CS 271: Distributed Systems
Final Project
Demo and Due Date: March 7, 2025

network / router

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 Raft protocol should be used while the two-phase commit protocol (2PC) 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 Raft protocol. Finally, the two-phase commit protocol is discussed.

1.1 Architecture and Application

In this project, 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. Our focus is on real-world applications where a large-scale key-value store 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.

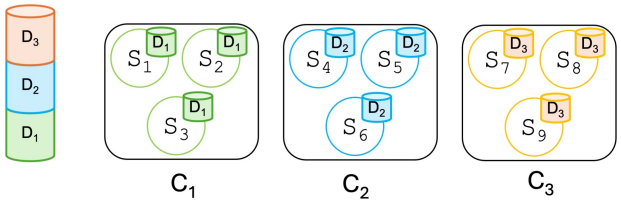

Figure 1: System Architecture

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.

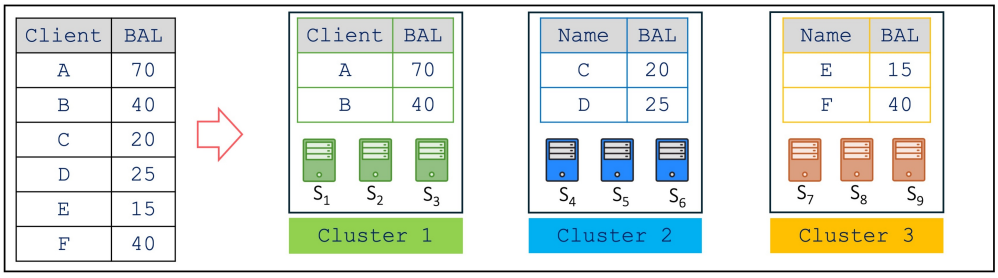

Figure 2: System Architecture

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 Raft protocol to reach consensus on the order of the request.

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.

so i'd better maintain a msg queue for each leader, and process them in order, and async reply to client?

1.2 Intra-Shard Transactions: Raft Protocol

To process intra-shard transactions, we utilize the Raft protocol, where consensus is reached for each individual transaction, which is clearly marked as *intra-shard*.

The leader needs to check 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 needs to obtain locks and all other nodes obtain locks too. The leader executes the Raft protocol. Once then entry is committed on the log, the data store is updated. For , we assume these two operations are atomic, hence no failures happens between these two events (centralized recovery algorithms using write-ahead logs can be used, but this is not the focus of this course). The leader then sends 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.

clients are blocked? or it can async and wait for a reply by the leader

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 and the transaction is clearly marked as *cross-shard*.

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.

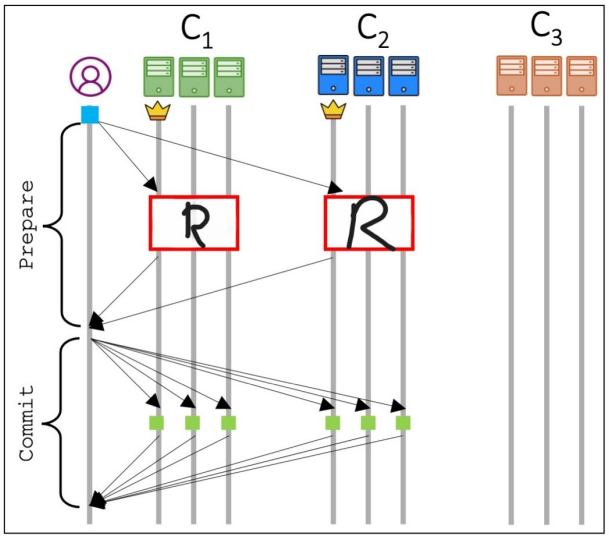

Figure 3: Cross-shard Transactions

Each of these servers then starts the Raft protocol to achieve consensus on the transaction within their respective clusters. 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 on accessed items (item x in one cluster and item y in the other). You will need to implement a *lock table* to keep track of locks. If another transaction has already secured a lock on either item, the cluster will decide to simply abort the cross-shard transaction.

Upon reaching consensus, the transaction is considered committed, but we do not execute the operation on the data store (as would be normal in Raft). We need to wait till the transaction is committed. Note this might not be efficient, but it will do for our project.

The leader of each cluster sends either a prepared/yes or abort message back to the transaction coordinator. Upon receiving prepared/yes messages from both involved clusters, the client (coordinator) sends a commit message to all servers in both clusters. At this point, the transaction is truly considered committed, and the corresponding data store is updated. 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. Now, the transaction is not executed on the data store.

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 not execute the operation and releases its locks. In either scenario, the server sends an Ack message back to the coordinating client. It should be noted that this is a simplified (probably unsafe) design of 2PC over Raft, 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.

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 (using a simple file is 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. Note that this data structure maintains all committed transactions and is different from the key-value store that maintains the balance of all clients.

- 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 Submission Instructions

Will be posted later.

4 Deadline, Demo, and Deployment

This project will be due on March 7. We will have a short demo for each project on that day, ie, March 7.

5 Tips and Policies

5.1 General Tips

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

5.2 Possible Test Cases

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

- 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