Raft

Raft is a consensus algorithm used for managing a replicated log. A primary challenge in the previous design structure is the lack of visibility into the actual state of the order server, such as database stocks and client responses. If the leader Order Server or the Frontend Server crashes and is restored, selecting a server with a higher ID for leadership could lead to issues.

For instance, consider a scenario with three Order Servers in a cluster, each maintaining a record of transactions. Server 3 is the leader responsible for handling all client requests. If Server 3 crashes and the system automatically elects Server 2 (which has the highest ID among the alive Order Servers) as the new leader based on its ID, problems may arise. If Server 2 lacks the latest committed entries from Server 3 due to synchronization lag or packet loss, it will start processing new client requests based on outdated information. Alternatively, the client may receive a successful response, but the new leader's database lacks the corresponding record. This situation could result in inconsistencies such as duplicate orders or incorrect stock levels, compromising the integrity of the database and potentially leading to customer dissatisfaction.

In a scenario where a leader in a Raft-based system crashes after committing an entry but before fully replicating it to all followers, the protocol includes mechanisms to maintain system consistency and prevent the loss of committed entries. In Raft, only the leader handles requests that result in state changes, such as POST, PUT, and DELETE. Upon receiving a request, the leader stores it in its in-memory logs (but does not commit it to the database yet) and sends an AppendEntries RPC to its followers to verify if this log matches their log history. If it does, the log is also stored in the in-memory logs on their servers. The leader collects responses from all peers, and only if a majority of the servers accept this log, can the leader commit the request and return a successful response to the client.

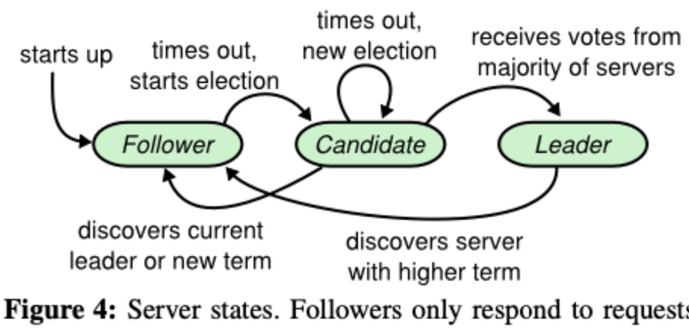
starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

Leaders send regular heartbeats (AppendEntries RPC with empty entries) to assert their leadership status. If a leader crashes, followers cease receiving heartbeats, prompting them to become candidates and initiate an election by sending RequestVote RPCs at random intervals (to prevent simultaneous candidacy and self-voting by all followers). Only the candidate with the most recent log index and term can win the election. After winning the election (receiving successful RequestVote RPC responses from the majority of peers) and becoming the leader, it resumes sending heartbeats, causing other candidates to revert to followers. In the previous example discussed, where the former leader committed a log and returned a successful response to the client but did not inform others about the commitment, Raft addresses this through its leader election mechanism. Consequently, only the server with this log can be the next leader. Since the log exists in the majority of peers' in-memory logs, the new leader will still commit this log (stored in the database) to ensure it is not missed.

For a detailed understanding of Raft, including its design and proof, please refer to the paper cited in the references. Here, our focus is on implementing Raft within our previous structure.

We use `USE_RAFT=True` when starting the Order Server and the Frontend Server to operate in Raft mode, distinct from the previous operation mode.

Front-end Service

In Raft, the Frontend Server does not determine which Order Server is the leader, as it lacks visibility into which Order Server holds the accurate records. Selecting the server with the highest ID does not ensure consistent records between the database and the client, as discussed previously. In Raft, each Order Server autonomously selects its leader without relying on other servers. Therefore, the Frontend Server can send requests to any order server, mirroring real-world scenarios. If the server receiving the state-changing request is not the leader, it redirects the request to the current leader.

Order Service

Most of the design is referenced in Figure 2 of the Raft paper.

Raft Class

We define the Raft class in `src/order/app/utils/raft.py`, which contains all the attributes and methods needed to implement the Raft design. When an Order Server starts, it instantiates a Raft instance in `src/order/order/wsgi.py`.

Views

We add two endpoints to handle the AppendEntries RPC and RequestVote RPC and modify the original post order process under Raft mode.

