CSE 535: Distributed Systems
Project 1
Due: October 11, 2024 (11:59 pm)

Abstract

The objective of this project is to implement a variant of the Paxos consensus protocol. Paxos can achieve consensus with at least $2f + 1$ nodes where f is the maximum number of concurrent faulty (crashed) nodes. The basic Paxos protocol can be used to reach an agreement on a single value. This makes Basic Paxos impractical in real distributed systems that need to process tens of thousands of requests every second. In this project, you are supposed to implement a sequence of Paxos instances. As part of the project, you will create a simple distributed banking application on top of the Paxos consensus protocol.

1 Project Description

1.1 Banking Application

In this project, you will deploy a simple banking application that allows clients to submit their transfer transactions as (s, r, amt), where s represents the sender, r denotes the receiver, and amt specifies the amount of money to be transferred. We will utilize state machine replication to ensure that all transactions are consistently maintained across all nodes (replicas), and consensus will be required for each individual transaction.

1.2 Basic Paxos Protocol

Basic Paxos, as discussed in the class, is a *leader-based* approach to consensus. We briefly explain the protocol in three main parts: (1) Leader Election, (2) Normal Operations, and (3) Node Failure.

Part I: Leader Election. Following the Paxos leader election routine (prepare and promise messages), there will be *exactly one leader*, and $2f$ *backups* after the leader election stage among the $2f + 1$ nodes. Proposer node n_p tries to become the leader only if it receives a request message m from some client. Node n_p initiates leader election by sending a $\langle \text{PREPARE}, b \rangle$ where b is its ballot number, e.g., if the system is just initiated, the ballot number will be $< 1, n_p >$. When a node receives the prepare with a ballot number higher than any previously received ballot number, the node sends an $\langle \text{ACK}, b, \text{AcceptNum}, \text{AcceptVal} \rangle$ message to the proposer node n_p promising not to accept any ballots smaller than b in the future. The promise message includes AcceptNum and AcceptVal to inform the leader about the latest accepted value (i.e., client transaction) and what ballot it was accepted in.

Part II: Normal Operations. Once node n_p becomes the leader, it broadcasts an accept message $\langle \text{ACCEPT}, b, m \rangle$ to all other nodes where b is the ballot number and m is the client request message. Note that if the leader receives any value (i.e., client request in our context) in promise messages, it proposes that value instead of proposing its own client request.

Upon receiving a valid accept message from the leader, each node sends an accepted message $\langle \text{ACCEPTED}, b, m \rangle$ to the leader.

The leader logs all accepted messages and once it receives f accepted messages from different nodes (plus itself becomes $f + 1$, a majority of the nodes), it broadcasts a commit message $\langle \text{COMMIT}, b, m \rangle$ (the same as decide message discussed in the lecture) to *all* nodes. The leader also executes the requested transaction and sends a reply message back to the client.

Part III: Node Failure. A failed node might be a backup or a leader. If a backup fails, the program should still be able to perform normal operations without any interruption as long as the number of faulty nodes is no more than f. In case of leader failure, Paxos should still be able to process transactions by going through a leader election. As discussed earlier, the goal of including AcceptNum and AcceptVal in promise messages is to let the new leader know that some transactions have been proposed and accepted (but might not be committed due to leader failure).

1.3 Stable-Leader Paxos

The Paxos protocol is designed for achieving consensus on a single value (such as a transaction). However, this approach becomes impractical in distributed applications that require continuous transaction processing. A naive solution might involve running a separate instance of Paxos for each transaction, necessitating leader election and replication phases for every single value. This method, however, incurs significant overhead, requiring at least two rounds of communication between a proposer (leader) and other nodes. Additionally, with multiple concurrent proposers, the likelihood of conflicts and restarts increases substantially.

Upon close examination of the two primary phases of the Paxos protocol, it becomes evident that no consensus values are exchanged during the first phase (prepare). In this phase, the proposer selects the highest unique ballot number, secures votes from a majority for this ballot, and gathers information on all smaller ballot outcomes. It is only in the second phase (accept) that the leader proposes its initial value or the most recent value learned from phase one.

Inspired by Multi-Paxos, this project aims to elect a leader to manage multiple Basic Paxos executions rather than conducting a separate election for each instance. By adopting this approach, a single prepare phase would be sufficient for multiple transactions, eliminating the need for a leader election phase with every request. The prepare phase will only be executed if the current leader is suspected of being faulty.

