CSE 535: Distributed Systems
Lab 1
Due: October 17, 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. As part of the project, you will create a basic (but not entirely realistic) distributed banking application using a group of servers. Each server will handle outgoing transactions for a specific account (client) locally and keep track of them in its own log. In cases where an account lacks sufficient funds for a transaction, a modified Paxos protocol will be initiated among all servers to retrieve the latest transactions. Once consensus is reached, each server will add a block of transactions to its datastore.

1 Project Description

We will present the project in two phases. First, we will outline a basic (Strawman) design, followed by a discussion of its challenges. This will lead to the introduction of the actual project that you are expected to implement. The purpose of explaining the Strawman design is to simplify understanding of the project.

1.1 Basic Design

We illustrate the Strawman design using an example. Let's consider a bank with *three* servers that keep track of all transactions made by clients. The bank uses Paxos as its underlying consensus protocol to keep an updated transaction history stored in a simple datastore (which can be implemented in any form) on all servers to ensure proper replication, and at the same time, to ensure fault-tolerance from server crash failures.

The goal is to implement a simple banking application where each client has a single account. Clients can send requests to transfer money from their accounts to other accounts. A transfer transaction (S, R, amt) initiated by some client c consists of a sender, S, a receiver, R, and an amount of money amt. The transaction is valid if c is the owner of the sender account and the account balance is at least amt.

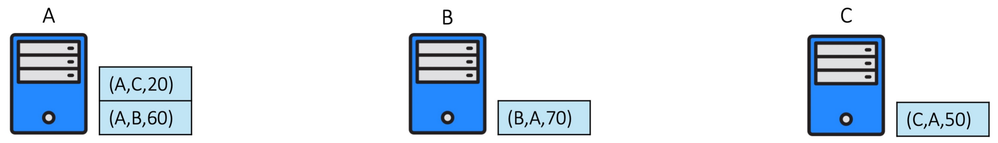

Figure 1: Basic Design: First Snapshot

Consider a very simple scenario with three servers A, B, and C and three corresponding clients A, B, and C where each server is responsible for processing the outgoing transactions of its corresponding client. Each client initially has 100 units in its account (maintained by the corresponding server). Figure 1 presents the log of each server after a few transactions. As can be seen, client A has initiated two transactions (A,C,20) and (A,B,60). Client B has initiated a single transaction (B,A,70) and client C has initiated a single transaction (C,A,50). At this point, all transactions have been executed locally and the servers are not aware of the transactions that are executed on other servers.

Now, let's assume client A initiates a transaction (A,B,30). Server A checks the client's account and realizes that client A has only 20 units in its account. Therefore, it cannot execute the transaction locally. Server A, thus, sends a *synchronization request* $\langle \textsc{cync}, A\rangle$ to all other servers (i.e., B and C). When a server (e.g., B) receives a synchronization request, it checks the balance of the requester server (i.e., A) on its log. If the balance is positive, the server initiates a modified Paxos protocol to share its logged transactions (as the consensus value) with other servers. As a result of establishing consensus (which is explained in more detail later), all servers get all executed transactions on another server and appends a block of transactions to their datastores.

For example, as shown in Figure 2, server B initiates consensus and shares all transactions in its log with others. As a result of that transaction block B_1 has been appended to the datastore of all servers. Note that once transactions are added to the datastore, server B removes such transactions from its local log.

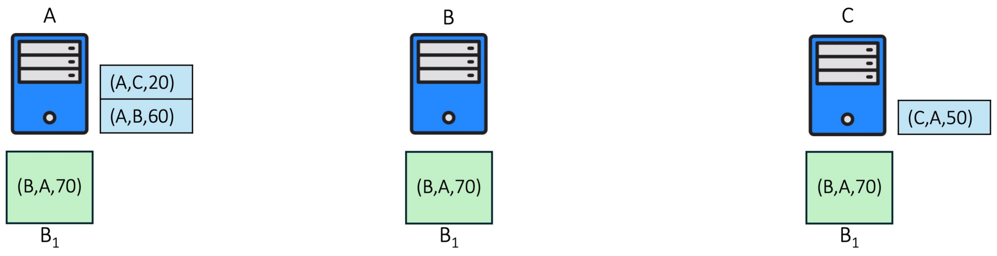

Figure 2: Basic Design: Second Snapshot

Now, the balance of client A (on server A) is 90 and A can easily initiate transaction (A,B,30). Similarly, clients B and C also initiate transactions (B,C,20) and (C,B,10) which are executed locally on servers B and C respectively. Figure 3 shows a snapshot of the system.

