CSE 535: Distributed Systems
Project 3
Due: December 07, 2025 (11:59 pm)

Abstract

The goal of this project is to implement a practical, high-performance fault-tolerant distributed transaction processing system. To this end, we group nodes into multiple clusters where each cluster maintains a data shard. Each data shard is replicated on all nodes within a cluster to provide fault tolerance. The system supports both read-only and read-write transactions where read-write transactions are either intra-shard or cross-shard. An intra-shard transaction accesses the data items maintained by a single cluster (shard), while a cross-shard transaction accesses the data items across multiple shards. To process intra-shard transactions, the Multi-Paxos protocol implemented in the first project will be used, while the two-phase commit protocol is needed to process cross-shard transactions.

1 Project Description

We first explain the architecture of the system and the supported application. We then discuss the two-phase commit protocol and some required functionalities of this project.

1.1 Architecture and Application

In this project, similar to the first and second projects, you will deploy a simple banking application that allows clients to submit their requests. We support two types of requests: (a) balance (read-only) transactions (`s`), where the leader node of the corresponding cluster returns the balance of sender `s` and (b) transfer transactions (`s`, `r`, `amt`), where `s` represents the sender, `r` denotes the receiver, and `amt` specifies the amount of money to be transferred. While balance transactions can be performed without running consensus, consensus will be required for each individual transfer transaction, and we will utilize state machine replication to ensure that all transfer transactions are consistently executed across all nodes (replicas). 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.

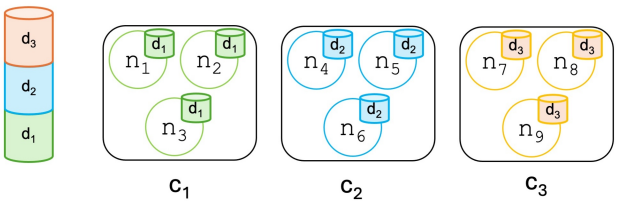

Figure 1: System Architecture

As shown, the data is divided into three shards: d_1, d_2, and d_3. The system comprises a total of nine nodes, labeled n_1 through n_9, organized into three clusters: c_1, c_2, and c_3. Each

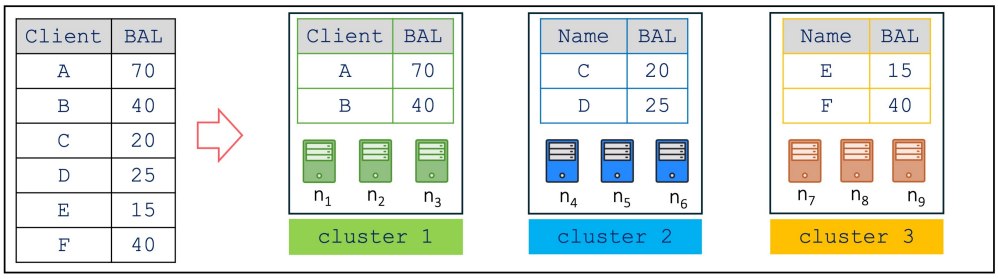

Figure 2: System Architecture

data shard d_i is replicated across all nodes within its respective cluster c_i to ensure fault tolerance, operating under the assumption that nodes adhere to a fail-stop failure model.

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 nodes.

Clients initially have access to the shard mapping, which informs them about the data items stored in each cluster of nodes. We later discuss how this mapping might get updated. Clients can initiate balance (read-only), intra-shard, and cross-shard transactions.

For a read-only transaction, a client sends its request to the leader node of the corresponding cluster and waits for a reply message from the leader node. If the client does not receive a reply and its timer expires, it retries by sending the transaction to another node or all nodes within the cluster. You need to develop a reasonable retry mechanism based on what we have learned and the possible design choices.

Intra-shard transactions are processed in the same manner as our first project, where clients send their requests to the leader node of the corresponding cluster, and the nodes within the cluster reach consensus on the transaction order and execute it accordingly. The only difference is that for each intra-shard transaction (s, r, amt), the leader needs to ensure that there are no locks on data items s and r. If data item s or r is already locked, the leader simply skips the transaction. Otherwise, it obtains locks on both items before sending an accept message to other nodes within the cluster. The locks are released once the transaction is executed.

Cross-shard transactions are sent by the client to the leader of the cluster that maintains the sender record (called the coordinator cluster) and will be processed by the involved clusters using two-phase commit on top of Multi-Paxos. We will discuss the protocol in more detail.

