CSE 486/586 : Class Project Handout

RAFT (Total Points: 35 + 5 bonus points)

Phase 4 : Safe Log Replication with leader election

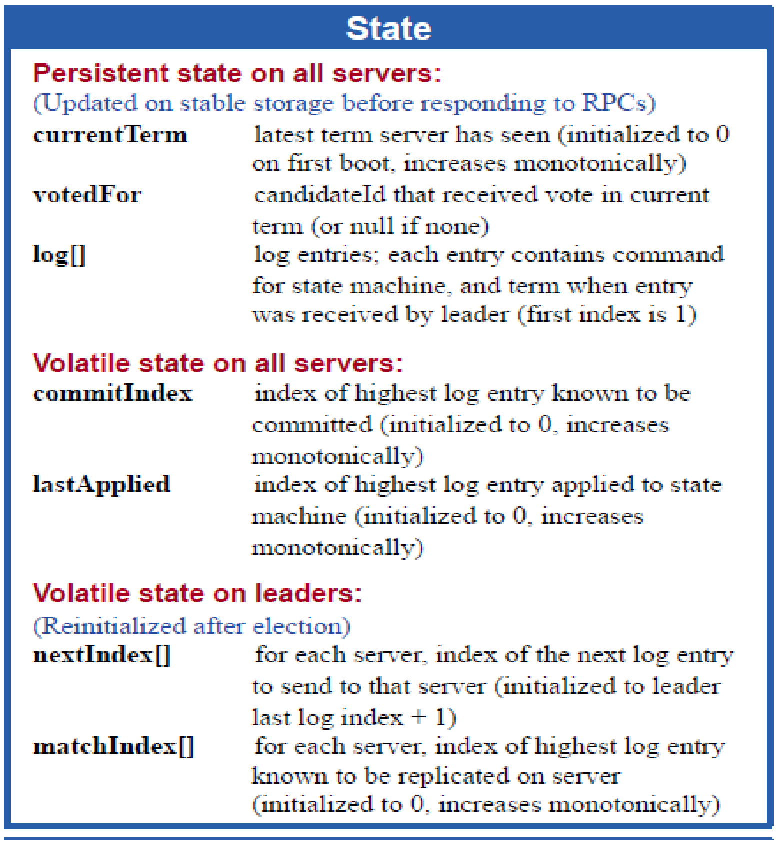
Deadline : 5/6/2022 11:59 PM (FRIDAY)

In the previous phase, we worked on leader election and how a new leader would be elected when there is a timeout. To recap, once a follower times out, it increases its own term number, converts to candidate state and sends out RequestVote RPC to all the other nodes. On the receiver side, the node when it receives a RequestVote RPC, it would compare its own term number with the candidate's term and it's own log entries with the lastLogIndex and lastLogTerm values (sent out during RequestVote RPC) and decide whether to give a vote or not.

In phase 3, the lastLogIndex and lastLogTerm comparisons were trivial and not considered as the logs entries were always empty as no client requests were being made. However in this phase we will see how these comparisons play an important role in deciding who can and who cannot become the leader.

Note : All references to AppendEntry RPC's also include heartbeats since both are practically the same thing