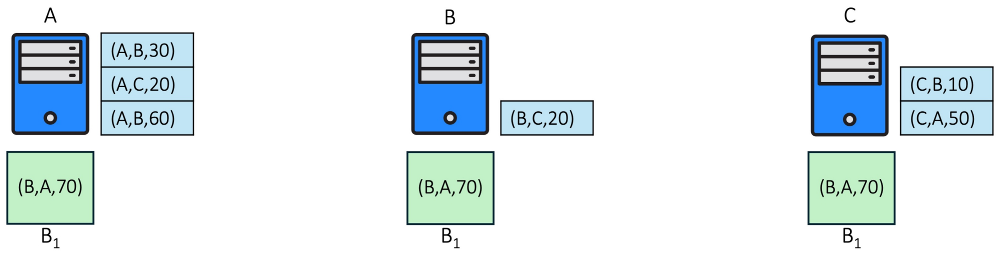

Figure 3: Basic Design: Third Snapshot

At this point, client B sends transaction (B,C,30), however, its balance is 10 and server B cannot process the transaction. As a result, server B sends a synchronization request $\langle \textsc{cync}, B\rangle$ to all other servers (i.e., A and C). Let's assume server A initiates the consensus protocol earlier and becomes the leader. Server A shares its local log with all other servers.

As shown in 4, as a result of consensus, a block of transactions will be added to the datastore and server B can also execute the requested transaction.

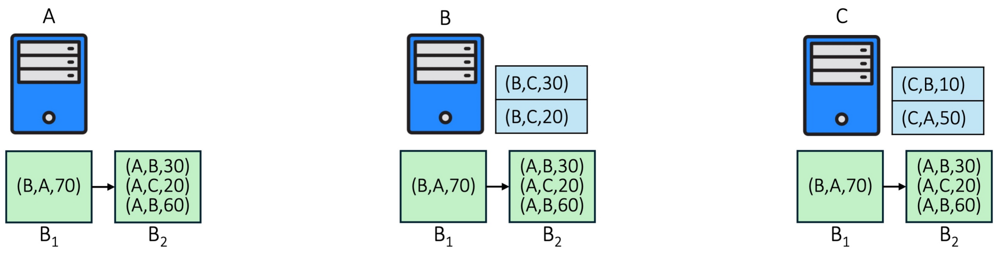

Figure 4: Basic Design: Fourth Snapshot

1.1.1 Paxos Protocol

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.

In the first part, we will discuss the leader election of the Paxos protocol. In the second part, will discuss the normal operations of the Paxos protocol along with the datastore to store transaction information. Finally, in the last part, you will deal with the leader failure of Paxos protocol while at the same time, supporting normal operations. In this example, we will assume f, the maximum number of possible failures that can be tolerated, as 1.

Part I: Leader Election: Following the Paxos leader election routine (prepare and promise messages), there will be *exactly one leader*, and *two followers* after the leader election stage among the three servers. Server S tries to become the leader only if it receives a synchronization request from another server S′ and the balance of the corresponding client to S′ on server S is positive, i.e., there is at least on transaction (S,S′,amt) in the log of S (where the receiver is S′). Server S initiates leader election by sending a $\langle \text{PREPARE}, n \rangle$ where n is its ballot number. When a server receives the prepare with a ballot number higher than any previously received ballot number, the server sends an $\langle \text{ACK}, n, AcceptNum, AcceptVal \rangle$ message to the sender promising not to accept any ballots smaller than n in the future. The promise message includes AcceptNum and AcceptVal to inform the leader about the latest accepted value (set of transactions) and what ballot it was accepted in.

Part II: Normal Operations: Once a server S becomes a leader, it multicasts an accept message $\langle \text{ACCEPT}, n, T(S) \rangle$ to all other servers where n is the ballot number and $T(S)$ is a list of the transactions that were executed locally on server S but have not been appended to the datastore. Note that if the leader receives any value in promise messages, it proposes that value instead of propping its own value (set of transactions).

Upon receiving a valid accept message from the leader, each server S′ sends an accepted message $\langle \text{ACCEPTED}, n, T(S) \rangle$ to the leader.

The leader logs all accepted messages. Once the leader receives f accepted messages from different servers (plus itself becomes $f + 1$, a majority of the servers), it multicasts a commit message $\langle \text{COMMIT}, n, T(S) \rangle$ to *all* servers. The leader then appends the block of transactions

$T(S)$ to its datastore and executes the requested transaction. The leader also needs to remove transactions in $T(S)$ from its log. It might not be straightforward especially if the server has received new transactions in between. Upon receiving a commit message from the leader, each server appends block $T(S)$ to the datastore.

