Shirley Lu (shirllu@mit.edu)
Min Zhang (mzhang94@mit.edu)
Chongyuan Xiang (xiangcy@mit.edu)

Consistent Distributed Banking System (CDBS)

1. Introduction

CDBS is a distributed banking system that provides consistent money transactions. Clients can deposit, withdraw, check account balance, and transfer money to a group of clients' accounts. Our design is an enhanced version of the key/value service implemented in Lab 4, supporting cross-shard transactions. Similar to Spanner, CDBS ensures all-or-nothing atomicity. Our implementation integrates two-phase commit with Paxos to ensure consistency and fault tolerance and stores Paxos state on disk to promote persistence.

2. Design & Implementation

CDBS supports four major operations: deposit, withdraw, check balance, and multi-way transfer. Deposit and withdraw are similar to Put() and PutHash() operations in Lab 4. Check balance is the same as Get() operations in Lab 4. The major addition is henceforth transaction(), which supports cross-shard multi-user transactions.
Our main implementation consists of two parts: the new transaction() API and persistence to make our system resilient to different types of server crashing.

The link to our source code can be found at:
https://www.dropbox.com/sh/hx4thvf6ndg76hb/AADdqdEKr-szuA0RIOK6iqyEa

2.1 Supported APIs

```
bool Create(key string)
(bool, float64) CheckBalance(key string)
(bool, float64) Deposit(key string, value float64)
(bool, float64) Withdraw(key string, value float64)
(bool, []RequestReply) Transaction([]Request)
Request{
        RequestType string
        Key string
        Value float64
}
RequestReply{
        Err String
        Value float64
}
```

```
RequestType {
        Create
        CheckBalance
        Deposit
        Withdraw
}
```

2.2 Transaction Operation

To ensure atomicity, CDBS treats each transaction as a single entity. CDBS also treats single-group transactions differently from cross-group transactions. Single-group does not Paxos to reach consensus. Since intra-shard transactions performs significantly faster than cross-shard transactions, only cross-group transactions needs to be sent to the master to ensure consistency. To implement this enhancement, the client continuously pings the transaction master for its configuration. If there are multiple groups, then the transaction is sent to the transaction master, which stores all incoming transactions in a Paxos log and runs the Paxos algorithm to agree on their order. Otherwise, it is directly handed off to the key/value server. If the master has only one group, then the transaction is directly sent to the key/value server. Large transactions are divided into smaller transactions, each of which is based on only one shard. Each shard then handles these small transactions independently.

