A distributed and transactional storage system with replicas

Hang Yuan
Duke University

Yuhao Li
Duke University

Abstract

We implement a storage system with replicas, in which each server has *N* replicas, and replicas support write and read operations with guaranteed strong consistency. We integrate this storage system with the transactional bank transfer system implemented in Lab3. We demonstrate the effectiveness of our system with test cases. Results show: 1.) our system supports strong consistency across replicas. 2.) our system functions correctly on the failure of some replicas. 3.) our system can update a stale replica when it recovers from failure.

1 Introduction

Distributed storage systems are prevailing nowadays. Many aspects should be taken into account in designing a distributed system, such as scalability, reliability, and availability.

In this project, we focus on the reliability aspect of distributed system. It has been reported that $1,000$ out of $1,800$ clusters experiences failure in the first year[1]. To tolerate the high server failure rate, replica servers are added into the distributed system as backups for the primary storage server. For example, in Google File System, each chunk server has 3 replicas.

We propose our design of distributed storage system with replicas and integrate it into our Lab3. This design is tightly-coupled with the structure of the transactional bank transfer system we built in Lab3. Specifically, we found out that: 1.) to guarantee the strong consistency across replicas, we only need to guarantee the actions of acquiring locks for the data are consistent. 2.) updating the staled server is critical in the system.

To manage the strong consistency among replicas, we choose to select a primary server (based on the initial ID number) from replicas. Lock acquires are only sent to the primary. We decide to use a version number comparison algorithm to find out the server that contains latest information and uses it to update the stale server.

Next section presents the details of our design.

2 Design Overview

Our system is a transactional bank transfer system. It consists of three tiers: an application tier representing clients of a bank transfer system, a KV-client tier representing the bank transfer system, and a KV-store tier representing the storage system. Each server at the storage server has its replica. In the following parts, 3.1 gives a brief introduction on the original transactional system design; 3.2 describes the addition of replicas into the system; 3.3 describes how we simulate server failures; 3.4 provides detail APIs at client and storage server,

2.1 Transaction system Design

2.1.1 Application Tier Design

The application server represents an account owner. The account is represented by the application ID. The owner can either check its account, deposit money to its account, withdraw money from its account or transfer money from its account to another account. All these operations should be sent into the client tier after `Begin_transaction` is sent to the client tier, and before the `End_transaction` is sent to the client tier.

2.1.2 KV-client Tier Design

A client receives transaction operations from application tier, and then issues corresponding read/write requests to the store servers. Since a client server may issue requests to different servers, it needs to use the two-phase commit protocol to coordinate the commit: the client is the coordinator in the two-phase commit protocol. When it wants to commit all the operations, it first sends an `init_commit` to all the store servers that have received

requests from the client and waited for the feedback. If all the store servers agree to commit, the client then sends a commit to all the store servers. If one or more of the servers disagree to commit, the clients send an abort to all the store servers to ask them to abort all the write operations that are executed before. Furthermore, a client needs to grab locks from the servers when it needs to access the data. To avoid deadlocks, we use the two-phase locking protocol in this lab. A client needs to figure out all the locks it need before it begins to acquire locks, and ask for all the locks at one time – this is achieved by sending a request of a list of locks to servers at one time. The client releases locks when it either commits or aborts. A client server also maintains a cache into improve the read efficiency. When the client server tries to issue a read operation to the server, it first checks whether it already caches the data or not. When it writes to the server, it will write to the cache and validate the cache in other clients simultaneously.

2.2 Replica System Design

We decide to add replicas to the storage tier. Each storage server at the storage tier has N-1 replicas, where N is a configurable parameter. The each Each key is mapped to a cluster of N servers using a mod

There are two design problems we need to answer when designing such a replica system: 1.how to ensure the consistency between replicas? 2.What to do when a failure (either crash of one machine or a network partition).

First, we found out that consistency is guaranteed inside one transaction, as all the operations in one transaction are atomic. Furthermore, In our design, one operator in a transaction can be issued to storage tier only when locks of all operations are grabbed by the client. Therefore, we only need to ensure all the locking operations are consistent across replicas.

To ensure the consistent locking, we select one replica as the primary replica. When a client wants to acquire a lock for a key, it contacts the primary server and asks for locks. The primary server handles the lock request and update the corresponding book-keeping structure. After updating the book-keeping data structures, the primary broadcasts updates to the replicas. Each replica receives update operations and update the corresponding book-keeping structures. The primary only reports success when it receives responses from all replicas. If the client successfully obtain locks from the primary server, it can begin issue operations. The client reads from the primary and write to all replicas.In this way, we can ensure that all the updates are consistent across replicas.

When a server fails, it can be either primary or the secondary replica. The client would notice the failure of the primary while it tries to acquire locks from the primary or read from the primary. If a primary fails, the client acts a coordinator to select the new primary from the replica. It first sends messages to notify the replica to notify the failure of the primary, and randomly choose the new primary based on this rule: if number i^{th} server in the N replicas serves as the primary and it fails, then number $(i+1)^{th}$ server becomes the new primary. We choose this simple algorithm because it can be consistent to different clients: we do not need to broadcast the new primary to all the clients at the KV-client tier if we use this algorithm, since each client automatically learns the new primary if the old one fails. The old primary cannot come back to be the primary even after it recovers from the failure. When a secondary replica fails, the client stops sending operations to it.

