University of California, Irvine
CS 223 Transaction Processing and Distributed Data Management

Implementation of Data Replication of Transactions on Top of PosrgreSQL

Wenbo Li, Shiran Wang, Xinyi Tang

June 2022

Contents

1 Introduction

In this project, we implement data replication of transactions on top of PostgreSQL. We set up 3 servers with corresponding agents on PostgreSQL for our basic setup. The basic mechanism of the system is that, we will choose a server to be the leader, and the remaining two servers will be the participants/follower. Then a client connect to the leader to send transactions, it processes transactions using optimistic concurrency control protocol. It sends read operations to the leader and buffers write operations. When the transaction completes processing, the client will then send a COMMIT request to the leader to try to commit all changes. For the leader, a READ request is to return the most recently committed value of the requested data item. A COMMIT request is to check whether the transaction can or cannot commit. If not, the leader will send ABORT message and rollback the transactions. Otherwise, it sends COMMIT to all replicated databases and apply all writes to the database, responding to the user with a COMMIT, and appending the transaction's writes to the Replication Log.

Specifically, when a transaction is committed successfully, the leader sends the corresponding entry to the Replication Log of all the followers, and the followers will apply the writes to their databases.

More specifically, we improved the above mechanism by adding two more functions: to enable read-only transactions at any node, and to enable leader replacement in case of leader failure. In the following sections, we will describe the algorithm we used to realize these functions, and provide a test result to validate them.

2 Basic Setups

2.1 Servers

The setup for the 3 servers environment can be seen in the README file. The leader is initially assigned by us, and will remain unchanged if there are no failures. We will see how it executes leader replacement in later parts.

2.2 The Client

The setups for the client are mainly written the "client.java" file. when we first start the client, it need to make a connection to the leader to send transactions. This is realized using pointers in the "main.java" file. After we start the 3 servers, we choose the leader and link it to the client (see last 2 lines in Figure 1).

```
public static void main(String args[]) {
    System.out.println("\nStarting experiment.\n");
    Utils util = new Utils();
    Agent[] agents = new Agent[3];
    Client client = new Client(agents);
    int[] ports = new int[] { 5500, 5501, 5502 };
    for (int i = 0; i < 3; i++)
        agents[i] = new Agent("Wenbo Li", "postgres", ports[i]);
    System.out.println("Databases connected.\n");
    // clear and initiate DBs
    util.clearAllAndAddTestValues(agents);
    // set leader and make connections
    client.leader = agents[0];
    client.leader.agents = agents;
```

Figure 1: Initialize database and link the client to the leader before sending transactions.

Then, we make the client processes transactions using **Timestamp Ordering (TO)**. In the beginning of the sub-function "public boolean processesTransactions()" in "client.java", as is shown in Figure 2, we implemented TO using "Ts", which is a queue that stores the timestamp of each transaction. Queues follow first-in-first-out rule, so we can always return the oldest transaction, so that we can follow the timestamp ordering rules. Therefore, we first clears the read and write sets, then ensuring serializability using timestamps.

```
public boolean processesTransactions() {
    readSetID.clear();
    readSetValue.clear();
    writeSetID.clear();
    writeSetValue.clear();
    // We use TO protocal to schedual transactions. i.e. using a queue
    String nextT = Ts.peek();
    String[] temp = nextT.split(";");
```

Figure 2: Timestamp ordering used to schedule transactions in client.

Here, the "clearAllAndAddTestValues" is implemented in "utils.java", which is used to create and implement new tables (and snapshot) for the database. If the database is successfully setup, it will print "Databases are ready for experiment". This file also contains a "closeConnection" function to close all the connections with DBs. This is not the main concern of our project, you may refer to source code for details.

Notice that the client sends read operations to the leader, but it buffers write operations. Therefore, it only fetched the values needed from leader, and all values will be first stored in its read/write sets. In the following Figure 3 (a), we can see that the client reads the transactions, and do READ, ADD, MULTIPLY and SET(WRITE), and then store results into read/write set. In Figure 3 (b), we show the implementation of read and write specifically. You can see that the read/write values will be pushed to the read/write set respectively for commit later. The implementation of ADD and MULTIPLY can be found in "client.java". They directly read and change an item's value, and can increase the concurrency as described in class.

```
// transaction starts processing
for (String t : temp) {
    String[] curr = t.split("[(,+)]");
    // read
    switch (curr[0]) {
        case "read":
            read(curr[1], curr[2]);
            break;
        case "add":
            updateAdd(curr[1], curr[2], curr[3]);
            break;
        case "set":
            set(curr[1], curr[2]);
            break;
        case "muti":
            mutiply(curr[1], curr[2], curr[3]);
            break;
    }
}
```