Part III: Node Failure: A failed server might be a follower (non-leader) or a leader. If a follower fails, the program should still be able to perform normal operations. In case of leader failure, Paxos should still be able to perform normal operations 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 a set of transactions has been proposed and accepted (but might not decided due to leader failure). Note that once a server fails, its copy of the datastore might become out-of-date.

1.1.2 Challenges

Although the proposed solution can handle transactions, run consensus, and manage potential failures, it still faces two significant challenges. Firstly, when a client lacks sufficient balance to process transactions, the corresponding server has to issue synchronization requests to other servers and wait for them to initiate consensus. This can lead to higher latency, especially in scenarios where synchronization requests are lost or multiple servers send requests simultaneously. Secondly, every time consensus is run, servers become aware of transactions processed on only a single server. This greatly hampers system performance as servers need to consistently run consensus in order to process their local transactions. For example, in a deployment with a large number of servers (e.g., 100), each time a server lacks sufficient balance, an instance of the consensus protocol needs to be run among all servers. Consequently, running consensus makes servers aware of local transactions from only one server at a time. We try to address these challenges in the next design.

1.2 Main Design

To explain the main design (the one that you are supposed to implement), we continue with our simple scenario with three servers A, B, and C and three corresponding clients A, B, and C. As before, each server is responsible for processing the outgoing transactions of its corresponding client and each client initially has 100 units in its account. Figure 5 presents the log of each server after a few transactions (the same set of transactions as the basic design). Client A has initiated two transactions (A,C,20) and (A,B,60). Client B has initiated a single transaction (B,A,70) and client C has initiated a single transaction (C,A,50). At this point, the servers are not aware of the transactions that are received on other servers.

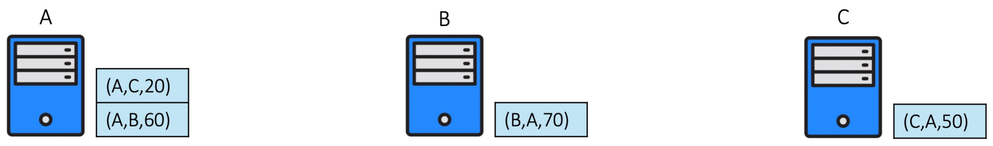

Figure 5: Final Design: First Snapshot (Same as the first snapshot of the basic design)

Again, client A initiates a transaction (A,B,30). Server A checks the client's account and realizes that client A has only 20 units in its account. Therefore, it cannot process the transaction locally. In the main design, in contrast to the basic design, Server A initiates a modified Paxos protocol itself to get the most up-to-date information from servers B, and C. Once consensus is established, each server gets transactions on (at least the majority of) all servers and as shown in Figure 6, appends a block of transactions to its datastore.

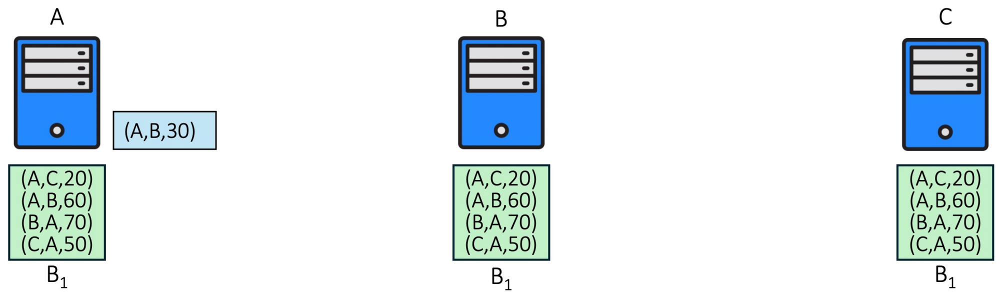

Figure 6: Final Design: Second Snapshot

Now, the balance of client A is 140 and A can easily initiate transaction (A,B,30). Similarly, clients B and C also initiate transactions (B,C,50) and (C,B,60) which are processed locally on servers B and C respectively, as shown in Figure 7.