Prerequisites of what each server should store



State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

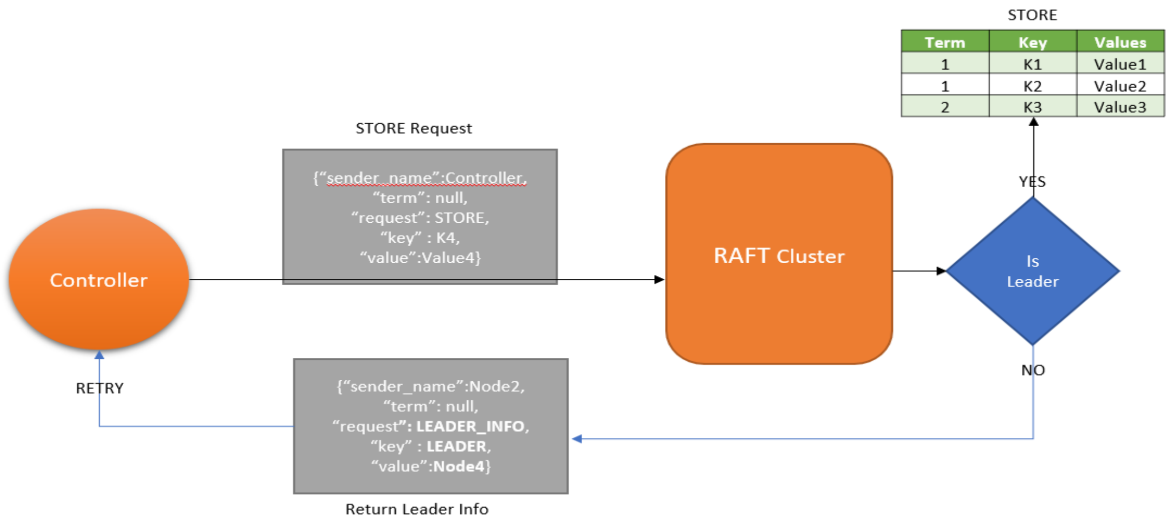
commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

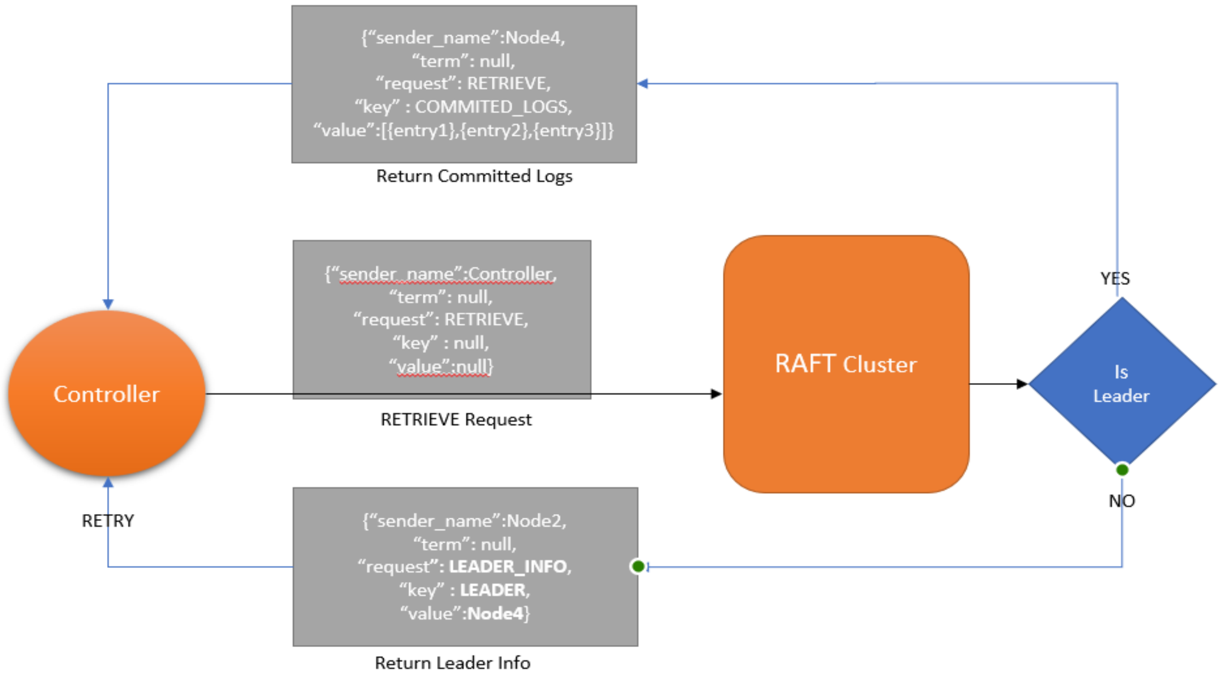
nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

Handling additional requests that will be sent out by controller. STORE & RETRIEVE (5 points)

STORE
The controller from phase 3 has a new purpose in this phase. The controller will be used to send out STORE requests to the RAFT cluster. This is similar to the client making a request to the RAFT cluster where the request gets appended to the log's of the leader and eventually gets appended to the follower's logs. You will be using the STORE cmd to make client requests.