When a server recover from failure (either crash or network partition), it needs to be updated with the latest information:locking information and key-value data. We rely on version numbers of each key to select latest data to update the stale server. Each working server maintains a version number for each key. When the client needs to select the server containing the latest information, it compares the list of version number between each client and obtains the latest server from comparison. When updating the stale server, we just pass all data structures containing critical information from one server to the stale server. This may cause large overhead in the real system, but does not affect the emulation performance in the scala system.

2.3 Failure Simulation

We define two APIs to emulate failure during transactions. The LoadMaster can send `Disconnect_replica()` to replica to make it disconnected. When a replica fails, it will not process any request from a client. If the client requires a reply from the replica, the replica will send a *FAILED* back to let the client know that the replica is disconnected. The LoadMaster can send `Connect_replica()` to reconnect the replica. In the beginning, we assume that all replicas are connected.

2.4 Lists of APIs at each layer

2.4.1 Client Tier APIs

The client provides following APIs to the application tier:

`Begin_Transaction`: When a client receives this signal, it creates a operation list with that ID for later purpose.

`Balance`: The application sends the balance query to ask for the balance in its account. The client transforms

the balance query into a read operation and appends it onto the operation list.

`Deposit`: The application sends the deposit query to deposit some amount of money into his account. The client transforms the deposit query into one read operation and one write operation.

`Withdraw`: The application sends the withdraw query to withdraw some amount of money from his account. The client transforms the deposit query into one read operation and one write operation.

`Transfer`: The application sends the transfer query to transfer some mount of money from his account to another account. The client transforms transfer operations into one read operation and two write operations.

`End_Transaction`: The application sends `End_Transaction` to the client to finish transaction. Upon receiving the `End_Transaction`, the client determines the locks for this transaction and the store server involved in this transaction. Then it sends all the lock requests to the servers. If it successfully obtain all the locks, it then continues to send operations to the store servers. Otherwise it aborts the transaction and send the abort message to the client. When the client sends out all the operations. It begins to commit using two-phase commit protocol. The client releases all the locks at the end of two-phase commit protocol.

2.4.2 Storage Tier APIs

The server provides following APIs to the client for transactions:

`Primary_Grap_locks`: Clients use this API to ask for locks and initiate the transaction if the locks are acquired. Client only sends this API to the primary.

`Put`: Clients use this API to issue a write request to the server. `Get`: Clients use this API to issue a read request to the server. `Init_commit`: Clients use this API to initiate the commit at the first stage of two- phase commit.

`Commit`: Clients use this API to finalize the commit at the second stage of two-phase commit.

`Abort`: Clients use this API to abort the commit at the second stage of two-phase commit.

3 Evaluation

First, we did the test that we used in Lab 3 to test the correctness of the system with replicas (with no failure). In this test case, there are 10 bank accounts altogether. In the first loop, each application does some transactions without interacting with other applications. And after the loop, the balance is 40.

```
for(i <- 0 until 10)
{
    applications(i) ! App_Begin_transaction()
    applications(i) ! App_check()
    applications(i) ! App_deposit(100)
    applications(i) ! App_withdraw(60)
    applications(i) ! App_End_transaction()
}
```

```
3: 1 done, balance:0
5: 1 done, balance:0
4: 1 done, balance:0
8: 1 done, balance:0
0: 1 done, balance:0
2: 1 done, balance:0
7: 1 done, balance:0
9: 1 done, balance:0
4: 1 done, balance:100
5: 1 done, balance:100
8: 1 done, balance:100
9: 1 done, balance:100
7: 1 done, balance:100
8: 1 done, balance:40
9: 1 done, balance:40
5: 1 done, balance:40
4: 1 done, balance:40
7: 1 done, balance:40
6: 1 done, balance:0
1: 1 done, balance:0
8 : 1 commit
9 : 1 commit
6: 1 done, balance:100
6: 1 done, balance:40
3: 1 done, balance:100
6 : 1 commit
3: 1 done, balance:40
2: 1 done, balance:100
3 : 1 commit
4 : 1 commit
7 : 1 commit
5 : 1 commit
2: 1 done, balance:40
2 : 1 commit
1: 1 done, balance:100
0: 1 done, balance:100
1: 1 done, balance:40
0: 1 done, balance:40
1 : 1 commit
0 : 1 commit
```