1.2 Two-Phase Commit Protocol

To handle cross-shard transactions, we need to implement the two-phase commit (2PC) protocol across the involved clusters. Since our focus is on transfer transactions, each cross-shard transaction involves two separate shards. In this setup, the cluster that maintains the sender's record serves as the coordinator for the 2PC protocol.

Client c sends its request message $m = \langle \text{REQUEST}, t, \tau, c \rangle$ including a cross-shard transaction t =(s, r, amt) to the leader node of the coordinator cluster (i.e., the cluster that

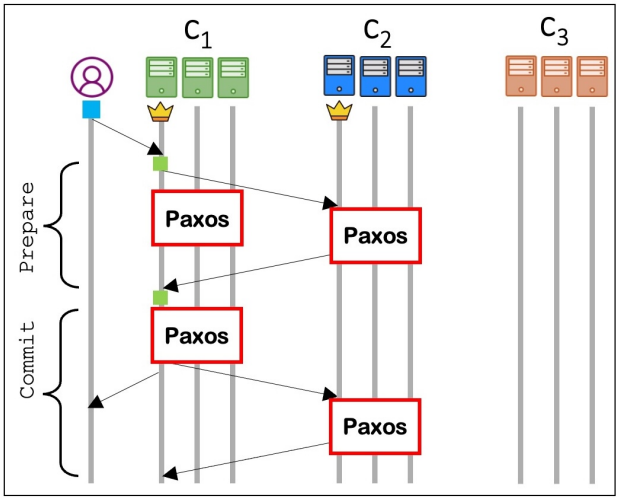

Figure 3: Cross-shard Transactions

maintains the data item s).

The leader of the coordinator cluster performs the following checks to ensure conditions for processing are met: 1. Confirm that there are no locks on the data item s, and 2. Verify that the balance of s is equal to or greater than amt.

If both conditions are satisfied, the leader node proceeds to: 1. Lock the record s, 2. Send a prepare message $\langle \text{PREPARE}, t, m \rangle$ to the leader of the participant cluster (the cluster that maintains data record r), and 3. Initiate an instance of the Multi-Paxos protocol within its own cluster by assigning a sequence number to the request and broadcasting an acc message $\langle \text{ACCEPT}, b, s, \text{P}, m \rangle$ with ballot number b and sequence number s to all nodes in the cluster. Since for each cross-shard transaction, two rounds of consensus are needed, the leader adds a parameter 'P' to the message to show that this consensus instance is for the prepare phase of a cross-shard transaction. You can use the digest of the request message instead of the actual message m.

The nodes within the coordinator cluster then run the consensus protocol to commit and execute the transaction. They must also maintain and update their write-ahead logs (WAL) to enable the possibility of undoing changes in the event of an abort.

The leader of the participant cluster also verifies that there are no locks on the data item r. If the record r is accessible, it will: 1. Lock the record r, and 2. Initiate an instance of the Multi-Paxos protocol within its cluster by assigning a sequence number to the request and sending an accept message $\langle \text{ACCEPT}, b, s, \text{P}, m \rangle$ to all nodes in the cluster in the same way as the coordinator cluster.

Once the leader of the participant cluster receives accept messages from a majority of nodes, it issues a commit message to the nodes of its cluster and sends a prepared message $\langle \text{PREPARED}, t, m \rangle$ to the leader of the coordinator cluster. Similar to the coordinator cluster, if the transaction is committed and executed, the nodes will update their write-ahead logs (WAL) to enable them to undo changes in case an abort occurs.

If there is a lock on r, the leader of the participant cluster initiates an instance of the Multi-Paxos protocol within its cluster by sending an accept message $\langle \text{ACCEPT}, b, s, \text{A}, m \rangle$ to

replicate the abort record (note the value A in the accept message). Once the transaction is committed, the leader node simply sends an abort message $\langle \text{ABORT}, t, m \rangle$ back to the leader of the coordinator cluster.

If the leader of the coordinator cluster receives a prepared from the leader of the participant cluster and the request has already been committed in its own cluster, it initiates the commit phase by: 1. Sending a 2PC commit message $\langle \text{COMMIT}, t, m \rangle$ to the leader of the participant cluster, and 2. Initiating consensus within its cluster to replicate the commit entry by broadcasting an acc message $\langle \text{ACCEPT}, b, s, \text{C}, m \rangle$ to all nodes in its own cluster. Here, the sequence number s is the same as the sequence number used in the consensus instance of the prepare phase, and these two rounds of consensus are distinguished from each other using 'P' and 'C' values.