(a) Client Process

```
public void read(String idd, String saveTo) {
    int id = Integer.parseInt(idd);
    int curr = leader.handleRead(id);
    ht.put(saveTo, curr);//a -> 10
    // add to read set
    readSetID.add(id);
    readSetValue.add(curr);
}

public void set(String idd, String value) {
    // add to wite set
    writeSetID.add(Integer.parseInt(idd));
    try {
        writeSetValue.add(Integer.parseInt(value));
    } catch (NumberFormatException e) {
        writeSetValue.add(ht.get(value));
    }
}
```

(b) Client Read and Set

Figure 3: Client processing the transactions (left), and Read/Set(Write) implementation (right)

As soon as the transaction is completed, the client will send a COMMIT request that includes both the read and write sets to leader. If this is done successfully (no crashes), the following code in Figure 4 will be applied:

```
try {
    // transaction completes processing, send it to leader for 2pc.
    leader.receiveReplicationLog(readSetID, readSetValue, writeSetID,
            writeSetValue);
    boolean rslt = leader.handleCOMMITRequest();//true COMMIT; false ABORT
    Ts.poll();
    return rslt;
```

Figure 4: Code when the client can successfully send the read/write sets with COMMIT.

Note that we assume the client will not face crashes during the processes. However, it is possible that the database crashes. If the leader crashes while processing the message, we will use the "Leader Replacement" algorithm to handle this. More details will be covered in Section 4.

2.3 The Leader and Followers

Under this section, we will focus on how the leader handles the COMMIT request, and then we want to construct 2PC for the leader and followers.

First of all, the leader sends the corresponding entry in the replication log to the three agents and let them to receive it. Then we will check if the value of the readset is the same as the value of the previous writeset like Figure 5 have done. To avoid the first transaction get into this loop, we will check if the writeset is empty or not; then, we use the hash table(query time = $O(1)$) called prevWriteSet to record the writeset of the current transaction for the futures transactions to check if any conflicts of concurrency. **Notice that to pass conflicts check, the values in the readset of the current transaction should be identical to the writeset of the previous transaction.**

```
// send Log to agent
for (Agent agent : agents) {
    if (agent != this)
        agent.receiveReplicationLog(ReplicationLogReadID, ReplicationLogReadValue, ReplicationLogWriteID,
                ReplicationLogWriteValue);
}
// check if readSet == prevWriteSet
if (!prevWriteSet.isEmpty()) {
    while (!ReplicationLogReadID.isEmpty()) {
        int currValue = ReplicationLogReadValue.poll();
        Integer prevValue = prevWriteSet.get(ReplicationLogReadID.poll());
        if (prevValue != null && prevValue != currValue)
            return false;
    }
}
```

Figure 5: Code for handling the commit request in Agent.java.

After finishing checking the conflicts, we will clear the previous writeset and begin voting. If all voting yes, we then implement 2PC like Figure 6 on the following page shown below. If the replication log of the leader is not empty, we copy the writeset from the previous writeset. For each agent, 2PC lets the clients to vote

here, and if one agent has been crashed before it commits, we will break the loop and do ABORT; otherwise, we will do the redo log until we go over all the agents. We also poll to the agent's replication log after success. Last not but not the least, we check if anyone has committed or not: if there is no commit, we do rollback over each agent and clear the previous writeset; otherwise, this means all agents have prepared, and we can commit all to the database with a COMMIT decision back to the client. Here, the 2PC process is done.

```
// if passed, then 2pc is implemented here
boolean COMMIT = true;
while (!ReplicationLogWriteID.isEmpty()) {
    int id = ReplicationLogWriteID.poll();
    int rslt = ReplicationLogWriteValue.poll();
    // copy writeSet to prevWriteSet
    prevWriteSet.put(id, rslt);
    for (Agent agent : agents) {
        // 2pc to let clients vote
        if (!agent.setValue(id, rslt)) {
            COMMIT = false;
            break;
        }

        //simulate leader failure
        if (simulateFailure)
            System.out.println(1/0);

        // log after success
        agent.ReplicationLogWriteID.poll();
        agent.ReplicationLogWriteValue.poll();
    }
    if (!COMMIT)
        break;
}
```

Figure 6: Code for the basic part of the 2PC implementation in Agent.java.