test code and output of Loop 2

In the second loop, 20 is transferred from the accounts with even indexes to accounts with odd indexes. So the accounts with even indexes will have 20 and others will have 60.

```
for(i <- 0 until 5)
{
    applications(2*i) ! App_Begin_transaction()
    applications(2*i) ! App_transfer(2*i+1,20)
    applications(2*i) ! App_End_transaction()
}

Thread.sleep(2000)


for(i <- 0 until 5)
{
    applications(2*i+1) ! App_Begin_transaction()
    applications(2*i+1) ! App_check()
    applications(2*i+1) ! App_End_transaction()
}
```

```
0: 2 done, balance:20
4: 2 done, balance:20
6: 2 done, balance:20
2: 2 done, balance:20
8: 2 done, balance:20
0 : 2 commit
4 : 2 commit
2 : 2 commit
6 : 2 commit
8 : 2 commit
3: 2 done, balance:60
1: 2 done, balance:60
7: 2 done, balance:60
9: 2 done, balance:60
5: 2 done, balance:60
3 : 2 commit
1 : 2 commit
7 : 2 commit
5 : 2 commit
9 : 2 commit
```

test code and output of Loop 2

In the third loop, we are going to test the concurrent transactions. We can see that there is a conflict between applications with even indexes and applications with odd indexes. From the result, we can see that only one of the pair of transactions is committed and the other one is aborted.

```
for(i <- 0 until 5)
{
    applications(2*i) ! App_Begin_transaction()
    applications(2*i) ! App_check()
    applications(2*i+1) ! App_Begin_transaction()
    applications(2*i+1) ! App_transfer(2*i,10)
    applications(2*i+1) ! App_End_transaction()
    applications(2*i) ! App_End_transaction()
}
```

```
4: 3 done, balance:20
2: 3 done, balance:20
8: 3 done, balance:20
2 : 3 commit
4 : 3 commit
8 : 3 commit
1: 3 done, balance:50
7: 3 done, balance:50
1 : 3 commit
7 : 3 commit
6 : 3 abort
3 : 3 abort
5 : 3 abort
0 : 3 abort
9 : 3 abort
```

test code and output of Loop 3

In the final loop, we check the balance of all accounts. The balance of each account is right. So the system works well within and without concurrent transactions.

```
for(i <- 0 until 10)
{
    applications(i) ! App_Begin_transaction()
    applications(i) ! App_check()
    applications(i) ! App_End_transaction()
}
```

```
7: 4 done, balance:50
0: 4 done, balance:30
2: 4 done, balance:20
5: 4 done, balance:60
4: 4 done, balance:20
7 : 4 commit
0 : 4 commit
1: 4 done, balance:50
4 : 4 commit
8: 4 done, balance:20
3: 4 done, balance:60
2 : 4 commit
5 : 4 commit
1 : 4 commit
9: 4 done, balance:60
8 : 4 commit
3 : 4 commit
6: 4 done, balance:30
9 : 4 commit
6 : 4 commit
```

test code and output of Loop 4

The second test is to test the correctness of the system during failures of replicas. We assume that there is always majority in a sharding so that the system can work well. In our test, we invalidate the local cache of client on purpose to make sure that during each transaction, the client has to contact with the replicas. There are four transactions in the test case and they are all the same. The difference is which of the replicas are working during each transaction. In the first one, all replicas are working. In the second one, replica 0 and replica 1 are disconnected and they become stale. Before the third transaction, replica 0 replica 1 reconnect and replica 3 replica 4 disconnect. So among all the replicas, only replica 2 has the newest data. After the third transaction, replica 2 disconnect and replica 3 replica 4 reconnect. So at this moment, all of the replicas have disconnected at some specific moment. However, according to the result of four transactions, the system is working well. The balance of the account is always correct and never becomes stale.

```
applications(0) ! App_Begin_transaction()
applications(0) ! App_deposit(5)
applications(0) ! App_End_transaction()

Thread.sleep(2000)

server(0) ! Disconnect_replica()
server(1) ! Disconnect_replica()

applications(0) ! App_Begin_transaction()
applications(0) ! App_deposit(5)
applications(0) ! App_End_transaction()

Thread.sleep(2000)

server(0) ! Connect_replica()
server(1) ! Connect_replica()
server(3) ! Disconnect_replica()
server(4) ! Disconnect_replica()

applications(0) ! App_Begin_transaction()
applications(0) ! App_deposit(5)
applications(0) ! App_End_transaction()

Thread.sleep(2000)

server(3) ! Connect_replica()
server(4) ! Connect_replica()
server(2) ! Disconnect_replica()

applications(0) ! App_Begin_transaction()
applications(0) ! App_deposit(5)
applications(0) ! App_End_transaction()

Thread.sleep(2000)
```

```
0: 1 done, balance:5
0 : 1 commit
replica 0 failed
replica 1 failed
replica 0 failed
replica 1 failed
0: 2 done, balance:10
0 : 2 commit
replica 3 failed
replica 4 failed
replica 3 failed
replica 4 failed
0: 3 done, balance:15
0 : 3 commit
replica 2 failed
replica 2 failed
0: 4 done, balance:20
0 : 4 commit
```

test code and output of test 2

4 Conclusion

In the final project, We implement a replica system with strong consistency, and integrate this system with the transactional system we implement in Lab3. We developed several test cases to validate its correctness.

References

[1] Failure Rates in Google Data Centers *http://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers/.*