CSE 535: Distributed Systems
Project 3
Due: November 28, 2024 (11:59 pm)

Abstract

The objective of this project is to implement a fault-tolerant distributed transaction processing system that supports a simple banking application. To this end, we partition servers into multiple clusters where each cluster maintains a data shard. Each data shard is replicated on all servers of a cluster to provide fault tolerance. The system supports two types of transactions: intra-shard and cross-shard. An intra-shard transaction accesses the data items of the same shard while a cross-shard transaction accesses the data items on multiple shards. To process intra-shard transactions the modified Paxos protocol implemented in the first project (or the Multi-Paxos protocol) should be used while the two-phase commit protocol is used to process cross-shard transactions.

1 Project Description

We discuss the project in three steps. We first, explain the architecture of the system and the supported application followed by a brief overview of the modified Paxos protocol (from the first project). Finally, the two-phase commit protocol is discussed.

1.1 Architecture and Application

In this project, similar to the first and second projects, you are supposed to deploy a simple banking application where clients send their transfer transactions in the form of (x, y, amt) to the servers where x is the sender, y is the receiver, and amt is the amount of money to transfer. This time, we shift our focus towards real-world applications by examining a large-scale key-value store that is partitioned across multiple clusters, with each cluster managing a distinct shard of the application's data. The system architecture is illustrated in Figure 1.

As shown, the data is divided into three shards: D_1, D_2, and D_3. The system comprises a total of nine servers, labeled S_1 through S_9, organized into three clusters: C_1, C_2, and C_3. Each data shard D_i is replicated across all servers within its respective cluster C_i to ensure fault tolerance, operating under the assumption that servers adhere to a fail-stop failure model, where at most one server in each cluster may be faulty at any given time.

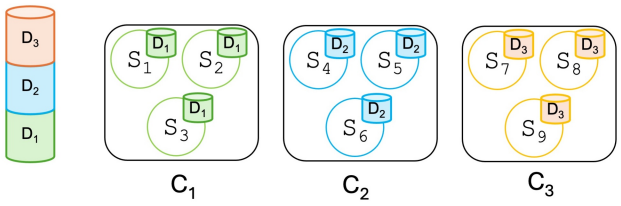

Figure 1: System Architecture

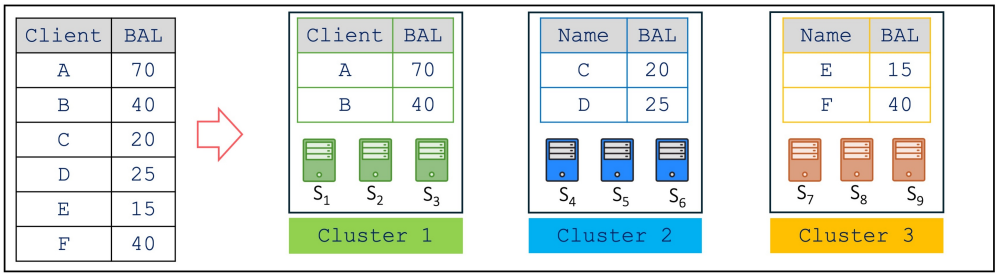

Figure 2: System Architecture

We assume that clients have access to the shard mapping, which informs them about the data items stored in each cluster of servers. Clients can initiate both intra-shard and cross-shard transactions.

For an intra-shard transaction, a client sends its request to a designated server within the relevant cluster (the cluster that holds the requested data items). The servers in that cluster then execute the modified (or Multi-) Paxos protocol to reach consensus on the order of the request. It's important to note that, unlike in the first project, consensus is required for each individual transaction, as there is no local log maintained.

In the case of a cross-shard transaction, the client sends its request to a designated server in each of the relevant clusters. In this scenario, the client acts as the coordinator for the two-phase commit protocol executed among the involved clusters.

Figure 2 shows an example of the system where a dataset consisting of 6 data items is partitioned into 3 different data shards. Each data shard is then maintained by (replicated on) a cluster of servers.

1.2 Intra-Shard Transactions: Modified Paxos Protocol

To process intra-shard transactions, we utilize a modified version of the Paxos protocol from the first project, with some adjustments. The key distinction in this project is that we require consensus for each individual transaction, and servers do not maintain any local transaction logs. This design enhances fault tolerance, as all transactions are replicated across all servers within a cluster. If you implemented Multi-Paxos in the initial project, you may choose to use it in place of the modified Paxos protocol to achieve better performance.

Here is a brief overview of the modified Paxos protocol. A client sends an intra-shard request (x, y, amt) to a server in the cluster that holds both data items x and y.