2.4 Test Results

For the basic setup, we will use the following inputs to see if the whole database can run without errors under normal condition (Figure 7 on the next page). We input 3 transactions, here "read(1,a)" means read value in position 1 for a, "add(a,a,10)" means "a=a+10", and "set(1, a+10)" means write value a+10 to position 1.

```
public Client(Agent[] agents) {
    this.agents = agents;
    readSetID = new LinkedList<Integer>();
    readSetValue = new LinkedList<Integer>();
    writeSetID = new LinkedList<Integer>();
    writeSetValue = new LinkedList<Integer>();
    Ts = new LinkedList<String>();
    Ts.add("read(1,a);" +
            "add(a,a,10);" +
            "set(1,a+10)");

    Ts.add("read(1,a);" +
            "read(2,b);" +
            "add(c,a,b);" +
            "set(2,c)");

    Ts.add("read(1,a);" +
            "read(2,b);" +
            "add(c,a,b);" +
            "set(3,c)");
}
```

Figure 7: The input for testing the basic setups.

The output is shown in Figure 8, and the database result is also shown. The initial value of the items are 100, 200, 300, 400. The result values, 110, 310, 420, 400 are exactly what we have expected, and it indicates that our basic setups are successful.

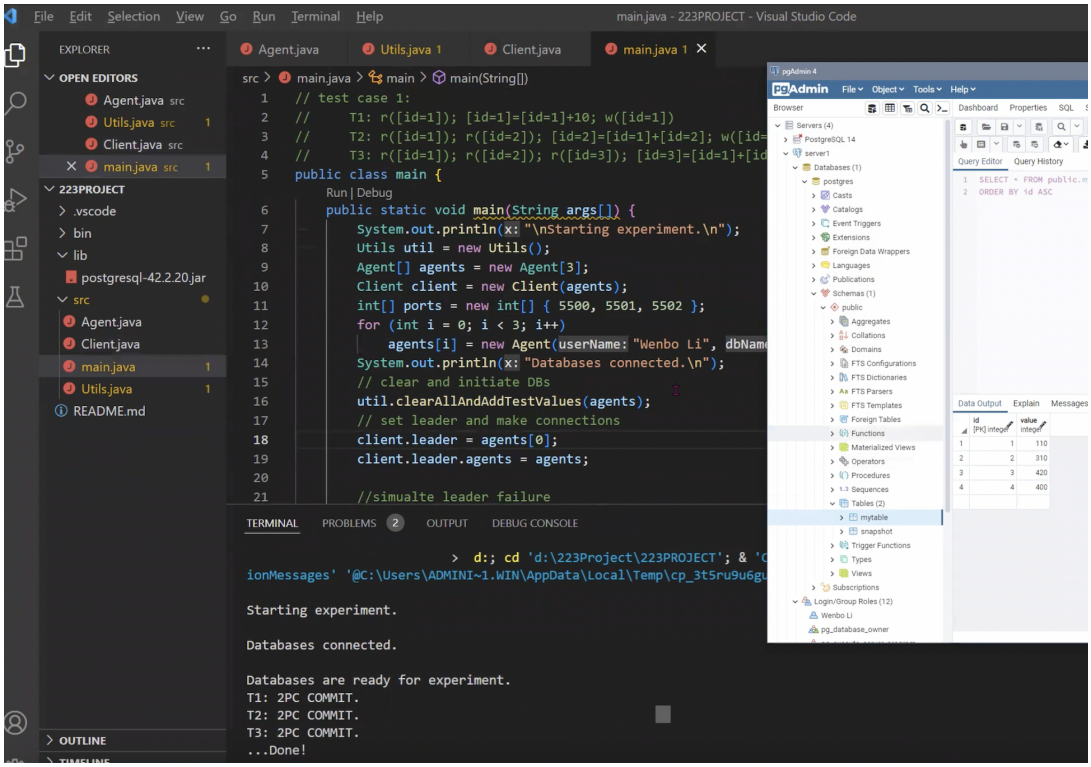

Figure 8: Successful test result.