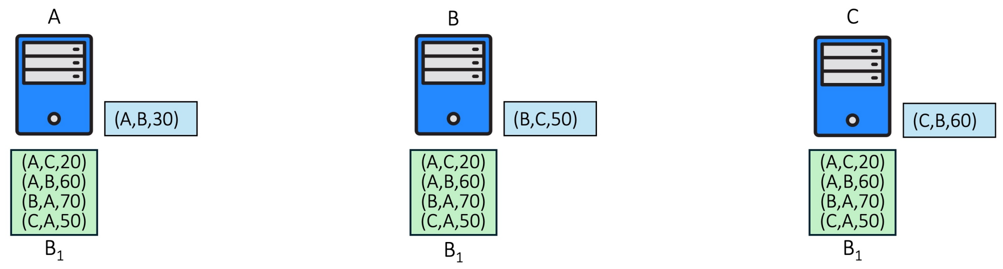

Figure 7: Final Design: Third Snapshot

At this point client B sends transaction (B,A,60), however its balance is 40 and server B cannot process the transaction. As a result, server B will initiate a consensus protocol among all servers to get the most recent list of transactions. As shown in 8, as a result of consensus, a block of transactions will be added to the datastore and server B can also execute the requested transaction.

1.2.1 Modified Paxos protocol

In order to accommodate the new design, we will need to make adjustments to the Paxos protocol. In essence, in Paxos, the leader suggests a value, which can either be an accepted value from a previous round that has not yet been decided, or a value owned by the leader. For instance, in the original design, the leader proposes its own local log (a series of transactions) as the value. However, in our new design, we want the leader to propose a value that has been collected from other servers (i.e., sets of local transactions). The primary question

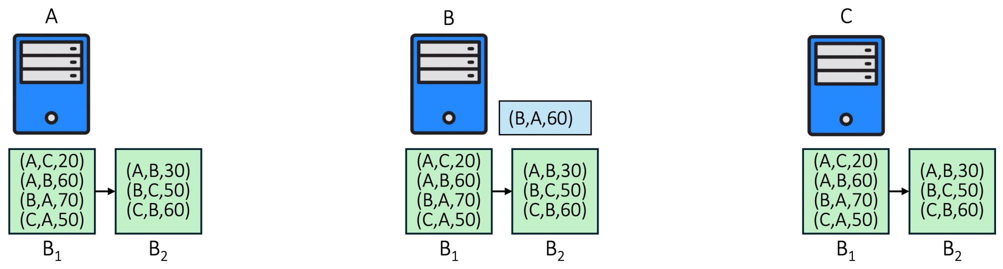

Figure 8: Final Design: Fourth Snapshot

now is how to modify the Paxos protocol so that we can collect these transactions without being susceptible to safety issues when failures occur.

One possible way is to add one more phase of communication between the leader (proposer) and all other servers. In this phase, the leader can collect all their local logs and generate a block of transactions (let's call it a major block) that can be later used as the consensus value. While this design does not have any correctness issues, it might not be the most efficient one. This is because we add one phase of communication and then we probably need to think about possible failures in this phase.

Another possible approach is to change one of the Paxos phases in order to enable the leader to collect local logs. There are two phases that can be changed: the prepare (i.e., the leader election) phase or the accept phase. Let's start with changing the accept phase and assuming that servers send their local logs to the leader as part of their accepted messages. The leader then collects local logs, generates a block of transactions and sends the block back to the servers in its commit message.

The protocol works perfectly in normal situations. However, what happens if the leader fails right before sending the commit messages? In Paxos and in such a situation, the next leader is supposed to propose the value that has been accepted but not decided (using AcceptNum and AcceptVal). However, is any server aware of the value in such a situation? The value in our scenario is basically a collection of all local logs but the leader has collected those logs and probably no one else is aware of that. Hence, this design does not work under some failure scenarios.

The only remaining option (and the approach that your implementation needs to follow) is to change the leader election phase in such a way that the leader can collect local logs of other servers.

2 Implementation Details

2.1 Modified Paxos Protocol

Now that we have a grasp of the difference between the original Paxos protocol and the modified Paxos protocol (that we will use in this project), it is time to revisit different phases of the protocol.