For simplicity, we assume that the leader server is designated for some amount of time (i.e., each cluster has a contact server that tries to become the leader every time it receives a request). The contact server, however, does not overwrite itself (i.e., does not initiate the next instance of consensus before receiving sufficient accepted messages for the previous one). This design choice makes your project much simpler, as many of the conflicting and livelock scenarios might not occur anymore.

Note that you should still be able to change the contact server in your implementation over time (each cluster has its own contact server in each test set and the contact server is supposed to be non-faulty throughout that test set).

When the contact (proposer) server receives an intra-shard request (x, y, amt) from a client, the server initiates the consensus protocol by sending a prepare message with ballot number n to all other servers in the cluster. As a reminder, servers must synchronize when sending the prepare message by sharing their states, which consist of either the number of committed transaction blocks or the ballot number of the latest committed block along with its associated transaction. It is important to note that in our project, every block corresponds to a single transaction. The same catch-up mechanism used in the first project should work here.

When a server receives a prepare message from a proposer server, it first compares its current datastore (sequence of committed transactions) with the datastore provided by the proposer. If the states do not match, the server takes the necessary steps to synchronize with the proposer or visa versa. If the datastores are consistent, the server responds by sending a promise message $\langle \text{ACK}, n, AcceptNum, AcceptVal \rangle$ back to the proposer, where AcceptNum and AcceptVal indicate the latest accepted (but not yet committed) transaction and the ballot number in which it was accepted. Note that the servers might need to update even their AcceptNum and AcceptVal once they are synchronized (if these variables are related to some old ballot number and not valid anymore). Unlike in Project 1, there is no need to send local logs, as this component is not applicable in the current project.

Once the leader is synchronized and receives a sufficient number of promise messages, it first checks two conditions: (1) there are no locks on data items x and y, and (2) the balance of x is at least equal to amt. If both conditions are met, the leader enters the accept phase and finally commits the request if all conditions are satisfied. The leader needs to obtain locks once it enters the accept phase and all other nodes obtain locks before sending the corresponding accepted messages.

The leader needs to send a message back to the client letting the client know that the transaction has been committed. The client waits for a reply from the leader to accept the result. For the sake of simplicity, you do not need to use timers on the client side or resend the requests.

1.3 Cross-Shard Transactions: Two-Phase Commit Protocol

To process cross-shard transactions, we need to implement the two-phase commit (2PC) protocol across the clusters involved in each transaction. Since our focus is on transfer transactions, each cross-shard transaction involves two distinct shards. In this configuration, the client acts as the coordinator for the 2PC protocol.

The client initiates a cross-shard request (x, y, amt) by contacting: (1) The contact (leader) server from the cluster that holds data item x, and (2) The contact (leader) server from the cluster that holds data item y.

Each of these servers then starts the modified Paxos protocol to achieve consensus on the transaction within their respective clusters, treating the transaction as an intra-shard transaction. While reaching consensus, the cluster responsible for data item x must verify that the balance of x is at least equal to amt.

To prevent concurrent updates to the data items, all servers in both clusters acquire locks (using two-phase locking) on accessed items (item x in one cluster and item y in the other).

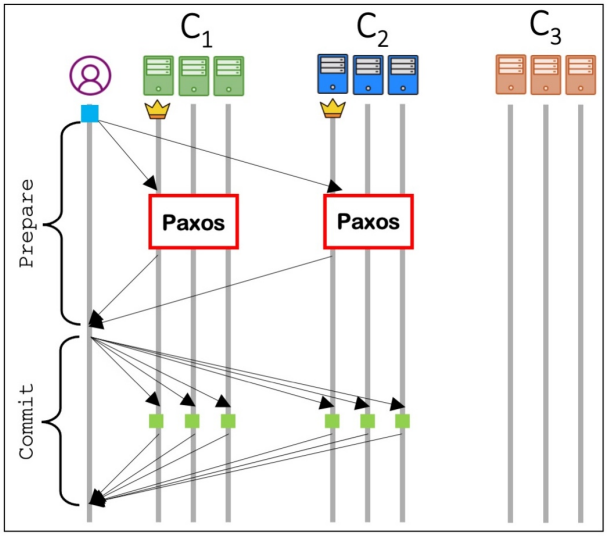

Figure 3: Cross-shard Transactions

If another transaction has already secured a lock on either item, the cluster will decide to simply abort the cross-shard transaction.