There is possibility that the 2PC protocol, some participants will vote "abort". We also simulated this situation, by uncommenting the following comment in "agent.java" line 230-231 (Figure 9). Besides, by changing the "id" value in the if statement, we can simulate aborting any of the 3 transactions. The next Figure 10 shows the test result if we abort T3, which indicates our 2PC implementation is correct.

```
public boolean setValue(int id, int newValue) {
    // //simulate node voting ABORT(false)
    // if (id==1)
    // return false;
```

Figure 9: Extra code for testing abort-rollback situation.

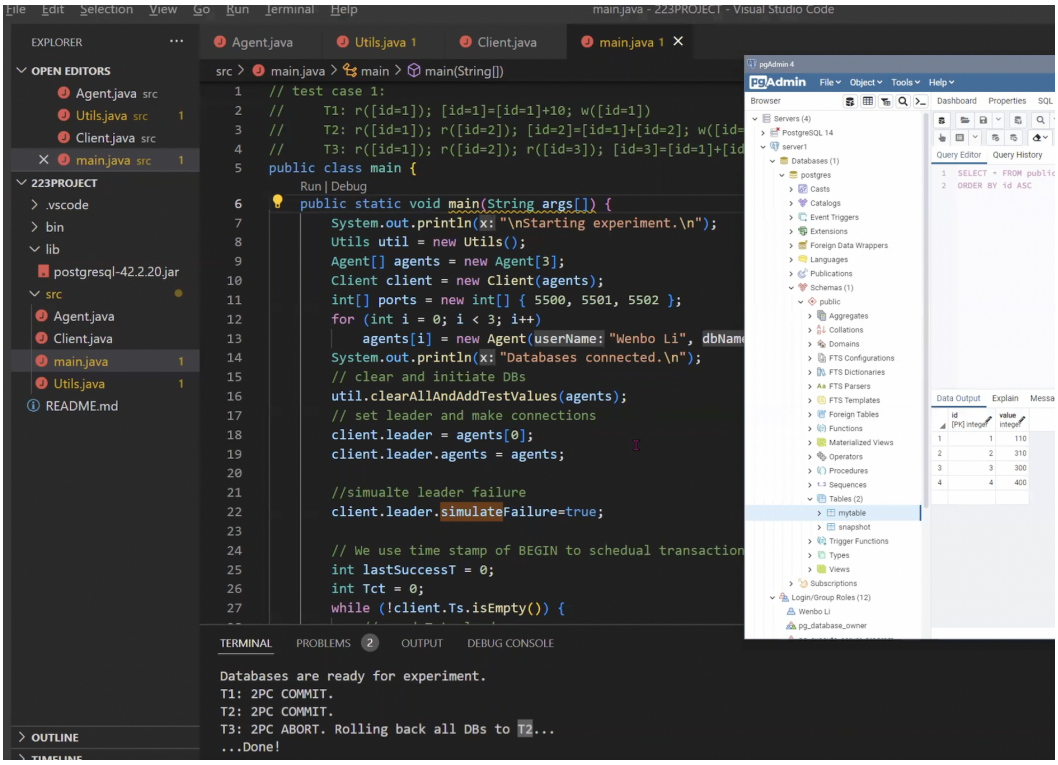

Figure 10: Test result when T3 is aborted.

3 Read-only Transactions

For the basic setup, we only let client read from the leader node. In this part, we extend the ability such that agent at all nodes can perform a read-only transaction while maintaining conflict serializability.

3.1 Algorithm Description

The main concept we applied here is using another table called **snapshot** in all DBs. Originally, the client send READ request to the leader and leader will find from the database and return a result. Now, by taking a snapshot of the whole database after each successfully committed transaction if there is a write update, we will always hold a snapshot for the latest database content after last successful update. When client wants to perform a read-only transaction, it will directly reach the snapshot of any agent of node it wants (even

the followers), and run concurrently with other transactions. To achieve this, we implemented the following function ("Agent.java"), see Figure 11:

```
public Integer handleRead(int id) {
    Statement stmt = null;
    try {
        stmt = c.createStatement();
        //read from snapshot
        ResultSet rs = stmt.executeQuery("SELECT * FROM SNAPSHOT;");
        while (rs.next()) {
            if (rs.getInt("id") == id) {
                int rslt = rs.getInt("value");
                // System.out.println(id+", "+rslt);
                rs.close();
                stmt.close();
                return rslt;
            }
        }
        rs.close();
        stmt.close();
    } catch (Exception e) {
        System.err.println(e.getClass().getName() + ": " + e.getMessage());
    }
    // System.out.println("Print done.\n");
    return null;
}
```

Figure 11: Function for read-only transactions. We can see that it reads from the snapshot.