Part I: Leader Election: As discussed in the previous section, we need to change the leader election phase so that the leader is aware of the local logs of the servers involved in the consensus. A server S can contest for leader election only if it has a transaction (S,S',amt) and the balance of client S is less than amt. Now, server S needs to start consensus with other servers to retrieve their local transactions in order to process the transaction. Server S initiates the leader election by sending $\langle \text{PREPARE}, n \rangle$ to all servers (note that the prepare message structure might need more elements, as discussed in Section 2.2, point 5). Upon receiving a prepare message from server S, each server S' sends a promise message $\langle \text{ACK}, n, AcceptNum, AcceptVal, T(S') \rangle$ to S where $T(S')$ is a list of transactions that were processed locally on server S' but have not been appended to the datastore. You can think of two scenarios here. (1) if there is a value (i.e., a set of transactions) that has been proposed and accepted (but might not decided due to leader failure) in the previous phase, server S' uses AcceptNum and AcceptVal to inform the leader about the latest accepted value (set of transactions) and what ballot it was accepted in (and $T(S')$ would be empty). (2) if there is no such a value, server S' uses $T(S')$ to inform server S about the list of transactions that were processed locally on server S' but have not been appended to the datastore.

All other details remain similar to the Paxos protocol.

Part II: Normal Operations: The server contesting leader election (here, server S) logs all promise messages. Once it receives f promise messages from different servers (plus itself becomes $f + 1$, a majority of the servers), it either proposes an accepted but not decided value from previous rounds or it creates a major transaction block MB combining its own set of local transactions and all the transaction lists received in promise messages. The leader then multicasts a accept message $\langle \text{ACCEPT}, n, MB \rangle$ to *all* servers. Note that the leader might need to synchronize slow servers that send AcceptVal values that have already been decided (committed). This point will be discussed in Section 2.2.

Upon receiving a *valid* accept message from the leader, each server S' sends an accepted message $\langle \text{ACCEPTED}, n, MB \rangle$ to the leader which is followed by a decide phase of Paxos to persist the changes permanently. Every server must also delete its local transactions that are part of the main block MB from its log. As the server may have recorded new transactions (between sending its log to the leader in a promise message and receiving a commit message from the leader), it needs to carefully examine transactions in its log and remove only the outdated ones. One simple method for accomplishing this task is to assign an order (such as timestamp or sequence number) to local transactions when adding them to the log. Figure 9 demonstrates the normal case execution of Paxos protocol with 3 servers.

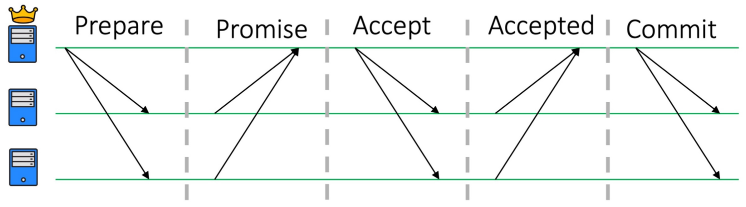

Figure 9: Normal Case execution of Paxos protocol

Part III: Node Failure: This part mainly remains the same as the original Paxos protocol. Note that you should be able to synchronize servers that recovered from failures with other servers, as discussed in Section 2.2.

2.2 Design and Implementation Thoughts

1. **Catch-up mechanism.** After recovering from a failure, a server's datastore may have outdated information compared to other servers that have processed new transaction blocks. As a result, the server must synchronize its state with others in order to process incoming transactions. There are various methods for implementing this synchronization mechanism. One active approach involves the server contacting the leader or another server to obtain new transaction blocks by sharing its current datastore state (e.g., the number of the last block committed on the server). On the other hand, a passive approach entails the server waiting for a prepare message from a leader and notifying the leader that its state is outdated, after which the leader attempts to synchronize the server's datastore. This process is somewhat similar to what leader nodes do in the Raft protocol. Including both the current ballot number and the ballot number of the last committed block in the prepare message will resolve many of the challenges. You could choose any preferred method for implementing the catch-up mechanism or even come up with your alternative correct approach.

2. **Data loss.** If a server crashes and its log is not stored persistently, any client transactions within the log will be lost. While it's not necessary for this project to store logs on persistent storage (although we encourage you to do so), it is important to ensure that servers do not send reply messages to clients if the client transaction has not been stored in persistent storage. This precaution is necessary because if the server sends a reply to the client while the transaction is in the log and then subsequently fails, the client transaction may never be executed despite being told by the server that it was successful. To simplify, servers are not required to send reply messages to clients at all.

3. **Quorum construction.** In Paxos, the proposer typically waits for promise messages from a majority of servers, including itself, before sending accept messages. However, in our project's context, this means that the proposer does not collect the transaction log of every server. One potential improvement is for the proposer to wait for promise messages from all servers before sending the accept messages. However, if some servers are crashed or very slow, or if the network is unstable, this approach may result in long wait times for the leader.