While the servers within the clusters reach an agreement on the transaction, they need to also update their write-ahead logs (WAL), allowing them to undo changes in the case of an abort. Upon reaching consensus a record will be added to the datastore of each node (in the same way as an intra-shard transaction)

The leader of each cluster sends either a prepared or abort message back to the transaction coordinator. Upon receiving prepared messages from both involved clusters, the client (coordinator) sends a commit message to all servers in both clusters. Conversely, if any cluster aborts the transaction or if the transaction coordinator times out, the coordinator will issue an abort message to all servers in both clusters.

If the outcome is a commit, each server releases its locks. If the outcome is an abort or if a timeout occurs, the server will use the WAL to undo the executed operations before releasing the locks. In either scenario, the server sends an Ack message back to the coordinating client and adds an entry to its datastore. It should be noted that this is a simplified (probably unsafe) design of 2PC over Paxos, as in its original design servers need to run consensus again to make sure that the "commit" or "abort" entry is also appended to the datastore of all nodes while in our design the client simply multicasts the outcome to all nodes.

Figure 3 demonstrates the flow of a cross-shard transaction between clusters C_1 and C_2. As mentioned before, when a cross-shard transaction is performed, each server (within one of the involved shards) needs to append two entries to its datastore: one after the prepare phase and one after the commit phase.

Figure 4 shows how a sequence of transactions updates different data structures of a single server. We consider a simple database with only 4 clients. During the processing of intra-shard transaction (A, B, 20), locks on records A and B are acquired and released by every node. As shown, the transaction is executed on the database resulting in updating the balance of A and B. The second transaction is a cross-shard transaction between clients

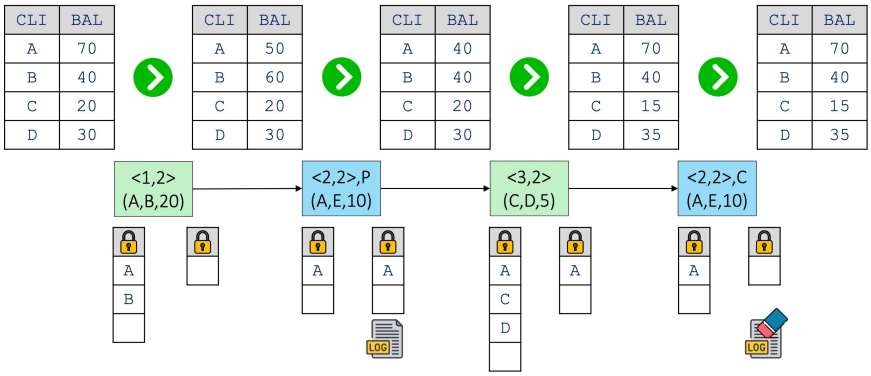

Figure 4: Processing Transactions

A and E. Here nodes acquire the lock on record A, run Paxos, append an entry to the datastore showing that the prepare phase was successful, execute the transaction (update the balance of A) and add records to the WAL. Before receiving the commit message for the second transaction, another intra-shard transaction (C, D, 5) takes place and updates the database and the lock table. Finally, the server receives a commit message from the client for the second transaction. The server appends an entry to the datastore, releases the lock on record A, and deletes the corresponding records from the WAL.

2 Implementation Details

Your implementation should support a dataset of 3000 data items with $id = 1$ to $id = 3000$. As shown in Figure 1, there are 9 servers organized into three clusters of size 3. The data needs to be maintained in a key-value database (use some lightweight DBMS, using files is not accepted). Your program should first read a given input file. This file will consist of a set of triplets (x,y,amt). Assume all data items (i.e., client records) start with 10 units. A client process sends requests to the (randomly chosen) servers from the corresponding shard(s). Relying on a single client process to initiate all requests would be okay. Each request is a transfer (x,y,amt) which is either intra-shard (if x and y belong to the same shard) or cross-shard (if x and y belong to different shards).

NOTE: For this assignment, a fixed shard mapping has been defined, and it is **mandatory** to use this provided shard mapping in your implementation. This ensures consistency in testing your solution and establishes a common understanding of intra-shard and cross-shard transactions for everyone.

Cluster	Data Items
C1	[1,2,3,...,1000]
C2	[1001,1002,...,2000]
C3	[2001,2002,...,3000]

Table 1: Shard Mapping

IMPORTANT: A transaction should be aborted not only in cases of insufficient balance or lock unavailability but also when the system fails to reach consensus, such as in the absence of a majority agreement. Additionally, any other scenario where consensus cannot be achieved will also result in the transaction being aborted.

