Replicated log

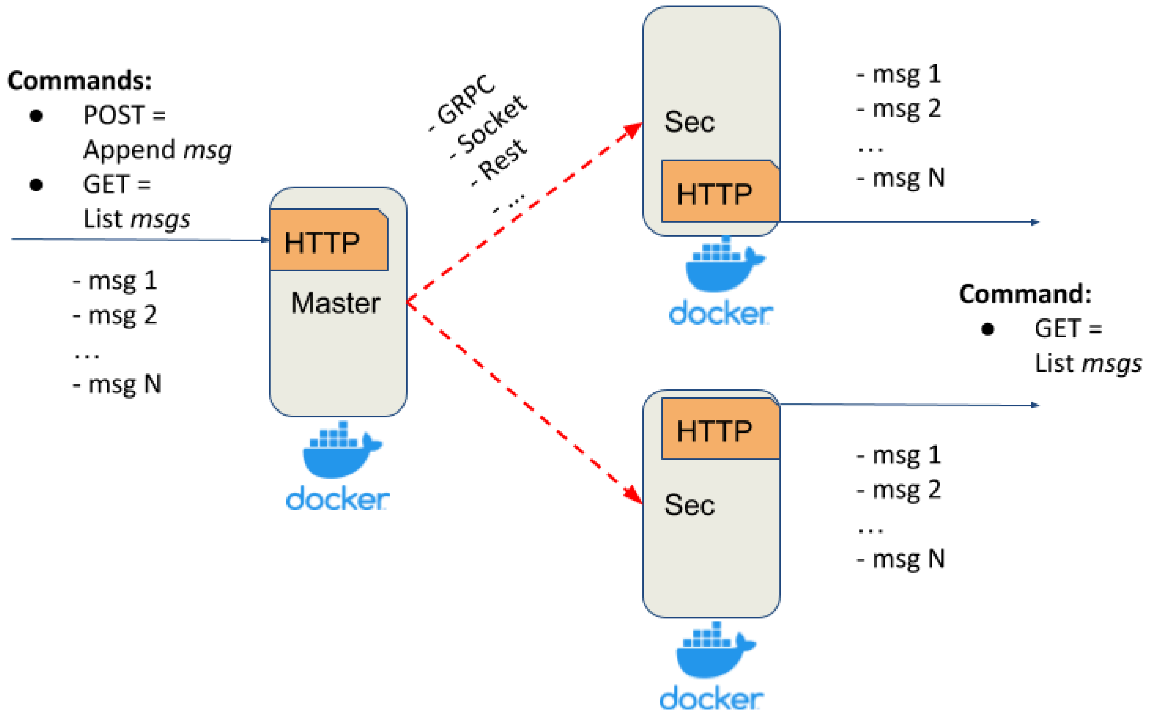
Iteration 0.

Choose a desirable language for implementation and try to implement (or find the implementation) a simple *Echo Client-Server* application.

Iteration 1.

- 10 points

The Replicated Log should have the following deployment architecture: one ***Master*** and any number of ***Secondaries***.



Master should expose a simple HTTP server (or alternative service with a similar API) with:
- *POST method* - appends a message into the in-memory list
- *GET method* - returns all messages from the in-memory list

Secondary should expose a simple HTTP server(or alternative service with a similar API) with:
- *GET method* - returns all replicated messages from the in-memory list