On the other hand, if the participant cluster has aborted the transaction or the transaction coordinator times out or the consensus has not achieved in the coordinator cluster, the leader of the coordinator cluster initiates consensus within its cluster to replicate the abort entry by broadcasting an acc message $\langle \text{ACCEPT}, b, s, \text{A}, m \rangle$ to all nodes in its own cluster. The leader of the coordinator cluster then sends a 2PC abort message $\langle \text{ABORT}, t, m \rangle$ to the leader of the participant cluster. The 2PC abort message can be skipped if the participant cluster has already aborted the transaction (i.e., the participant cluster is no longer waiting for the outcome of the transaction).

Similarly, the leader of the participant cluster initiates consensus within its cluster to replicate the commit or abort entry by broadcasting an acc message to all nodes in its own cluster. The leader also sends an acknowledgment message back to the leader of the coordinator cluster once consensus on the second phase is achieved.

When the outcome is commit, each node will simply release the lock on the corresponding records. However, if the outcome is an abort or if a timeout occurs, the nodes will utilize the WAL to undo the executed operations before releasing the locks.

The leader of the coordinator cluster waits for an acknowledgment message from the leader of the participant cluster, and if the acknowledgment message is not received, it re-sends the commit or abort message. The leader of the coordinator cluster also sends a reply message back to the client, letting the client know about the outcome.

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 node (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 node. We assume that the node maintains a data shard with only four records A to D. 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 A and E. Here, nodes acquire the lock on record A, run Paxos, append an entry to the log 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 db and lock table. Finally, the node receives a commit message from the client for the second transaction. The node appends an entry to the datastore, releases

the lock on record A, and deletes the corresponding records from WAL.

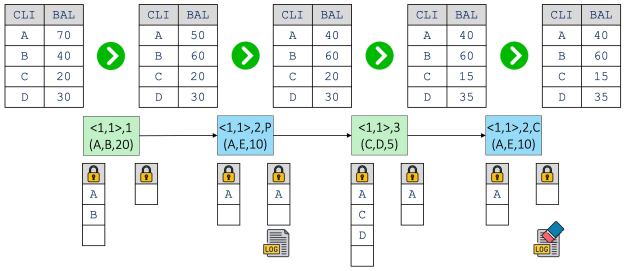

Figure 4: Example flow

1.3 Benchmarking

Testing the performance and functionality of distributed systems presents a significant challenge. To effectively evaluate and compare the performance of these systems, the use of standardized benchmarks is essential. These benchmarks consist of specifications and program suites designed for system assessment. In the area of distributed databases, several benchmarks have been proposed, including the Yahoo! Cloud Serving Benchmark (YCSB) and TPC-C (Transaction Processing Performance Council Benchmark C). Another relevant benchmark is SmallBank, which simulates a banking application.

For this project, you are required to implement one of the existing benchmarks. You may select any benchmark, but it must support three specific parameters:

1. The ability to configure the percentage of read-only versus read-write transactions,
2. The ability to adjust the percentage of intra-shard versus cross-shard transactions,
3. The option to modify the data distribution, allowing for a uniform distribution, where the likelihood of accessing various records is equal, or a skewed distribution, which can create "hotkeys" or "hotspots" by making a small number of data items more popular, thereby increasing the conflict rate. The benchmark should have a parameter to define the skewness of the workload (typically, in the range of 0 to 1).

1.4 Shard Redistribution

While we initially employ range partitioning to distribute data items across different clusters, this approach is not the most effective method of data sharding.

Firstly, cross-shard transactions are significantly more resource-intensive compared to intra-shard transactions. If the records accessed by cross-shard transactions were located within the same shard, we could save both time and resources.

Secondly, data access is not uniform; some (hotspot) records are accessed more frequently than others, leading to an unequal load distribution across clusters. Factors such as the distance between clients and their designated clusters, the varying hardware and network resources of these clusters, and their respective resilience can all contribute to load imbalance.

Our objective is to (1) develop a resharding function that determines the optimal mapping of data items to clusters based on transaction history, aiming to minimize cross-shard transactions while maintaining balanced shards, and (2) effectively carry out the resharding process by relocating data items between clusters.

To define the resharding function, we can utilize hypergraph partitioning. This involves modeling the last n transactions received as a graph, where the vertices represent data items and the hyperedges represent transactions. The goal is to partition the vertices of the hypergraph into three roughly equal-sized groups in such a way that the cost function for the hyperedges connecting vertices across different partitions is minimized.