- Your program should be able to process a given input file containing a set of transactions, with each set representing an individual *test case* (see section 6.4 for more details).
- Your program should have a PrintBalance function which reads the balance of a given client (data item) from the database and prints the balance on all servers. This function accepts a client ID (or data item ID) as input and retrieves the balance for the specified client ID across all servers within the corresponding cluster.
- Your program should have a PrintDatastore function that prints the set of committed transactions on each server (i.e., datastore in your first project). Note that this data structure maintains all committed transactions and is different from the key-value store that maintains the balance of all clients. Refer to the test case guide for the required log format in the datastore.
- Your program should have a Performance function which prints throughput and latency. The throughput and latency should be measured from the time the client process initiates a transaction to the time the client process receives a reply message.
- While you may use as many log statements during debugging please ensure such extra messages are not logged in your final submission.
- We do not want any front-end UI for this project. Your project will be run on the terminal.

3 Bonus!

We briefly discuss some possible optimizations that you can implement and earn extra credit.

1. **Shard redistribution.** The number of clusters can fluctuate over time due to factors such as the addition of more resources or the complete failure of a cluster. In these cases, it may be necessary to efficiently redistribute data items across the various clusters. For instance, if multiple transactions have accessed two particular data items, those items should reside within the same data shard following the redistribution. Our objective is to (1) create a resharding function that determines the optimal mapping of clusters and data items based on transaction history (to reduce the number of cross-shard transactions while shards are still balanced), and (2) carry out the resharding effectively by moving data items between different clusters. The input to this function should be the new clustering scheme, which is different from the initial one, e.g., adding cluster C_4 or removing cluster C_3.

2. **Configurable clusters.** Your implementation is supposed to support three clusters of size three. One easy extension is to make your system scalable by letting the user choose the number of clusters and the cluster size (number of servers per cluster), e.g., 4 clusters each with 5 servers.

4 Submission Instructions

4.1 Lab Repository Setup Instructions

Our lab assignments are distributed and submitted through GitHub. If you don't already have a GitHub account, please create one before proceeding. To get started with the lab assignments, please follow these steps:

1. **Join the Lab Assignment Repository**: Click on the provided link to join the lab assignment system.

   Important: If you are not able to accept the assignment please contact one of us immediately for assistance.

To set up the lab environment on your laptop, follow the steps below. If you are new to Git, we recommend reviewing the introductory resources linked here to familiarize yourself with the version control system.

Once you accept the assignment, you will receive a link to your personal repository. Clone your repository by executing the following commands in your terminal:

```
$ git clone git@github.com:F24-CSE535/2pc-<YourGithubUsername>.git
$ cd 2pc-<YourGithubUsername>
```

This will create a directory named `2pc-<YourGithubUsername>` under your home directory, which will serve as the Git repository for this lab assignment. These steps are crucial for properly setting up your lab repository and ensuring smooth submission of your assignments.

4.2 Lab Submission Guidelines

For lab submissions, please push your work to your private repository on GitHub. You may push as many times as needed before the deadline. We will retrieve your lab submissions directly from GitHub for grading after the deadline.

To make your final submission for Lab 3, please include an explicit commit with the message **submit lab3** from your main branch. Afterward, visit the provided link to verify your submission.

NOTE: Please note that, unlike the previous assignment where submission mistakes like incorrect commit messages or workflow file modifications were waived without penalties, this lab will not allow any exceptions. Follow all instructions carefully, as no waivers will be granted for submission errors.

5 Deadline, Demo, and Deployment

This project will be due on November 24. We will have a short demo for each project (the date will be announced later).

6 Tips and Policies

6.1 General Tips

- Start early!
- Read and understand the two-phase locking and two-phase commit lecture notes before you start.

6.2 Implementation

- You need to continue with the programming language used in your first and second projects.
- Your implementation should demonstrate reasonable performance in terms of throughput and latency, measured by the number of transactions committed per second and the average processing time for each transaction. We will assess the performance of all submissions against each other and projects with unreasonably poor performance may face point deductions.
- There is no need for signing messages or adding digests in the messages as the setting is trusted.

6.3 Possible Test Cases

Below is a list of some of the possible scenarios (test cases) that your system should be able to handle.

- All test cases of project 1
- Concurrent independent intra-shard transactions in different clusters
- Concurrent independent cross-shard transactions
- Concurrent intra-shard and cross-shard transactions
- Concurrent (intra-shard or cross-shard) transactions accessing the same data item(s)
- All possible failures and timeout scenarios discussed in the two-phase commit protocol
- No consensus if too many servers fail (disconnect)
- No commitment if any cluster aborts

6.4 Example Test Format

The testing process involves a csv file (.csv) as the test input containing a set of transactions along with the corresponding live servers involved in each set. A set represents an individual test case and your implementation should be able to process each set sequentially i.e. in the given order.

