Middleware for PostgreSQL Database Replication, Failure and Recovery

Hari Kishore Chaparala
Dept of Computer Science, UCI
hchapara@uci.edu

Sai Vineeth Doddala
Dept of Computer Science, UCI
sdoddala@uci.edu

Manikanta Loya
Dept of Computer Science, UCI
manikanl@uci.edu

***Abstract*—In this project we have designed and built a distributed transaction processing system with replicated databases for a banking application. The transactions in different databases are processed through agents. We used a Single Master design with one agent acting as a leader and others as replicas. We have also created a client application that the users can use to communicate with the distributed database system. Leader in this system supports processing of reads and writes where as replicas only supports read-only transactions. Our system also supports leader replacement/ failover in case the current leader crashes and also a recovery process that synchronizes the recovered node with up to date transactions and data state. Our system supports any number of clients and agents (attached to databases) Project demo link for your reference: [1]. Our code is available at [2].**

I. Setup

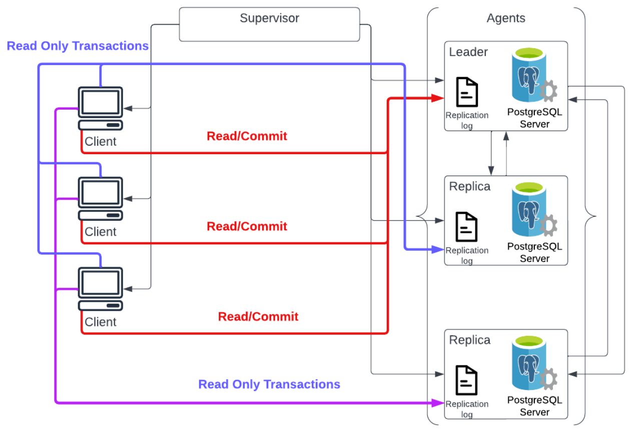

Fig. 1. Architecture of the system

The project setup requires Python version $\geq$ 3.9 and libraries psycopg2, Flask and requests. Docker needs to installed. Postgres doesn't need to be installed. We use the Dockerfile to create an image with Postgres installed. We expect our project to be setup using the provided setup shell script in Linux and macOS.

Assumptions:

- This system gives more priority to correctness of the database when compared to throughput and efficiency.
- Message failures will not occur in this system
- Current replica agents won't be crashed. A replica that becomes a leader can still be crashed.

II. Prototype

A. Client

Client is a web-server using Flask, which receives transaction sequence as input. Transaction sequence consists of read(attribute) and write operation(attribute : value). E.g. $r(1)w(2 = 300)r(3)w(4 = 600)$. We extract these operations and save them as set and dictionary data structures respectively. Client then buffers writes and send read request API call to leader. Upon processing the request if leader is able to process the request, it returns values of attributes in read set and timestamp of the system at which read was done.

Upon on reading the attributes, client sends commit request with both attributes and read timestamp of the transaction. The agent upon on processing the commit request return either "COMMIT" or "ABORT" message depending on its concurrency control mechanism (We used timestamp ordering and write set locking).

If the request contains a write attribute that is not already present in the database, we insert it, otherwise we modify the data.

The client uses Optimistic Concurrency control at it's end for concurrency control. Each transaction is associated with a transaction id that is always incrementing. We used python's **time_ns** for getting a unique transaction id. At the beginning of the read phase, we assign the id and use it throught the lifecycle of the transaction at the client. In the read phase, the client queries the current leader with it's read set. Next, in the validation phase, we get all validation transactions V and remove transactions that have been finished F before the current transaction read phase. The validation part is synchronized. If the current transaction read set overlaps with any of the $V - F$ write set, we abort the transaction. Otherwise, we again filter all the finished transactions from $V - F$. Let this be J. Now if the current transaction write set overlaps with any of the J's write set, we abort the transactions. Else, we add the current transaction to V, and send it to the leader agent. If there is a positive response, we add it to G, a set of finished transactions.

B. Database

We created a docker image using a Dockerfile. This image contains PostgreSQL installed and we use this image to create required number of database docker containers attached to different host ports which can be used for connection through the agents. Our database scheme consists of a single table which is created if doesn't already exists by the agent during its start up. We have a simple table **bank_balance** that contains columns **account_number, balance, created_at and updated_at** timestamps. All are integers. Our application only supports transaction processing using **INSERT, UPDATE and SELECT** statements.

C. Agent

For this project we have 3 agents with replicated databases. There can be any number of agents. See the **README.md** for the setup steps. One of the three agents acts as a leader and client communicates with it for updating the database. Agent can serve two types of requests, READ and COMMIT. We also expose other APIs in the agent for HTTP communication between agents and Supervisor. An Agent can receive requests from multiple clients.