RETRIEVE



The controller can send a request to any of the nodes to RETRIEVE all the committed entries at that particular node. However, only the Leader will respond with the committed entries**[{entry1},{entry2},{entry3}]**, any other node responds with Leader Info as depicted in the diagram above. Also, the format of what constitutes an entry is as below -

entry = {
"Term": *Term in which the entry was received(may or may not be current term)*
"Key": *Key of the message (Could be some dummy value)*
"Value": *Actual message (Could be some dummy value)*

}

Note 1 : For testing purposes, you will require a listener thread at the Controller end to receive messages from the leader. We will use our controller(hidden test-cases) to evaluate. As long as you follow the naming conventions in the JSON message and you are handling all the specified controller requests on the server-node's side there will not be any issues in testing.

Safe Log Replication with consistency checks: (15 points)

In phase 2, you would have built a simple mechanism of forwarding the client's request from the leader node to all the followers and ensuring that the request is executed on all the nodes.

In RAFT, the request first gets appended to the log's of each node and once the request has been appended to a majority of the node's logs, the request is said to be committed and will be executed by the leader. This is an extremely simplified explanation of what actually happens. There are certain rules to a client req being appended to the followers logs, whether a follower will accept or reject an AppendEntry RPC and how the logs are replicated. Details in the paper

The leader maintains a ***nextIndex[]*** for each follower which is the index of the log entry that the leader will send to that follower in the subsequent AppendEntry RPC. When a leader first comes to power (when a candidate first changes state to a leader) it initializes all ***nextIndex[]*** values to the index just after the last one in it's own log, this will happen every time a leader is elected.

Eg, If the last committed index on the new leader's own log is 5, then the leader initializes **nextIndex[]** for each follower node as 6 which means the leader is prepared to send the entry at position 6.

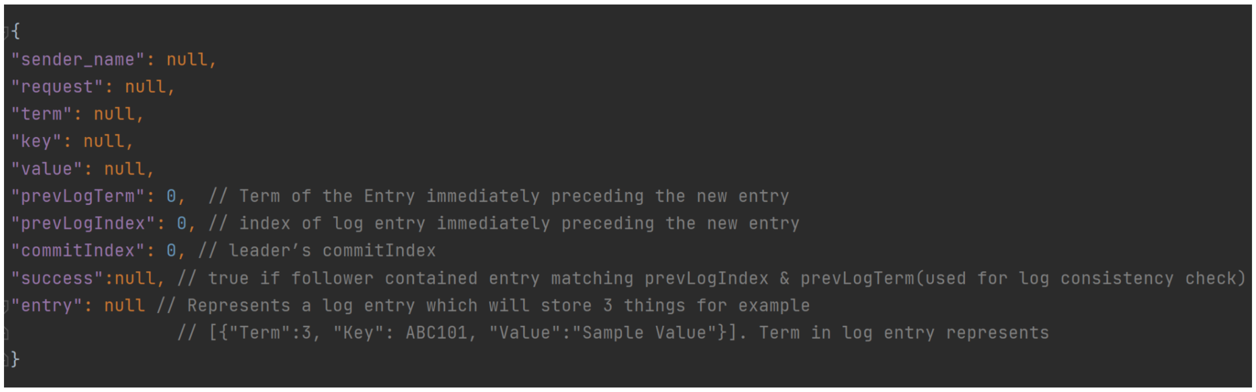
The AppendEntry RPC is used to replicate the log's on the follower nodes.