   To address this issue, one proposal is that the leader should wait for a designated amount of time to receive promise messages. When the leader receives promise messages from majority of servers, it starts a timer and once its timer expires, it can construct a major block using all received transaction logs (at least from the majority) and then send out the accept messages. It's important to note that setting a large timer can indeed slow down the protocol. This is because the proposer will need to wait for a longer period before it can proceed with constructing the major block and sending out the accept messages. Therefore, it's crucial to strike a balance between allowing sufficient time for receiving promise messages from more servers and avoiding excessive delays in the protocol execution. A careful consideration of various factors such as network stability and server responsiveness should inform the selection of an appropriate timeout value.

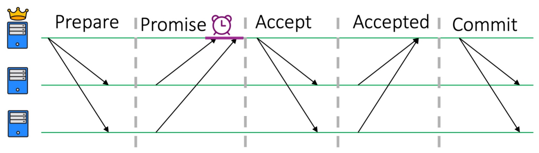

Figure 10: Paxos with a slow server

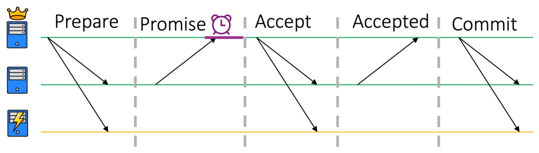

Figure 11: Paxos with a crashed server

In Figures 10 and 11, two scenarios are depicted. In one, the promise message is received from a slow server before the timer expires, while in the other scenario, the timer expires without the leader receiving promise messages from all servers and consequently contracting with a quorum of two servers. It is important to note that for the leader to proceed to the next steps, it must wait for messages from a majority of servers.

4. **Proposing AccpetVal.** In Paxos, the primary objective is to reach consensus on a single value. However, our project's goal involves reaching consensus on a sequence of values (i.e., a sequence of major blocks), which may pose challenges within the framework of traditional Paxos design. Specifically, in Paxos, when a leader receives an AcceptVal in *promise* messages, it simply uses the value with the highest AcceptNum in its accept message, even if it was received from just one server. While this mechanism is essential for Paxos protocol correctness, in our context where a sequence of values is being decided upon instead of just one, it could lead to significant latency issues. For instance, a slow server with an unstable network might continue sending AcceptVal values that have already been decided by other servers. In such cases, the leader would need to repeatedly propose these values before getting an opportunity to propose its own value.

   While the catch-up mechanism can partially mitigate this issue, it may still pose challenges when dealing with slow (non-faulty) servers. One potential strategy for the leader is to form a quorum consisting of servers that have empty AcceptVal values in their promise messages. However, this approach is only feasible if the leader receives such messages from at least a majority of servers (i.e., in case of failures the leader is required to include slow servers in its quorum).

   Alternatively, the leader can check whether it has already committed the transaction block included in the AcceptVal or any later transaction blocks. If yes, the leader will forward the corresponding commit message(s) to such servers to synchronize them. Otherwise, the leader will propose the AcceptVal value. As mentioned before, including both the current ballot number and the ballot number of the last committed block in the prepare message can help to detect slow servers and synchronize them. This approach could address latency issues and aid in maintaining consistency across all participating servers.

   Figure 12 demonstrates a scenario with 5 servers with ids 1 to 5 where a leader needs to synchronize other servers. For simplicity, lost messages are not shown.

   As demonstrated, server 1 tries to become the leader by sending a prepare message including its ballot number <1,1> to all other servers. Once server 1 receives promise messages from a majority of servers (2, 3, and 1 itself), it sends an accept message

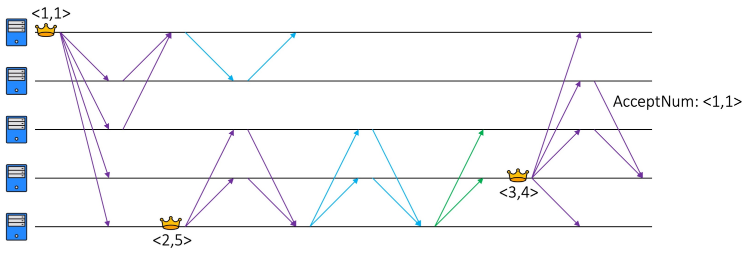

Figure 12: The need for synchronizing other servers

including the major block B to all servers. Note that since it is the first instance of consensus running between servers, the AcceptVal values of all received promise messages are equal to $\perp$. Due to network issues, only server 2 receives the accept message from server 1. Server 2 then updates its AcceptNum to <1,1>, AcceptVal to v_1, and sends an accepted message to server 1.