On receiving either READ or COMMIT request, leader first checks the availability of resources i.e. is there is a current process in commit state trying to update the resource. If the resources are not locked by any other transactions and its a READ request then the client returns the values of the requested attributes from the database along with the current timestamp of the agent(read_timestamp). And if any of the resources are locked i.e., being used by other transaction, read request is unsuccessful and transaction aborts. This avoids write-write/read-write conflicts between transactions.

For COMMIT requests, agent needs to validates these transactions because there can be multiple clients and this can be lead to stale reads. And committing transactions with stale reads will lead to conflicts(read-write). To avoid this implemented Timestamp Ordering(TO) to resolve any read-write conflicts between transactions of different clients. Client sends read_timestamp along with transaction details in its COMMIT request. Agent then uses this read_timestamp to check if any attributes in read set have been modified by any other transaction since it read the value. If it has any conflicts agent aborts the transaction. There by ensuring conflict serializability between transactions of different clients.

Agent then locks up the resources needed by agent for committing the transaction(PREPARED state). Leader then communicates with other agents using 2PC communication protocol(II-D). After successfully completing the request and committing the data, it returns "COMMIT" response to client. Else if it fails in 2PC protocol it returns "ABORT" as response.

D. 2 Phase Commit

The transaction enters PREPARED state, Leader node communicates with other replicas/agents using 2PC communication protocol to replicate the leader actions. It sends prepare request to all other agents. The other agents that receive prepare request checks the availability and locks up the resources needed for committing the transaction if resources are available. If the resources are available, transaction moves into PREPARED state and replies "YES" message back to leader. If any one node fails to allocate the resources then replies "NO" to the leader which aborts the transaction with an abort message to the agents.

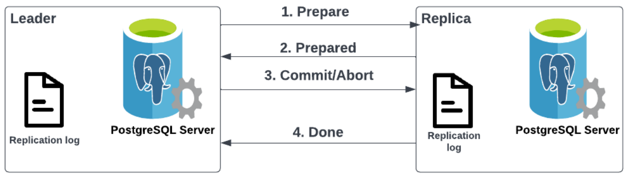

Fig. 2. Log replication using 2PC

Upon receiving "YES" from other agents in case everyone is prepared, Leader sends commit request to other agents. Other agents that receive commit request updates their databases, unlocks the resources and returns "YES" message back to leader. As soon as leader receives "YES" response from all other agents, it updates its database of the attributes with values in write_set.

E. Logging

We primarily use logging for recovery. We log before each state change of the transaction (WAL). For a transaction T, if the leader is able to allocate resources to process commit request it logs the event with {id}$$START$$. After locking the resources it logs {id}$$PREPARED$$ and upon receiving prepared response it logs {id}$$COMMIT$$. After receiving commit acknowledge from other agents and updating its database it logs {id}$$COMPLETED$$. The same logs aare replicated in the agents as well. The default recovery file name is **recover_log.txt**. A sample transaction: **1654830467652342000$$START$${'read_set' [1, 2], 'write_set' {'90': 10, '4000' 30} 'time_stamp' 1654830467645209000, 'transaction_id' 1654830467652342000}**

F. Supervisor

Supervisor is the main service in this system which handles multiple agents and the clients and its primary goal is to assist in the leader election during the failover and recovery of the failed node. It periodically polls and checks whether the leader is up or not. If the leader is down, it start the failover process as described in the failover section. During the recovery, supervisor gets the replication log from the leader and forwards it to the recovering node for it to sync with the current leader. It also informs the clients during about any leader and node updated for performing transactions. The supervisor is also responsible for sending HTTP requests to the agents for halting any ongoing transactions. This is required for consistency in our system.

III. Read Only Transactions

We designed our system such that all the agents have the ability to process read-only transactions. Whenever any client requests read-only transaction, agent first checks for the availability of the resources requested. If they are currently used/locked by another transaction, the read-only request is denied and an abort is sent. Note that we don't keep log read only transactions in the recovery log as it is only meant for data consistency across agents. The transactions processed by the leader are never aborted due to a conflicting read-only transaction in the replica. In order to have this property, we can't rely on just the locks. So we used a modified version of time stamp ordering. Let TR be a read-only transaction on an agent. We note it begin time as $T(TR)$. Then we read the values of the read-set of TR. We only process the read $x \in readset$ if the $T(TR)$ is greater than the highest timestamp of any write on x. In order to prove that this is conflict serializable, let us consider a read-set over values a, b, c. Now, reads over these values are only permitted if the transaction timestamp is greater than any write on respective values. This always ensures that the read is performed with on the latest set of values when no other transaction modified them. We can also prove this using contradiction with the SG (Serialization graph) like we did in the class. If there is a loop in the SG between $T1$ and Tn, it means that the time stamp of the $T1$ should be greater than the timestamp of the last transaction Tn because in time stamp ordering if $T2$ is followed by $T1$, $T1$'s timestamp is smaller than $T2$'s. With induction, we can say the $T1$ time stamp is smaller then Tn. But this is a contradiction.