The discussion of the protocol is organized into two parts: the first addresses normal operations wherein a leader has already been elected and is initiating transactions, followed by an examination of the leader election phase.

Part I: Normal Operations. In this section, we focus on the normal operational scenario when the system is initialized and there are no previously accepted messages. We will address the handling of previously accepted messages in the leader election phase.

Node n_l becomes the leader once it receives a sufficient number of promise messages. Client c sends its request message $m = \langle \text{REQUEST}, t, \tau, c \rangle$ with timestamp τ to execute transaction t. Transaction t in the context of this project is a transfer in the form of (c,c',amt). Timestamp

τ is used to ensure exactly-once semantics for the execution of client requests. Timestamps for requests of client c (assigned by client c) are totally ordered such that later requests have higher timestamps than earlier ones; for example, the timestamp could be the value of the client's local clock when the request is issued. All nodes maintain the last reply message they sent to each client and discard requests whose timestamp is lower than the client timestamp in the last reply they sent to the client to guarantee exactly-once semantics. The client waits to receive a reply from the leader. If the client does not receive a reply soon enough and its timer expires, the client broadcasts the request message to all nodes. If the request has already been processed, the nodes simply resend the reply. Otherwise, if the node is not the leader, it relays the request to the leader. Make sure to choose a reasonable timer duration for clients.

Node n_l initiates consensus by broadcasting accept messages of the form $\langle \text{ACCEPT}, b, s, m \rangle$. Here, b represents the ballot number, s is the sequence number (which starts at 1 during system initialization and increments subsequently), and m denotes the client request message received by the leader.

A backup node n_b will accept an accept message if the ballot number of the accept message is either equal to or greater than the highest ballot number that the node is aware of. Upon acceptance, it sends an accepted message formatted as $\langle \text{ACCEPTED}, b, s, m, n_b \rangle$ back to the leader.

The leader logs all received accepted messages. Once it collects $f+1$ accepted messages from different nodes (including itself), which constitutes a majority, for a request, it broadcasts a commit message of the form $\langle \text{COMMIT}, b, s, m \rangle$ to all nodes. The leader also executes the requested transaction if all requests with lower sequence numbers have already been executed. This ensures that all nodes execute requests in the same order as required to provide the safety property. Once the request is executed, the leader sends a reply message $\langle \text{REPLY}, b, \tau, c, r \rangle$ back to client c where b is the ballot number (enabling the client to identify the current leader), τ is the timestamp of the corresponding request and r is the result of executing the requested transaction. In our context, r can be modeled as a binary variable: `success` when the transaction is executed successfully or `failed` when the balance was not sufficient to process the transaction. Upon receiving a commit message from the leader, each backup node also executes the corresponding request if all requests with lower sequence numbers have already been executed on the node.

Part II: Leader Election. The leader election routine, similar to basic Paxos, includes prepare and promise messages. A proposer node n_p tries to become the leader if its timer expires. You can implement the timer in one of the two following ways:

1. A timer is initiated when the node receives a message from the leader and restarts when the node receives the next message from the leader. If the node does not receive any messages from the leader for t milliseconds, the timer expires.

2. A timer is initiated when the node receives a request, and the timer is not already running. The node stops the timer when it is no longer waiting to execute the request, but restarts it if, at that point, it is waiting to execute some other request.

You can choose either of the two strategies, and you need to choose a reasonable timer duration t.

Once a backup timer expires, the backup node attempts to assume the role of the leader by sending a message in the format of $\langle \text{PREPARE}, b \rangle$, where b represents its ballot number. To avoid multiple nodes initiating the leader election phase simultaneously, a node will only send a prepare message if it has not received any prepare messages in the last t_p milliseconds. It is important to determine the optimal value for t_p.

A node will accept a prepare message if its timer has already expired and the ballot number of the prepare message is greater than any ballot number previously received. If the timer has not yet expired, the node will log the prepare messages it receives and will accept the one with the highest ballot number once its timer does expire. Upon accepting a prepare message, the node responds with a promise message in the form of $\langle \text{ACK}, b, \text{AcceptLog} \rangle$ to the proposer. Unlike the Basic Paxos protocol, which incorporates AcceptNum and AcceptVal to indicate the latest accepted ballot number and its associated value, our approach includes an AcceptLog field. The AcceptLog field is an aggregation of triplets (AcceptNum, AcceptSeq, AcceptVal), where AcceptNum is the ballot number accepted by the node, AcceptSeq is the sequence number, and AcceptVal represents the corresponding client request. The AcceptLog set retains information about all requests accepted by the node since the system's initiation. We will suggest some enhancements to make this process more efficient.