The snapshots are taken using the following code shown in Figure 12 ("Agent.java"):

```
public boolean setValue(int id, int newValue) {
    // //simulate node voting ABORT(false)
    // if (id==1)
    // return false;
    Statement stmt = null;
    try {
        stmt = c.createStatement();
        String sql = "UPDATE MYTABLE set VALUE = " + newValue + " where ID=" + id + ";";
        sql += "UPDATE SNAPSHOT set VALUE = " + newValue + " where ID=" + id + ";";
        stmt.executeUpdate(sql);
```

Figure 12: Snapshot implementation. Each time the database is updated, so does the snapshot.

3.2 Proof of Conflict Serializability

Conflict serializability is obvious. Read-only transactions will only access the snapshot (the copy) on each agent, not the original database. We update the snapshot only after each update is successfully committed. Therefore, the read-only transactions will not conflict other normal transactions at all.

3.3 Test Result

Test for this function is trivial. Previous success test results in Section 2.4 show that the whole system can function without error with this function extension.

4 Leader Replacement

4.1 Algorithm Description

If we face leader failure when we get no response from the leader, we consider executing the leader replacement here. There are several reasons which might cause the leader failure: the leader fails before the sending the prepared message; It fails under the read request; It fails before voting, etc.. So for the above situations, we are going to change the leader and preform UNDO. One last special occasion, when one node finished commit and the leader accept, but then the leader fails before the second node commits. Here we **MUST** set the new leader and preform REDO.

Based on the above algorithm, let us take a look into the details. If the leader has crashed and any database has committed, we will implement REDO log. We set the new leader and let the new leader know who the current agents are. Then, we preform the REDO.

```
for (Agent agent : agents) {
    if (agent.commited) {
        // set new leader
        leader = agent;
        leader.agents = agents;
        // redo all
        System.out.println("Performing REDO.");
        leader.REDO();
        Ts.poll();
        return true;
    }
}
```

Figure 13: Leader Replacement when any database has committed.

If nothing has been committed, we will perform UNDO. By going over each agent, we find if any agent that has not been the leader, set this agent as the new leader, and break the loop. Then, we perform UNDO, and rollback all.

```
// if none DB committed, ABORT all, restart transaction
// set new leader
for (Agent agent : agents) {
    if (agent != leader) {
        leader = agent;
        leader.agents = agents;
        break;
    }
}
System.out.println("Performing UNDO.");
leader.rollBackAll();
```

Figure 14: Leader Replacement when none database has committed.

4.2 Test Result

We simulated the following 2 scenarios: the first possible situation is that the leader fails after client has sent COMMIT; the second is that the client crashed before receiving the COMMIT from client. To test the result, we add the following command, and add the following command shown in Figire 16 on the next page to the place where the leader might fail. We also has a line to enable this leader failure function, in “main.java”, change the condition to “true” means that we allow leader failure, see Figure 15 on the following page.

```
//simulate leader failure
if (simulateFailure)
    System.out.println(1/0);
```

Figure 15: Test code to trigger a leader crash.

```
//simualte leader failure
client.leader.simulateFailure=true;
```

Figure 16: Test code to plan a leader crash in the future.

By placing the code into "agent.java" line 83-86, we will fail the leader during the commit. The test result is shown in Figure 17. You can see that when the leader failed during commit, it will roll back and new leader will REDO all transactions.