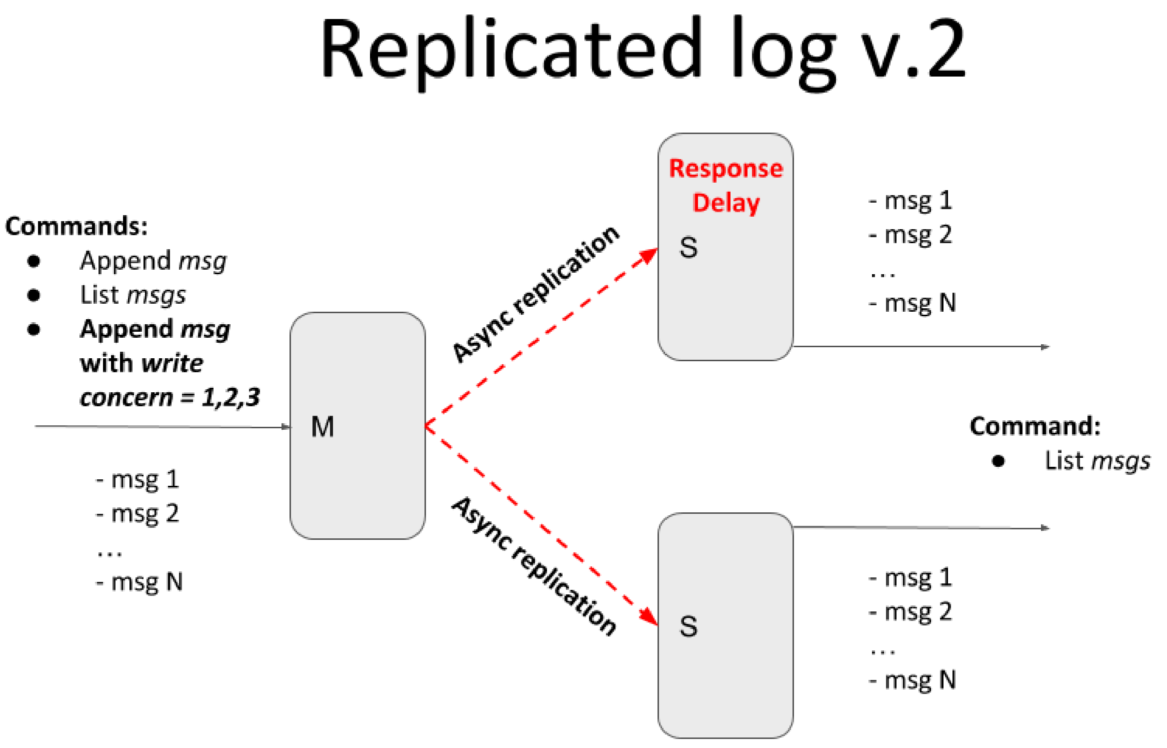
Properties and assumptions:
- after each POST request, the message should be replicated on every *Secondary* server
- *Master* should ensure that *Secondaries* have received a message via *ACK*

- *Master's POST request* should be finished only after receiving *ACKs* from all *Secondaries* (blocking replication approach)
- to test that the replication is blocking, introduce the delay/sleep on the *Secondary*
- at this stage assume that the communication channel is a perfect link (no failures and messages lost)
- any RPC framework can be used for *Master*-*Secondary* communication (Sockets, language-specific RPC, HTTP, Rest, gRPC, …)
- your implementation should support logging
- *Master* and *Secondaries* should run in Docker

Iteration 2.

- 20 points

In the previous iteration, the replication was blocking for all secondaries, i.e. to return a response to the client we should receive acknowledgements (ACK) from all secondaries.



Current iteration should provide tunable semi-synchronicity for replication, by defining *write concern* parameters.