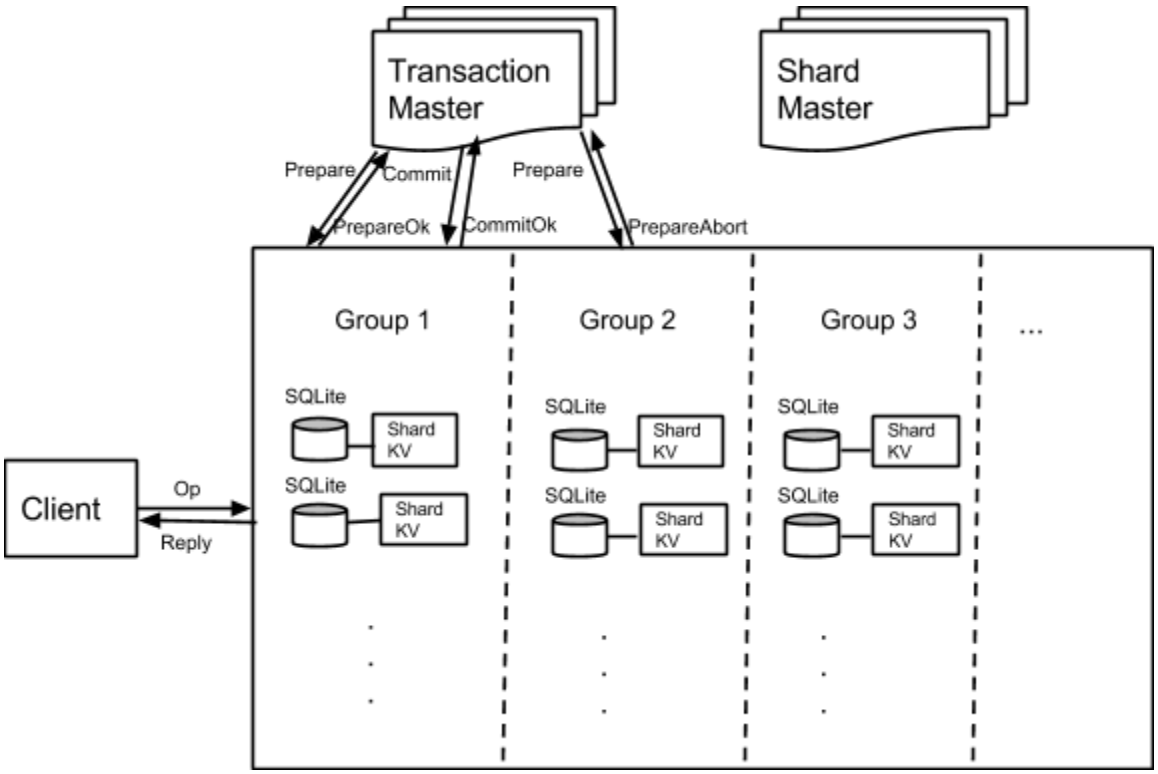

Figure 1. System Architecture

On receiving a client operation, shardkv decides whether it can process the operation on its own or needs to send the operation to the transaction master for reaching consensus.

2.2.1 Transaction server

Pseudocode:

```
Prepare Phase:
prepare_succ = true
for every small transaction t involving shard s:
for every shardkv server kv holding the shard:
send prepare request to s
if reply received
    if !reply.PrepareOk
        prepare_succ = false
    break


Commit Phase:
for every small transaction t involving shard s:
for every shardkv server kv holding the shard:
    if prepare_succ
send  commit request to s
else
    send abort request to s
if reply received and no error
    break
```

2.2.2 Shardkv server

Each server keeps two main data structures: transactionState[transactionId] map to store the state of each transaction, and preparing[accountName] map to avoid acknowledging requests while an account is preparing.

Prepare Handler

The handler first checks whether the incoming prepare request is valid. If it's not, the operation is aborted. Otherwise, it will be stored in transactionState map, with the transactionId being the key, and the following struct being the value:

reply
prepareOk bool
committed bool
changes map[string]float64

If the operation changes the balance on the account, such as create, deposit, or withdraw operations, then the changes on the account is stored in the changes map, and the server will try to store account-struct as key-value pair in the preparing map. If the account name already exists in the preparing map, it means that multiple clients are trying to sent prepare messages on the account. To avoid inconsistency, all such prepare messages are rejected. If it does not exist, then it is stored in the map and prepareOk is set to true.

Commit Handler

On receiving a commit, the server applies all the changes in the changes map, and deletes the key-value pair associated with that account name in the preparing map. committed is then changed to true.

Abort Handler
On receiving an abort, the server deletes the key-value pair associated with the transaction in the preparing map and sets the committed boolean to true.

If the transaction is only based on one group, then the handler sends the transaction directly to the key/value server by calling a Prepare-and-Commit request.

Prepare-and-Commit Handler
In the case of single-group transactions, the handler does not wait to prepare the transaction, it simply commits the transaction and performs the operations. If a request is rejected, then the encompassing transaction will be aborted in its entirety.

CheckBalance/Create/Deposit/Withdraw Handler
These handlers will be similar as *Put/Get* in lab4 except for one change. The server will reject requests for accounts that are in the preparing map.