IV. Failover Mechanism

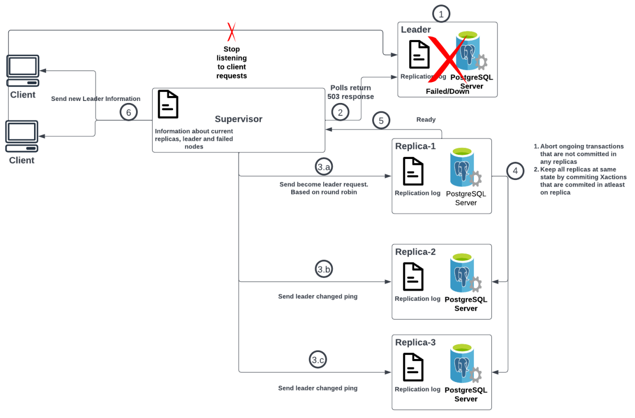
1
Leader
Replication log
PostgreSQL Server
Failed/Down
Stop listening to client requests
Client
Client
Send new Leader Information
6
Supervisor
Information about current replicas, leader and failed nodes
Polls return 503 response
2
5
Ready
Replica-1
Replication log
PostgreSQL Server
3.a
Send become leader request. Based on round robin
4
1. Abort ongoing transactions that are not committed in any replicas
2. Keep all replicas at same state by commiting Xactions that are commited in atleast on replica
Replica-2
Replication log
PostgreSQL Server
3.b
Send leader changed ping
Replica-3
Replication log
PostgreSQL Server
3.c
Send leader changed ping

Fig. 3. Leader failure and replacement

A. Leader Election

Once the leader goes down, supervisor in it's next round of polling (currently every 5 seconds) finds that the leader is down. In our design, we are electing the new leader in a round-robin manner. Supervisor informs the new leader and all other participants about the leader change. The supervisor also requests the agents through an API to stop processing any further transactions. READ-ONLY transactions are still permitted.

B. Synchronizing Non-Leader Agents

If leader goes done while processing a commit request, other agents must handle processing of these transactions before accepting any new requests. So any process which is in PREPARED state is aborted, we unlock the resources and ABORT log is written to the disk. For any other transactions in COMMITTED state, the new leader identifies the transactions that are committed in at-least on replica and ensures their completion in all nodes.

C. Client Notification

When the leader fails, once the leader election and replacement is completed, the supervisor sends an API call to all the clients informing about the new leader. Clients update the leader information in their system and starts sending the transactions to the new leader.

V. Recovery

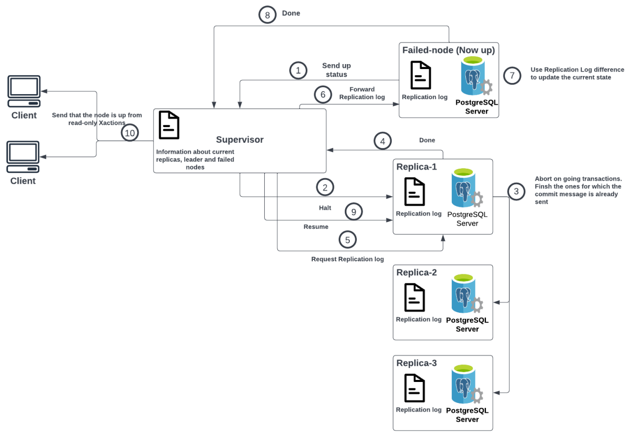

Fig. 4. Recovery of crashed agent

When the crashed agent(old leader) comes up, it informs the supervisor through an API call. Since the crashed agent might have missed few transactions when it was down, it requests the replication log of the current leader to recover through the supervisor. When an agent is recovering, we have made a design choice to stall the transactions in the current leader to maintain consistency. As other agents perform read-only transactions, they can continue serving their clients to improve the system efficiency. In the meantime, supervisor gets the replication log from the current leader and forwards it to the recovering agent. Once the agent recovery is done i.e it has updated its recovery log and its database, we resume the transactions in the current leader and the recovered node gets attached as a replica to the current leader.

A. Client Notification

Once the agent recovers, the supervisor informs all the clients about the recovered agent so that clients can send read-only transactions to the recovered agent.

References

[1] Project demo: https://uci.zoom.us/rec/play/XfYJdQV2mEUZMjVbGv0JtU6FDDWVYSBOgj-5NaPcyq53nPnh5l9dXuZib9WYSj0GHtVf22GWpIA3zzWn.8W8pCxp1H-Vg_A0y?continueModAMQSeE3-43xLYEQw.1654834209468.5781cf59baa7d81fe9d3aedfb64de68f&_x_zm_rhtaid=319

[2] Code link: https://github.com/manikanta-72/Transactions-Course-Project