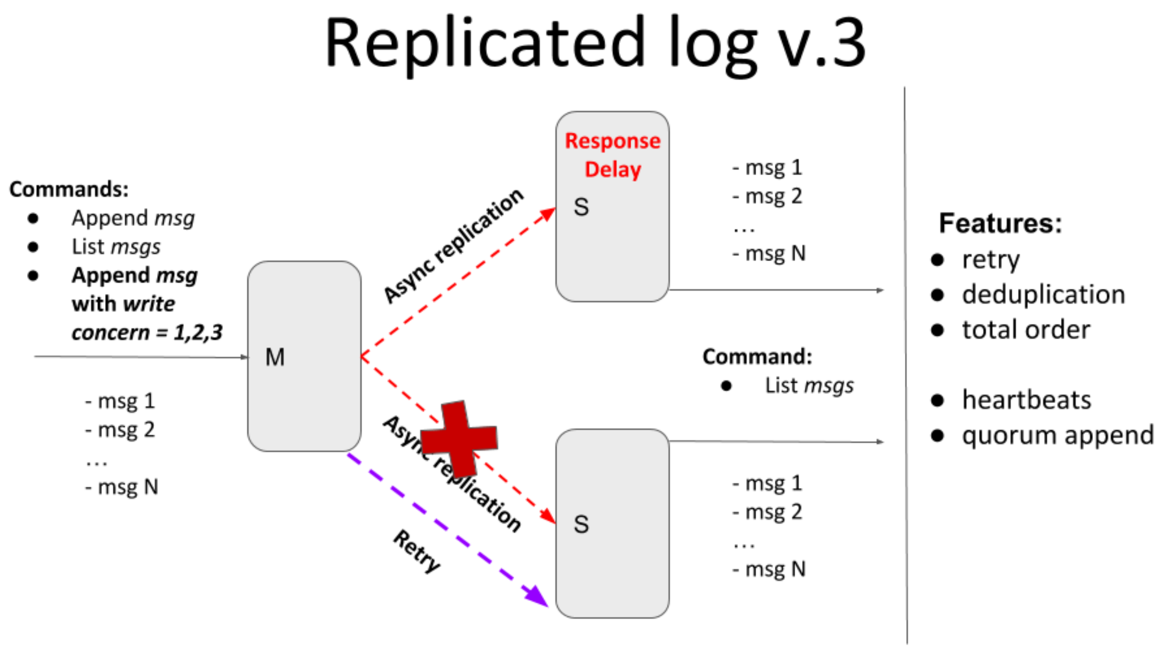
- client POST request in addition to the message should also contain *write concern* parameter *w=1,2,3,..,n*
- *w* value specifies how many ACKs the master should receive from secondaries before responding to the client
  w = 1 - only from master

w = *2* - from master and one secondary
w = *3* - from master and two secondaries

Please emulate replicas inconsistency (and eventual consistency) with the master by introducing the artificial delay on the secondary node. In this case, the master and secondary should temporarily return different messages lists.
Add logic for messages deduplication and to guarantee the total ordering of messages.

Iteration 3.

- 20 - 40 points



The current iteration should provide tunable semi-synchronicity for replication with a *retry* mechanism that should deliver all messages *exactly-once* in total order.

Main features (maximum 20 points):

- If message delivery fails (due to connection, or internal server error, or secondary is unavailable) the delivery attempts should be repeated - *retry*
    - If one of the secondaries is down and *w*=*3*, the client should be blocked until the node becomes available. Clients running in parallel shouldn't be blocked by the blocked one.
    - If *w>1* the client should be blocked until the message will be delivered to all secondaries required by the write concern level. Clients running in parallel shouldn't be blocked by the blocked one.
    - All messages that secondaries have missed due to unavailability should be replicated after (re)joining the master

    - Retries can be implemented with an unlimited number of attempts but, possibly, with some “smart” delays logic
    - You can specify a *timeout* for the master in case if there is no response from the secondary
- All messages should be present exactly once in the secondary log - *deduplication*
    - To test deduplication you can generate some random internal server error response from the secondary after the message has been added to the log
- The order of messages should be the same in all nodes - *total order*
    - If secondary has received messages *[msg1, msg2, msg4]*, it shouldn’t display the message *‘msg4’* until the *‘msg3’* will be received
    - To test the total order, you can generate some random internal server error response from the secondaries

Self-check acceptance test:

1. Start M + S1
2. send (Msg1, W=1) - Ok
3. send (Msg2, W=2) - Ok
4. send (Msg3, W=3) - Wait
5. send (Msg4, W=1) - Ok
6. Start S2
7. Check messages on S2 - [Msg1, Msg2, Msg3, Msg4]

Additional features:

- Heartbeats (+15 points)
  You can implement a heartbeat mechanism to check secondaries’ health (status): *Healthy -> Suspected -> Unhealthy*.
  They can help you to make your *retries* logic smarter.
  You should have an API on the master to check the secondaries’ status: *GET /health*

- Quorum append (+5 points)
  If there is no quorum the master should be switched into read-only mode and shouldn’t accept messages append requests and should return the appropriate message