To illustrate, let's consider a scenario where node n_2 becomes the leader with the ballot number $(1, 2)$ during the startup phase of the system. This node begins initiating transactions using the same ballot number but with unique sequence numbers, which are necessary in our protocol for differentiating between various transactions. An example sequence of accept messages might look like this (for clarity, we only display the contained transactions instead of entire client requests): ⟨ACCEPT, (1,2), 1, (A,B,10)⟩; ⟨ACCEPT, (1,2), 2, (C,E,2)⟩; ⟨ACCEPT, (1,2), 3, (D,B,4)⟩.

During this process, a backup node, such as n_1, may receive and store a subset of these transactions, potentially due to network instability. For example, let's say n_1 only captured the accept messages with sequence numbers 1 and 3. If the current leader, n_2, fails and n_1 then receives a prepare message $\langle \text{PREPARE}, (2, 3) \rangle$ from another node n_3, the backup node will aggregate its previously accepted requests into the AcceptLog set and send a promise message $\langle \text{ACK}, (2, 3), \text{AcceptLog} \rangle$ back to the proposer, where the AcceptLog is constructed as $\{((1, 2), 1, (A, B, 10)), ((1, 2), 3, (D, B, 4))\}$.

Once the proposer receives a quorum of $f + 1$ promise messages from different nodes (including itself), it creates a new-view message containing all requests that have been accepted by at least one node within the promise quorum. If a single client transaction has been accepted with multiple ballot numbers, the leader includes the one with the highest ballot number. Subsequently, the leader sends out the new-view message formatted as $\langle \text{NEW-VIEW}, b, \text{AcceptLog} \rangle$ to all nodes with b indicating the ballot number and AcceptLog being an ordered set of accept messages where each accept message, similar as before, includes (a) a ballot number, (b) a sequence number and (c) the respective client request; here (a) is the current leader ballot number and (b) and (c) are collected from the promise messages. This new-view message effectively condenses multiple accept messages (from the lowest sequence number (i.e., 1) to the highest sequence number observed in at least one promise message) into a single communication event. This integration is implemented to minimize the volume of message traffic, allowing us to process multiple requests in one consolidated accept phase rather than conducting separate accept phases for each individual request.

In cases where, for a given sequence number, the leader has not received any requests (potentially leading to gaps in the sequence), the leader will insert a no-op operation in place of the client request. The no-op operation will be executed through the protocol like any standard transaction, but it will not alter the state of the system once processed. This situation may arise when some accept messages from the previous leader are lost, resulting in a scenario where none, or at least none recognized by the promise quorum, have received the message for that particular sequence number.

For instance, let's assume that in response to the prepare message $\langle \text{PREPARE}, (2, 3) \rangle$, node n_3 receives the same promise message $\langle \text{ACK}, (2, 3), \{((1, 2), 1, (A, B, 10)), ((1, 2), 3, (D, B, 4))\} \rangle$ from the majority of nodes.

In this case, the corresponding new-view message will be $\langle \text{NEW-VIEW}, (2, 3), \text{AcceptLog} \rangle$ where the AcceptLog includes: $\langle$ACCEPT, (2,3), 1, (A,B,10)$\rangle$; $\langle$ACCEPT, (2,3), 2, no-op$\rangle$; and $\langle$ACCEPT, (2,3), 3, (D,B,4)$\rangle$.

Note that since the leader has not received any entry with sequence number 2 from the nodes, it inserts a no-op operation in place of the client request.

Backup nodes follow the normal operation of the protocol and send an accepted message to the leader for each accept message in the new-view message (even if they have already sent the accepted message in the previous view). The leader node then waits for accepted messages from a majority of nodes, commits and executes the request, sends a commit message to all backup nodes, and informs the client about the result. Similarly, the backup nodes follow the same routine by committing and executing each request. A node (the leader or a backup) might have already executed the request. In this case, they avoid re-executing client requests.