2.3 Persistence
CDBS assumes that the shardmaster and transaction servers never experience machine failures. The system only considers persistence for shardkv servers.

The system leverages a SQLite database, that contains a separate table for each Paxos peer. Each table stores the v_a, n_a pair as well as the decided value for that Paxos table. The values are updated in the database before being changed in memory.

There are three major failure scenarios:

2.3.1 One server crashes and restarts with disk contents intact
In the first iteration of design, the crashed server is restarted by restoring and all Paxos states on disk to memory and replaying all of them from an empty k/v store. This is not scalable, however, when the Paxos log is long.

In the final design, CheckPoints are used to provide fast recovery. Each CheckPoint is a snapshot of all the data in a shardkv's memory, including maxOpSoFar (the sequence number of the operation that has been executed in the paxos log), data (the key/value store), currconfig (the current configuration), checkDuplicate (a map to store client's last request in order to guarantee at-most-one RPC), etc.

Shardkv servers periodically create CheckPoints, serialize them, and store them on text files. When servers restart, they recover memory states from the text files to avoid restoring unnecessary states.

2.3.2 Complete crash of all servers

All crashed servers undergo the same procedure as in 2.3.1 to recover.

2.3.3 Single server crashes and loses its disk contents

Based on the assumption that servers never come back up after a disk crash. CDBS starts a fresh server and tries to keep it up-to-date, by an RPC request to other replicas asking for a checkpoint. The server then starts to receive clients' requests from the CheckPoint.

3. Benchmark

Each of the following scenarios is measured with 10 clients constantly send requests to servers. Throughput is calculated with the following formula:

number of operations/ (the last returning time - the first returning time)

Average latency is calculated by taking the average of operation round trip time.

Before running each experiments, 500 Create requests have been processed beforehand to warm up since we found warming up increases performance significantly. We will discuss more on this in analysis.

3.1 Data Collection

Create:
Throughput = 7.92 ops/s
Average latency = 1.03 s
Variance of latency = 0.90 s

Deposit:
Throughput = 7.66 ops/s
Average latency = 1.01 s
Variance of latency = 1.31 s

CheckBalance:
Throughput = 6.81 ops/s
Average latency = 0.85 s
Variance of latency = 0.70 s

TransactionSameGroup:
Throughput = 13.82 ops/s
Average latency = 0.50 s
Variance of latency = 0.34 s

TransactionDifferentGroup:
Throughput = 10.15 ops/s
Average latency = 0.87 s

Variance of latency = 1.60 s

3.2 Analysis and Trends

3.2.1 Latency Varied a Lot

The latency varied from 0 to 5, so we examined the latency of running the insertOpToLog() function. We discovered that this function consumed the majority of our latency. The bottleneck is, therefore, the persistant Paxos. This is because the Paxos needs to write v_a, n_a pair to the database table to disk, which is comparatively expensive.

3.2.2 TransactionDifferentGroup performed slightly better than TransactionSameGroup

It corresponds to our expectation, as TransactionSameGroup does not need two-phase commit.

3.2.3 Performance at Different Point During the Tests Varied Significantly

This can be explained by the long warm-up time of SQLite databases. We inserted 0, 500, 1000 operations to the SQLite database respectively before running tests and noticed a positive relationship between the number of existing operations in database and throughput. For example, when the computer ran the tests right after start, each request took latency up to 20s and throughput is around 1 op/s. However, after hundreds of requests had been processed, latency is improved to 1 s and throughput improved to almost 10 ops/s. So we concluded that database warm-up time added a significant overhead to our latency. We believe that the big improvement resulted from SQLite's caching behavior.

4. Conclusion

To summarize, CDBS combines and extends on two major fault-tolerance techniques: Paxos and two-phase commit, to ensure consistency across distributed servers. CDBS is a service of significant practicality and has real world application, as money transfer is an essential banking operation, and ensuring that clients observe consistent behaviors across accounts determines banks' credibility and profits.