1. The client's request gets appended to the leaders's log
2. Based on the next index, the leader sends this request to the followers by passing the entry(or request) in the AppendEntry RPC along with the prevLogIndex, prevLogTerm and leader's commitIndex
3. The follower will check to see if needs to accept or reject the the AppendEntry RPC by performing a **log consistency check**(read RAFT Paper)
4. If the follower accepts, then it appends the entry to its own log and sends an APPEND_REPLY with *success=True.* The Leader on Receiving a True will update the next index and match index values.
5. If follower rejects (checks rules for rejection in **Append Reply** section below), follower sends an APPEND_REPLY with success=False
    a. Once the leader receives a reject from the follower, the leader will decrement the **nextIndex[]** for that particular follower by 1, and will send the previous log entry at that position back to the follower.

   b. The leader will retry the AppendEntry RPC and eventually the we will reach a point where the leader and follower's log match
   c. Eg : If the leader, has nextIndex[] as 6 for a follower, but the follower rejects the leader's AppendEntryRPC, the leader will decrease nextIndex[] to 5 for the follower that has rejected. This happens until the follower will eventually accept the leader's AppendEntryRPC (This could be an empty AppendEntryRPC or an heartbeat)

Note 2 : Additionally, you will need to check if a particular entry has been replicated on majority of the followers(utilize matchIndex[]) and send that in the APPEND RPC to the followers which will apply that to their own logs for a final commit.

Note 3: A STORE req from the controller is not a trigger for the AppendEntryRPC. The AppendEntry RPC is triggered at regular intervals as a heartbeat (which is why it is also functioning as a heartbeat). The STORE req appends an entry to the leader's log and this new entry gets sent along to the followers in the subsequent heartbeat/AppendEntryRPC.



```
{
"sender_name": null,
"request": null,
"term": null,
"key": null,
"value": null,
"prevLogTerm": 0,  // Term of the Entry immediately preceding the new entry
"prevLogIndex": 0, // index of log entry immediately preceding the new entry
"commitIndex": 0, // leader's commitIndex
"success":null, // true if follower contained entry matching prevLogIndex & prevLogTerm(used for log consistency check)
"entry": null // Represents a log entry which will store 3 things for example
               // [{"Term":3, "Key": ABC101, "Value":"Sample Value"}]. Term in log entry represents
}
```

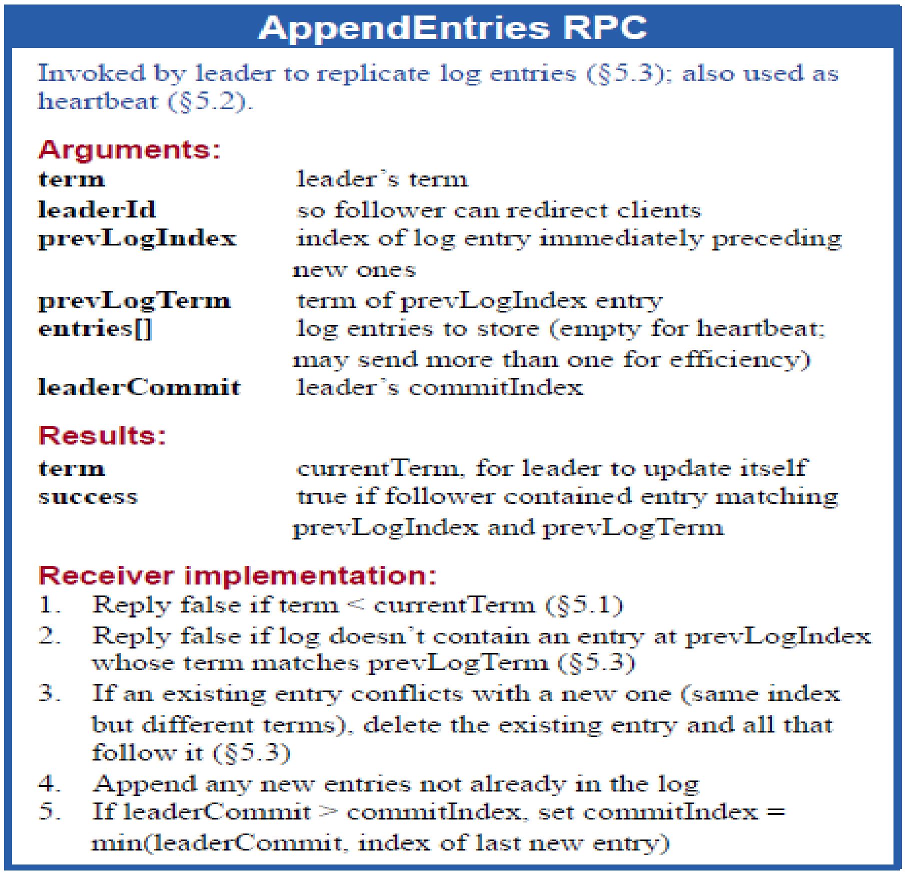
Fig : How an example message JSON would look like. First five fields are mandatory fields and the remaining fields are dependent on the type of **"request".** For instance, the above JSON is an example of an AppendEntryRPC