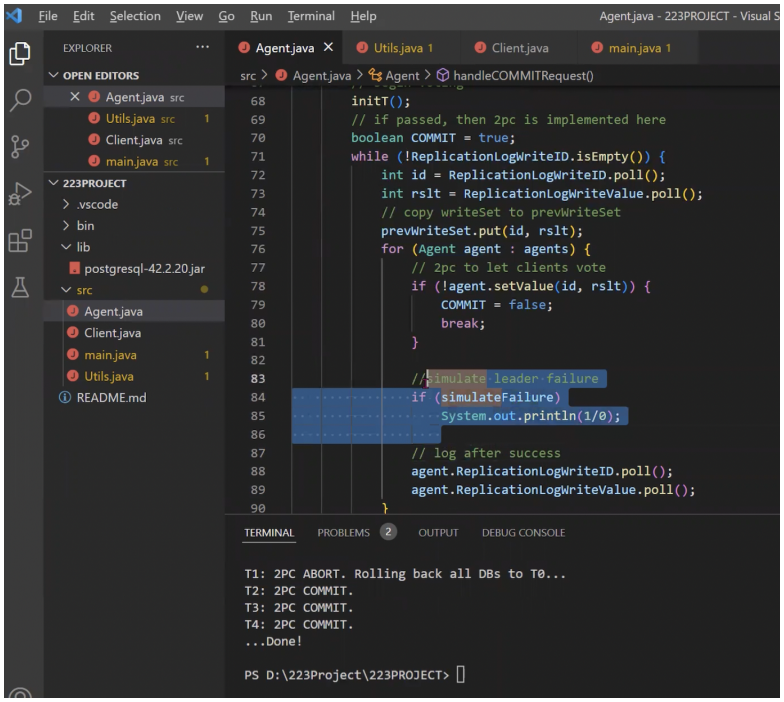

Figure 17: Leader-replacement situation 1: leader fails during commit.

By placing the code into "agent.java" line at the beggining of function " handleCOMMITRequest" , we will fail the leader at the begining. The test result is shown in Figure 18 on the following page. You will see that the leader did not respond to the COMMIT from client, so the system will choose a new leader and UNDO the 3 transactions.

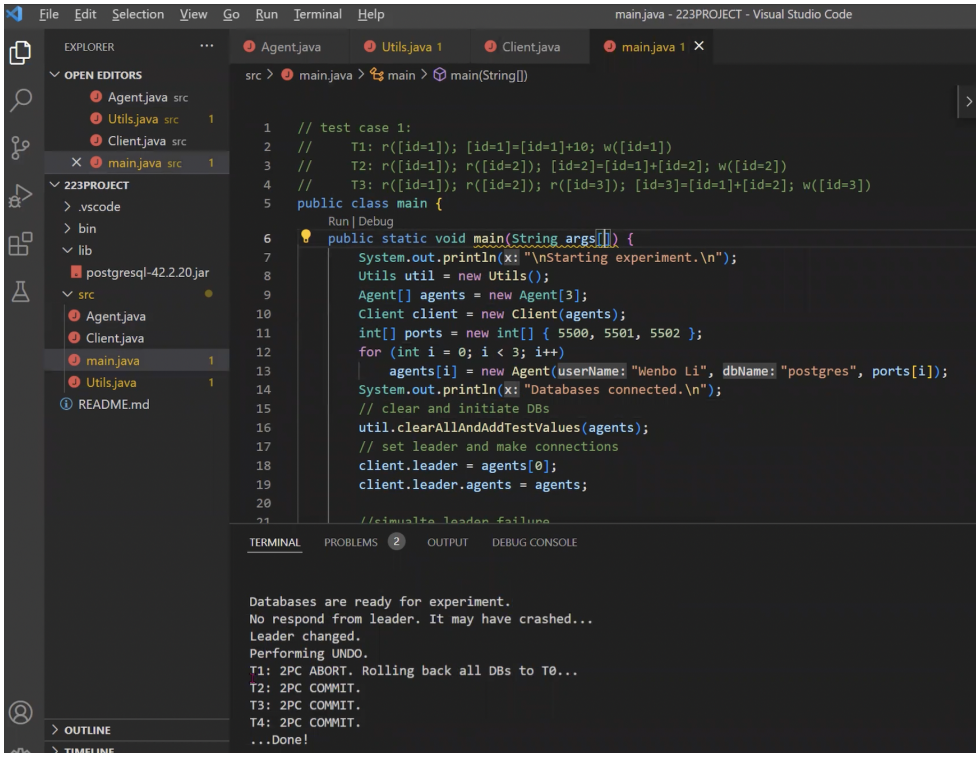

Figure 18: Leader-replacement situation 2: leader fails before commit.

5 Conclusion

We construct data replication with three servers corresponding to the three agents. Then, we implement the client processes transaction based on the Timestamp Ordering and the COMMIT request and 2PC for the leader and followers. Moreover, we extend the Read-only function by applying a snapshot here and the leader replacement with UNDO OR REDO based on different situations. Last but not the least, the followers can append log information sent by the leader to their replication logs. Overall, we successfully build a replicated database system for this project.

Note: there maybe some sub-functions that are not specifically mentioned in this report. Please refer to the source code for any further details.