or something along those lines)