For this project, the resharding function can be implemented in a centralized manner by a process that has visibility over all processed transactions (such as a client process). This central process can also manage the movement of data items. It is acceptable to carry out data movement in an offline manner when the system is not actively processing requests, meaning there is no need to be concerned about current transactions accessing moved data items.

2 Implementation Details

Nodes can be implemented using processes, coroutines, or threads. Processes are a better way of implementing nodes as they are independent and can simulate a distributed environment more naturally. Nodes should not have any shared memory or storage. Communication between nodes can be achieved through various methods. For instance, in CPP, TCP/UDP sockets are recommended for inter-process communication, but RPCs can also be used with some additional effort.

- Your implementation should support a dataset of 9000 data items with $id = 1$ to $id = 9000$. As shown in Figure 1, there are 9 nodes 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).

- Initially, we use a fixed range-based shard mapping, and it is **mandatory** to use the provided shard mapping in your implementation.

Cluster	**Data Items**
C1	`[1,2,3,...,3000]`
C2	`[3001,3002,...,6000]`
C3	`[6001,6002,...,9000]`

Table 1: Shard Mapping

- All users (data items) start with 10 units.

- A client process sends requests to the leader node of the corresponding shard. You can rely on a single client process to initiate all requests, and hence, the client process does not need to be closed-loop.

- At the beginning of every test set, the client sends its request to the first node of each cluster: n_1 for cluster c_1, n_4 for cluster c_2, and n_7 for cluster c_3. Hence, n_1, n_4, and n_7 should be elected as the leaders of the different clusters.

- The leader of each cluster processes requests *out-of-order*, which means it does not need to wait for the accepted or commit messages of the previous requests before sending the accept message for the next request. Each node maintains a datastore that can be represented as a key-value store to keep the balance of all clients.

- Your program should first read a given input file. This file will consist of read-only transactions (s), read-write transactions (s,r,amt) and special commands (i.e., 'F' and 'R').

- 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 nodes. This function accepts a client ID (or data item ID) as input and retrieves the balance for the specified client ID across all 3 nodes within the corresponding cluster. The input and output format should be the same as the following example.

  Input: PrintBalance(4005); **Output:** $n_4 : 8$, $n_5 : 8$, $n_6 : 10$

- Your program should have a PrintDB function which prints the balance of data items that have been modified in this test case on all nodes. When PrintDB is called, the function should output the results for all 9 nodes in parallel (with a single command).

- Your program should include a PrintView function that outputs all new-view messages (including all its parameters) exchanged since the start of the test case. If the system has undergone 3 leader elections during the test case, the PrintView function should display 3 new-view messages shared during these changes.

- 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.

- Your program should have a PrintReshard function that triggers data resharding functionality of the protocol and outputs the data records that have been moved between different clusters and their source and destination clusters. The output should be a set of triplets, e.g., $(2007, c_1, c_2)$.

- 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. **Multiple consistency levels.** Our current protocol implementation supports linearizability as its consistency level. Consider the various consistency levels we discussed and try to implement two additional consistency levels in your project. When a client submits a transaction, they should specify the desired consistency level, and your system must accommodate that. For example, a read-write request could be formatted as: (s, r, amt, linearizable).

2. **Configurable clusters.** Your implementation supports three clusters, each consisting of three nodes. A straightforward enhancement is to enable scalability by allowing users to select the number of clusters and the size of each cluster (the number of nodes per cluster). For instance, users could configure the system to have 4 clusters with 5 nodes each, incorporating the ability to perform sharding and resharding as required.

3. **High-performance development.** As previously mentioned, a key objective of this project is to create a practical system. Effective distributed transaction processing systems are capable of handling nearly 1 million transactions per second. While we do not expect your project to achieve 1 million transactions per second, due to the lack of optimizations like batching and the limitations of your local systems, you will earn bonus points if each cluster in your system can process $1,000$ read-write transactions per second, leading to a total of $3,000$ read-write transactions per second across all clusters.

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:F25-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

Push your work to your private repository on GitHub. To make your final submission for project 3, please include an explicit commit with the message **submit lab** on the **main** branch. Afterwards, visit the provided link and add your GitHub username at the end of the link to verify your submission. Submission after the deadline will override your previous submission.

4.3 Deadline, Demo, and Deployment

This project will be due on December 07. We will have a short demo for each project on December 12. For this project's demo, you can deploy your code on several machines. However, it is also acceptable if you just use several processes on the same machine to simulate the distributed environment.