As before, the program should perform normal operations without any interruptions if a backup fails, as long as the number of faulty nodes remains at or below f. When a backup node recovers, it resumes processing new transactions by accepting accept messages and sending accepted messages back to the leader. However, for the backup nodes to execute new transactions, they must first receive the transactions that were previously committed (i.e., those that were processed while the node was faulty). Various protocols employ different strategies to facilitate this catch-up process. Some may actively request the missing transactions from a neighboring node or the leader, while others may passively wait for a new-view message (or a checkpoint message if the bonus part 1 is implemented) to synchronize their logs. Consider the potential mechanisms, along with their advantages and disadvantages, and choose one to implement in your project.

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. We suggest using RPC as it helps in understanding important concepts like data serialization, marshalling, and inter-process communication over RPCs. However, the use of TCP/UDP sockets is also completely acceptable.

Your implementation should support 5 nodes ($2f + 1$ nodes where $f = 2$) and 10 clients. Nodes know each other and all clients (during the system initialization step). Clients can also communicate with all nodes. A client sends a request to what it believes is the current leader (reply messages include the current ballot number, allowing the client to track the current leader). If the node that receives a request message is not the leader, the node simply transmits the request to the leader. The clients should be able to resend the requests to the system (all nodes) if they have not received the reply on time (i.e., when the client timer expires). A leader processes requests *out-of-order*, which means it does not need to wait for the accepted messages of the previous requests before sending the accept message for the next request.

Your program should first read a given input file. This file will consist of a set of triplets (s, r, amt) where s is the client who initiates the transaction. Assume all clients start with 10 units. Then the clients initiate requests, e.g., request messages containing transactions like (A,B,4), (D,A,2), (C,B,1), ...

- 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 PrintLog function which prints the log of a given node, displaying each request's metadata. This log represents all messages that the node has processed or is processing, helping to track the progress of each request through the protocol stages.
- Your program should have a PrintDB function which prints the current datastore. The database can be represented as a key-value store that maintains the balance of all clients.
- Your program should have a PrintStatus function that takes a sequence number as its parameter and outputs the status of the transaction associated with that sequence number at *each* node. The status labels should be as follows:
  - `A`: Accepted (when the node receives a valid accept message from the leader)
  - `C`: Committed
  - `E`: Executed
  - `X`: No Status
- 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.
- 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 need any front-end UI for this project. Your project will be run on the terminal.

Here are some notes on client implementation:

- You can implement all 10 clients as a centralized entity (e.g., a single process), but you must still manage 10 separate timers and any other necessary variables.
- Clients do not need to retry a request that has already been processed with a "failed" reply due to insufficient balance. They should only resubmit a request if they do not receive any reply before their timer expires.
- Each client can wait for the response to its previous request before sending the next one (closed-loop clients); however, different clients can submit their requests in parallel, as is common in real-world scenarios. Note that the leader imposes a total ordering on all requests, typically based on their arrival time, where requests that arrive first are assigned lower sequence numbers.
- Clients have no idea of what nodes are alive and what nodes are faulty.
- In the beginning, when a client sends its first message, it communicates with node n_1 since it does not know who the leader is. This implies that during the system's startup, the first client directs its request to node n_1, which then starts the leader election process.

3 Bonus!

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

1. **Checkpointing.** One challenge with the current design is that when a leader fails and a new one is elected, all transactions must be processed from the beginning, which can be costly in large-scale systems. To address this, we can implement checkpointing, where nodes periodically save the system's state and use these checkpoints instead of processing all previous requests. The current leader can generate checkpoint messages and send them to all nodes, for example, after every 100 committed requests (checkpointing period). Each checkpoint message is formatted as $\langle \text{CHECKPOINT}, n, d \rangle$, where n is the sequence number of the last request whose execution is reflected in the state and d is the digest of the state (datastore). Message digests are produced by collision-resistant hash functions, which transform any input data into a unique, fixed-size output. Examples of common collision-resistant hash functions include the SHA-2 family (SHA-256, SHA-512) and the SHA-3 family (SHA3-256, SHA3-512), which generate digests of varying bit lengths.

   Note that you can simply send the actual state, which includes the complete list of clients and their balances, instead of just the digest. While this approach may be effective for our project, given the limited number of clients, it can lead to considerable communication overhead in real-world applications.

   When a node receives a checkpoint message with sequence number n, it discards all new-view, prepare, promise, accept, accepted, and commit messages with sequence numbers

$\leq n$ from its log, as well as all previous checkpoints and checkpoint messages. During the leader election phase, nodes will include the sequence number n of the latest checkpoint message in their AcceptLog parameter for promise messages. Similarly, the leader includes the sequence number n of the highest checkpoint message it received in its promise quorum and an accept message for all requests with higher sequence numbers in its new-view message. When a node receives a new-view message, it may be missing some request message or a checkpoint (since these are not sent in new-view messages). It can obtain missing information from another node.