Simultaneously, server 5 attempts to become the leader by sending a prepare message with ballot number <2,5> to all other servers. While servers 1 and 2 were not able to receive any messages from server 5 (due to an unstable network), server 5 commits its major block B' with an agreement (promise and then accept messages) from a majority of servers (3, 4, and 5). Note that the AcceptVal value for all these three servers was $\perp$ (because none of them received the accept message sent by server 1).

Next, server 4 tries to become the leader by sending a prepare message with ballot number <3,4> to all servers. Server 2 and server 3 send promise messages to server 4. However, server 2 includes its AcceptNum (i.e.,<1,1>), AcceptVal (i.e., B) in the message. Now, it is the task of server 4 (the proposer) to inform server 1 about the past committed blocks (by sending the commit message that it has received from server 5 in the previous round) and synchronize the server.

Servers also need to reset their AcceptNum and AcceptVal values. A server sets these values as soon as it receives a valid accept message from the leader (and sends an accepted message). It can safely reset these values simply after committing the corresponding major block or receiving messages from the leader regarding the subsequent committed blocks.

5. **Datastore consistency.** This project, unlike Paxos, requires handling a sequence of values. Consequently, we need a mechanism to make sure that all servers have committed all transaction blocks and there is no gap in the datastore of different servers.

   An an example, consider the scenario presented in Figure 13. Server 1 is able to commit a major block B with ballot number <1,1> by agreement from a majority of servers (i.e., 1, 2, and 3). Once the block is committed, servers 1, 2 and 3 will reset their AcceptNum and AcceptVal values.

   Next, server 5 becomes the leader and successfully commits another block of transactions B' with ballot number <2,5>. This time servers 3, 4 and 5 where involved in consensus.

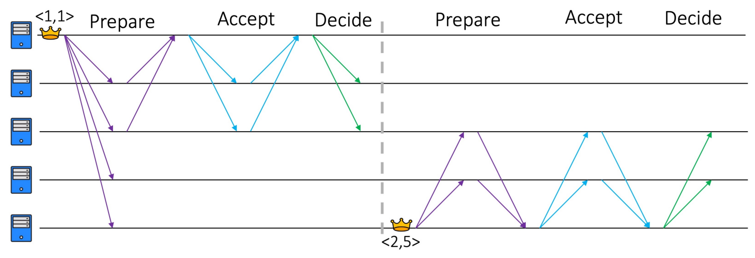

Figure 13: The need for resetting AcceptVal and AcceptNum

Now, if we look at the datastore on different servers, server 1 and 2 maintain a single block B, server 3 maintains two blocks B and B' and servers 4 and 5 maintain a single block B', demonstrating an inconsistency in datastores.

To resolve this issue, servers need to keep each other informed about their last committed blocks. One potential approach is for the leader to include the ballot number of its last committed block in its prepare message. Other servers will only accept a prepare message if the ballot number of the leader's last committed block is greater than or equal to their own last committed block's ballot number. This mechanism, along with the catch-up mechanism, should effectively address any issues related to failed or slow servers.

6. **Log consistency.** When a server receives a commit message from the leader, it needs to remove the corresponding transactions from its log. This involves checking which segments of its log have been incorporated into the proposed major block. It is crucial to conduct this check because (a) the server may have added new transactions to its log while consensus was being reached, and (b) in scenarios involving parallel proposers, the server might have sent different sets of transactions to different proposers. To address these challenges, incorporating unique sequence numbers into both the local log and final block can help maintain consistency and accurately identify which transactions should be removed. You should also be able to handle situations where a server sends some transactions to the leader in its promise message and fails before receiving the corresponding commit messages. In this situation and if the log has not been stored persistently, the server might receive a major block including its own local transactions while such transactions are not part of its current log. Another relevant scenario arises when a server sends transactions to the leader in its promise message but does not receive corresponding accept or commit messages, even though the leader has committed those transactions. In such cases, the server may forward these transactions from its log to the next leader. The leader should be capable of informing the server about previously committed messages (as discussed in the previous section on datastore consistency). As a result, the server will update its datastore and log to prevent duplicate entries.

   The leader can choose to either wait for a new promise message from the server or include only the new portion of the server's log in its proposed block by verifying the sequence number of transactions.