Your implementation should prompt the user before processing the next set of transactions. That is, the next set of transactions should only be picked up for processing when the user requests it. Until then, depending on your implementation, all servers should remain idle until prompted to process the next set. You are not allowed to terminate your servers after executing a set of transactions, as consecutive test cases may be interdependent.

After executing one set of transactions, when all the servers are idle and waiting to process the next set, your implementation should allow the use of functions from section 2.1.1 (such as *PrintBalance*, *printDatastore*, etc.). The implementation will be evaluated based on the output of these functions after each set of transactions is processed.

The test input file will contain four columns:

1. **Set Number**: Set number corresponding to a set of transactions.
2. **Transactions**: A list of individual transactions, each on a separate row, in the format `(Sender, Receiver, Amount)`.
3. **Live Servers**: A list of servers that are active and available for all transactions in the corresponding set.
4. **Contact Servers**: One of the live servers in each cluster that attempts to become the leader every time it receives a transaction.

An example of the test input file is shown below:

Set Number	Transactions	Live Servers	Contact Servers
1	(21, 700, 2)	[S1, S2, S4, S6, S8, S9]	[S1, S4, S8]
	(100, 501, 8)		
	(1001, 1650, 2)		
	(2800, 2150, 7)		
	(1003, 1001, 5)		
2	(702, 1301, 2)	[S1, S2, S3, S5, S6, S8, S9]	[S3, S6, S8]
	(1301, 1302, 3)		
	(600, 1502, 6)		

Table 2: Example Test Input File

This example test scenario demonstrates the basic structure and approach that will be used to assess your implementation.

6.5 Grading Policies

- Your projects will be graded based on multiple parameters:

  1. The code successfully compiles and the system runs as intended.
  2. The system passes all tests.
  3. The system demonstrates reasonable performance.
  4. You are able to explain the flow and different components of the code and what is the purpose of each class, function, etc.
  5. The implementation is efficient and all functions have been implemented correctly.
  6. The number of implemented and correctly operating additional bonus optimizations (extra credit).

- **Late Submission Penalty**: For every day that you submit your project late, there will be a 10% deduction from the total grade, up to a maximum of 50% deduction within the first 5 days of the original deadline. Even after 5 days past the original deadline, you still have an opportunity to submit your project until the deadline of project 4 and still receive 50% (assuming the project works perfectly). This policy aims to encourage punctual submission while still allowing students with extenuating circumstances an opportunity to complete and submit their work within a reasonable timeframe.

6.6 Academic Integrity Policies

We are serious about enforcing the integrity policy. Specifically:

- The work that you turn in must be yours. The code that you turn in must be code that you wrote and debugged. Do not discuss code, in any form, with your classmates or others outside the class (for example, discussing code on a whiteboard is not okay). As a corollary, it's not okay to show others your code, look at anyone else's, or help others debug. It is okay to discuss code with the instructor and TAs.

- You must acknowledge your influences. This means, first, writing down the names of people with whom you discussed the assignment, and what you discussed with them. If student A gets an idea from student B, both students are obligated to write down that fact and also what the idea was. Second, you are obligated to acknowledge other contributions (for example, ideas from Websites, AI assistant tools, Chat GPT, existing GitHub repositories, or other sources). The only exception is that material presented in class or the textbook does not require citation.

- You must not seek assistance from the Internet. For example, do not post questions from our lab assignments on the Web. Ask the course staff, via email or Piazza, if you have questions about this.

- You must take reasonable steps to protect your work. You must not publish your solutions (for example on GitHub or Stack Overflow). You are obligated to protect your files and printouts from access.

- Your project submissions will be compared to each other and existing Paxos protocol implementations on the internet using plagiarism detection tools. If any substantial similarity is found, a penalty will be imposed.

- We will enforce the policy strictly. Penalties include failing the course (and you won't be permitted to take the same class in the future), referral to the university's disciplinary body, and possible expulsion. Do not expect a light punishment such as voiding your assignment; violating the policy will definitely lead to failing the course.

- If there are inexplicable discrepancies between exam and lab performance, we will overweight the exam, and possibly interview you. Our exams will cover the labs. If, in light of your exam performance, your lab performance is implausible, we may discount or even discard your lab grade (if this happens, we will notify you). We may also conduct an interview or oral exam.

- You are welcome to use existing public libraries in your programming assignments (such as public classes for queues, trees, etc.) You may also look at code for public domain software such as GitHub. Consistent with the policies and normal academic practice, you are obligated to cite any source that gave you code or an idea.

- The above guidelines are necessarily generalizations and cannot account for all circumstances. Intellectual dishonesty can end your career, and it is your responsibility to stay on the right side of the line. If you are not sure about something, ask.