2. **Benchmarking.** Testing the performance and functionality of distributed systems is challenging. To be able to evaluate distributed systems and compare their performance against each other, we need to use standard benchmarks. Such benchmarks are specifications and program suites for evaluating systems. For distributed databases, different benchmarks have been proposed, such as Yahoo! Cloud Serving Benchmark (YCSB) or TPC-C (short for Transaction Processing Performance Council Benchmark C). Another benchmark that is close to what we have implemented is SmallBank, which simulates a banking application. This workload models a banking application where transactions perform simple read and update operations on their checking and savings accounts. All of the transactions involve a small number of tuples. The transactions' access patterns are skewed such that a small number of accounts receive most of the requests. It contains three tables and six types of transactions. The user table contains users' personal information, the savings table contains the balances, and the checking table contains the checking balances. If you are interested in evaluating the performance of your system in a real deployment, you can implement the SmallBank benchmark and evaluate your system under that. The details of the SmallBank benchmark can be found in [1].

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.

2. **Select Your Student ID**: On the next page, select your Student ID from the list.

   Important: If your Student ID is not listed, do not click "Skip." Instead, 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/cft-<YourGithubUsername>.git
$ cd cft-<YourGithubUsername>
```

This will create a directory named `cft-<YourGithubUsername>` under your home directory, which will serve as the Git repository for all your lab assignments throughout the semester. 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 1, 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.

NOTE: Please do not make any changes to the workflows in the GitHub directory. These workflows are designed to handle your submissions, and tampering with them would compromise your submission.

5 Deadline, Demo, and Deployment

This project will be due on October 11. You are supposed to give a short demo for each project on October 17. For this project's demo, you can deploy your code on several machines. However, it is also acceptable if you use different processes on the same machine to simulate the distributed environment.

6 Tips and Policies

6.1 General Tips

- This is a difficult project! Start early!
- Read and understand the Paxos paper [2] and the Paxos lecture notes before you start.

6.2 Implementation

- You are allowed to use any programming language that you are more comfortable with.
- 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.

6.3 Possible Test Cases

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

- Receiving transactions from clients and initiating leader election (on system startup)
- Basic agreement between nodes resulting in committing and executing transactions
- Failure of a backup node
- Failure of the current leader, resulting in initiating leader election
- Failure of the leader right before sending the commit message and its impact on the next consensus instance
- Concurrently started the leader election phase by different nodes
- Expired client timer
- No agreement if too many nodes fail (disconnect)

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 nodes 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 nodes should remain idle until prompted to process the next set. You are not allowed to terminate your nodes after executing a set of transactions, as consecutive test cases may be interdependent.

You should be able to fail the leader whenever requested (before executing a set of transactions).

After executing one set of transactions, when all the nodes are idle and waiting to process the next set, your implementation should allow the use of functions (such as printDB, printLog, 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 three 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 Nodes**: A list of nodes that are active and available for all transactions in the corresponding set.

Set Number	Transactions	Live Nodes
1	(A, C, 5)	[n_1, n_2, n_3, n_4, n_5]
	(C, E, 4)	
	(B, D, 2)	
	(E, A, 10)	
	(E, C, 3)	
2	(A, E, 4)	[n_1, n_3, n_5]
	(C, A, 1)	
	(A, C, 7)	

Table 1: Example Test Input File

An example of the test input file is shown below:
Explanation:

- The first set contains five transactions `(A, C, 5)`, `(C, E, 4)`, `(B, D, 2)`, `(E, A, 10)`, and `(E, C, 3)`. These transactions are processed with all nodes active, as indicated by the list [n_1, n_2, n_3, n_4, n_5].
- The second set (Set Number 2) contains three transactions `(A, E, 4)`, `(C, A, 1)`, and `(A, C, 7)`. For these transactions, only nodes [n_1, n_3, n_5] are active, meaning nodes n_2 and n_4 are **disconnected**.

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. 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 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 3 (December 07) 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.

- There is no distinction between those who provide the source and those who copy it.

- 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 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.

References

[1] Michael J Cahill, Uwe Röhm, and Alan D Fekete. Serializable isolation for snapshot databases. *Transactions on Database Systems (TODS)*, 34(4):1–42, 2009.

[2] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.