Append Reply (5 points)

When a follower receives an AppendEntry RPC (Note: heartbeats are also AppendEntryRPC), the follower can choose to accept or reject.
The follower can:

1. Reply false if *term<currentTerm* :
   - I.e if the follower's own term is greater than the term mentioned in the AppendEntryRPC, the follower rejects
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (or if there is no term at prevLogIndex) i.e. Log Consistency Check
3. If both the above conditions are not met, the the follower accepts the logs, appends any new entries to its own logs and sends an Append_reply with *success=True*

4. Also, update the follower node’s commitIndex based on the value sent by the leader and the length of the follower’s logs



AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

Leader Election (With log comparison to select leader) (5 points)

The leader election will be similar to what you did in phase 3, but this time the follower node will have additional rules to grant positive vote:

- Voting server will deny the vote if its own logs are more complete. I.e If:
    - $(\text{lastTerm}_V > \text{lastTerm}_C)$ OR $(\text{lastTerm}_V == \text{lastTerm}_C \ \&\& \ \text{lastIndex}_V > \text{lastIndex}_C)$

This ensures SAFETY i.e. once an entry has been committed by a Leader no other node that doesn’t have that entry gets to become the leader thereby preventing overwriting committed entries.

Report and Video Demo (5 points)

- A short video demo (Should be less than 8 mins and should be separately recorded for each teammate on their own laptop)
- A report following the structure similar to the previous phases

Note 4: The Report should contain a section discussing the difference between the basic log replication in phase 2 and the log replication you are implementing in this phase

Integrate Client UI from Phase 1 (Bonus : 5 points)

The client you might have implemented with an UI in phase 1, can be integrated with your RAFT cluster. This can be done by having your client send a STORE request to the leader, where the value of the STORE req can be the actual CRUD command that the client is making.

What to submit? (We will be deducting points if the naming convention for submission, the directory structure is not followed)

The submission must be a zip file with naming convention <ubid>_phase1.zip (eg : araman5_phase4.zip) uploaded to UBLearns. The zip file should contain the following (Read carefully):
The zipped file should contain:

- A folder containing the entire source code
- A short video demo
  - Please do not spend time explaining your code in the video demo
  - To save time, have your docker containers already running and the first leader already elected before you start recording
  - In your recording go over, you add entries to the logs, stop the leader, let a new leader be elected, add more entries to the logs, use RETRIEVE to get the logs, restart the old node which you had shutdown earlier and show how the new logs gets appended to the node which was just restart-ed
- A report following the structure similar to the previous phases (5 points for video + report)

Resource:
https://raft.github.io/
https://raft.github.io/raft.pdf
http://thesecretlivesofdata.com/raft/ (Highly recommended)
https://www.youtube.com/watch?v=YbZ3zDzDnrw
https://docs.python.org/3/library/socket.html
https://docs.python.org/3/library/threading.html