1. `handle_vote(request)`
   This function processes vote requests from other servers (candidates) in the Raft cluster. When a server wants to become the leader, it sends a RequestVote RPC containing its term, candidate ID, and log entries information. The function checks if the requesting candidate's term and log are up-to-date compared to its own. If the candidate's log is at least as recent and the current server hasn't voted for another server in the current term, it grants its vote to the candidate. The function responds with whether the vote was granted and the current term.
2. `handle_append_entries(request)`
   This function handles AppendEntries RPCs sent by the leader. These requests are used for log replication and as heartbeats. The function first checks if the term in the request is at least as recent as the term of the server; if not, the server rejects the request. If the log entry indices match, it appends any new entries provided in the request to its log. If the request is a heartbeat (contains no new entries), it updates its internal state to reflect that it has heard from the leader. Additionally, this function updates the server's commit index if the leader's commit index is ahead, applying any newly committed entries to the state machine, which involves creating orders in the database as specified by the log entries.

We modified the original `process_post_order_request()` function in the Raft mode to perform the following steps:

1. Check if the current server is the leader; only the leader can accept the request.
2. Use order_data to create an order object but don't save it to the database at this point.
3. Append the order object, term, and command to the log.
4. Send AppendEntries RPC to all other servers.
5. Check if successful replies constitute a majority or not.
6. If a majority, save the order and log object to the database (committed) and send a success response to the client.
7. If not a majority, send an error response to the client.
8. Update commitIndex and lastApplied and send AppendEntries RPC to all other servers with the updated commitIndex, which can be done by heartbeats.

Follower State

Upon instantiation, the Raft server will be a Follower first.

Ticker (Follower and Candidate): The server starts the ticker method in a thread, which is called periodically to check if the server has timed out and needs to start a new election.

Start a Leader Election: The follower will accept a heartbeat from the leader in `append_entries/` with POST method. If the entries are empty, it will record its `self.lastHeartbeatTime` in the Raft instance. The ticker will periodically check this field to decide whether it should start a leader election. If yes, it will execute `self.start_election()` and become a candidate.

Candidate State

Candidates send RequestVote RPCs containing their information to other peers. For efficiency, these requests are sent using multiple threads, and `thread.join()` is used to wait until all threads have finished. However, it's not necessary to collect all responses from the previous step; a candidate only needs a majority of votes to become the leader. Therefore, we use a `votesReceived` counter to track this. Additionally, if a response contains a higher term, it causes the candidate to revert to follower status, indicating that another server has a more up-to-date state, and the candidate cannot become the leader.

Leader State

Send Heartbeats: The leader uses `send_heart_beats()` to instruct both followers and candidates to revert to follower status and update their `lastHeartbeatTime` attribute.

Handle Requests: The leader handles requests from the Frontend Server. For the GET order request, we use the original one because this request doesn't affect the state, so Raft doesn't store it as a log. For POST order requests, the leader uses the modified `process_post_order_request` to handle it as mentioned above. The leader also uses multi-threads to send AppendEntries RPC and uses `thread.join()` to block the process until the leader receives all replies or gets a majority granted reply.

Revert to Follower: A leader may revert to follower status if it encounters another node with a higher term. This can happen due to network partitions that resolve, delayed starts of some nodes leading to unexpected leader elections, or network delays causing outdated but higher-term messages to be received. When a leader receives a request with a higher term, it updates its own term, surrenders leadership, and becomes a follower, ensuring the cluster remains consistent and avoids conflicting leadership.

Middleware

We add middleware to handle redirect requests to the leader. When starting the Order Server with `USE_RAFT=True`, only the leader server can accept POST order requests or GET order requests. If followers receive such requests, they will redirect them to the leader.

Model

In the Raft implementation, persistent states such as `currentTerm`, `votedFor`, and `log[]` need to be stored on all servers. To achieve this, we added a `RaftServer` model to store `current_term` and `vote_for` fields. Whenever these fields are updated, the changes are reflected in the database. Additionally, when a Raft server is instantiated, it retrieves this state from the database.

For storing logs, we use the LogEntry model, which includes fields such as index, term, order (a ForeignKey to the Order model), and command, to store information about committed logs.

Citations

1. Lab 2
2. GPT
3. https://stackoverflow.com/questions/25369068/python-how-to-unit-test-a-custom-http-request-handler
4. https://www.djangoproject.com/
5. http://nil.csail.mit.edu/6.824/2020/schedule.html, MIT 6.824 Lab 2,3
6. Raft Paper: Raft (extended) (2014), Section 7 to end (but not Section 6)