5 Tips and Policies

5.1 General Tips

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

5.2 Implementation

- You need to continue with the programming language used in your first and second projects.
- In this project, in addition to ensuring correctness, we aim for your implementation to achieve high performance. It should exhibit satisfactory performance in terms of both throughput and latency, which will be evaluated based on the number of transactions committed per second and the average processing time for each transaction (measured from the time the request is sent to the protocol until the client receives a reply message). Throughput and latency must be assessed from the client process. We will compare the performance of all submissions, and projects that demonstrate unacceptably low performance may incur point deductions.
- There is no need for signing messages or adding digests in the messages, as the setting is trusted.

5.3 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 nodes fail (disconnect)
- No commitment if any cluster aborts
- Testing the resharding functionality
- Testing the benchmarking unit

5.4 Example Test Format

1. The testing process involves a csv file (.csv) as the test input containing a set of transactions along with the corresponding live nodes involved in each set. A set represents an individual test case, and your implementation should be able to process each set individually. You should also be able to skip a set if requested.

2. 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.

3. After executing one set of transactions, your implementation should allow the use of functions from section 2 (such as PrintBalance and printDB). The implementation will be evaluated based on the output of these functions after each set of transactions is processed.

4. The current project, similar to the second project, requires you to **FLUSH** the entire system state `before processing each set of transactions`. This means that all data maintained on both the nodes and clients must be cleared completely. Each set of transactions represents a separate scenario, and the system performs a **complete reset** before processing the next set.

5. You should be able to fail or recover any of the nodes upon request. Test cases include two special types of requests labeled 'F(n_i)' and 'R(n_i)', indicating that node n_i should fail or recover at that point. When failure happens, you do not need to terminate the node process; simply make the node act like a disconnected node. This means it won't log any received messages, send messages, or participate in any consensus instances until it recovers or the current set is completed. Furthermore, you must stop the failed node's timer; otherwise, it will continue running, leading to an increased ballot number.

As a result, when the node eventually recovers, it will be unable to participate in the consensus process. You can send the 'F' and 'R' commands using the client thread.

6. While you have the option to send the transactions of a set in parallel, please maintain the order of 'F' and 'R' commands for each cluster. If a set contains an 'F' and an 'R' command, you must first process the transactions before the first 'F', then the transactions between the 'F' and the 'R', and finally the transactions after the 'R'.

7. The test input file will contain three columns:

   (a) **Set Number**: Set number corresponding to a set of transactions.

   (b) **Commands**: A list of individual commands, each on a separate row, in one of the four formats: (Sender, Receiver, Amount), (Sender), 'F(n_i)' or 'R(n_i)'.

   (c) **Live Nodes**: A list of nodes that are active and available for all transactions in the corresponding set.

   An example of the test input file is shown below:

Set Number	Transactions	Live Nodes
1	(21, 700, 2)	[n_1, n_2, n_3, n_4, n_5, n_7, n_9]
	(100, 501, 8)	
	F(n_3)	
	(3001, 4650, 2)	
	(7800)	
	(5003, 4001, 5)	
2	(702, 4301, 2)	[n_1, n_3, n_4, n_5, n_7, n_9]
	(5301, 5302, 3)	
	R(n_6)	
	(600, 6502, 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.

5.5 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. You are able to answer our questions regarding the project topic and your implementation during the demo.
6. The implementation is efficient, and all functions have been implemented correctly.
7. 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 40% deduction within the first 4 days of the original deadline. The final deadline for submitting Project 3 and still receiving 60% (assuming the project works perfectly) is December 11 (as December 12 is the demo day for ll project 3 submissions).

5.6 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, ChatGPT, existing GitHub repositories, or other sources). The only exception is that material presented in class or the textbook does not require citation.

- You should not use AI assistant tools to implement any protocol-related part of the project. For instance, while it is OK to ask ChatGPT how to read a csv file, you are not allowed to use any tools to help you in implementing the Paxos leader election.

- You must not seek assistance from the Internet. For example, do not post questions from our 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, existing Paxos and 2PC protocol implementations on the internet, and projects from the last year 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 project performance, we will overweight the exam, and possibly interview you. Our exams will cover the projects. If, in light of your exam performance, your project performance is implausible, we may discount or even discard your project 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.

- We do not concern ourselves with your graduation or job offers; integrity is non-negotiable.

- 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.