   Figure 14 demonstrates an example of such a scenario. Server 1 is acting as the leader

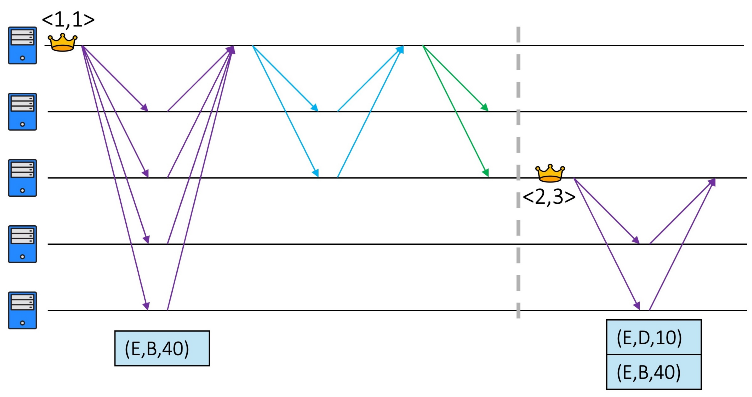

Figure 14: The need for updating log upon receiving commit messages for previous blocks

and includes all servers' transactions, including the single transaction (E,B,40) from server 5, in its proposed block. However, server 5 does not receive the corresponding accept and commit messages. Next, server 3 attempts to become the leader by sending a prepare message with the ballot number <2,3>. In response, server 5 sends a promise message to server 3 because the ballot number is greater than the last ballot number that it has seen (<1,1>). Server 5 also includes its logged transactions in the message. Notably, this includes both transactions ((E,B,40) and (E,D,10)), even though transaction (E,B,40) has already been committed on servers 1, 2 and 3. To resolve this issue, the leader must inform server 5 about past committed blocks in its prepare messages, as discussed in the previous part, in order to prevent duplicate entries.

In this example, if server 4, which is not aware of the last committed block, attempts to become the leader instead of server 3, the catch-up mechanism should be capable of updating the datastore of server 4. This means that proposer servers must include the ballot number of their last committed block, in addition to their current ballot number, in their prepare messages.

7. **Server implementation.** Servers can be implemented using processes, coroutines, or threads. Processes are a better way of implementing servers as they are independent and can simulate a distributed environment more naturally. Communication between servers 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.

2.3 Prescribed Conditions

Your implementation should support 5 clients and 5 servers where each server is responsible for processing the transactions that a single client initiates.

Each client sends its requests to the corresponding server. Each request is a transfer (S,R,amt) from the account of sender client S to receiver client R. There will be two situations: (1) If the balance on the client account is at least amt, the server logs the request and processes the transfer locally. (2) Otherwise (when account balance < amt), the server initiates a modified Paxos protocol among servers.

The database can be represented as a linked list where each block B consists of a *sequence number* and a *list of transactions.*

Your program should first read a given input file. This file will consist of a set of triplets (S,R,amt). Assume all clients start with 10 units. Then the clients initiate transactions, e.g., (A,B,4), (D,E,2), (C,B,1), ... Each client request needs to be processed by the corresponding server.

- 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 prints the balance of a given client on the corresponding server (considering both the datastore and local log of the server).
- Your program should have a PrintLog function which prints the local log of a given server.
- Your program should have a PrintDB function which prints the current datastore.
- Your program should have a Performance function which prints throughput and latency.
- 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. Implement a modified Multi-Paxos protocol! In Multi-Paxos, the leader is only changed when it is suspected to be faulty. After the initial leader election, servers can rely on the elected leader to perform next instances of balance synchronization. Whenever a client has insufficient balance, instead of running Paxos itself, the corresponding server sends a synchronization request to the elected leader. The leader then runs an instance of Multi-Paxos among servers. This approach could potentially reduce latency since servers do not need to run leader election phase of Paxos every time.
2. Currently, there are no constraints on how to implement the datastore. We recommend you use a standard database to store transaction blocks. Think about possible database choices and use the ones with minimum overhead (you don't need an Oracle DBMS to handle such a simple application).

3. Implement a function to retrieve the current balance of a specified client across different servers, taking into account both the datastore and local logs, all without requiring consensus among them. The function should effectively read the client's balance on each server and then return the aggregated total. You need to think about how to maintain the balance of each client (store it or calculate it every time).

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:F24-CSE535/apaxos-<YourGithubUsername>.git
$ cd apaxos-<YourGithubUsername>
```

This will create a directory named `apaxos-<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 lab1**. Afterward, visit the provided 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 them would compromise your submission.

5 Deadline, Demo, and Deployment

This project will be due on October 17. We will have a short demo for each project (the date will be announced later). For this project's demo, you can deploy your code on several machines. However, it is also acceptable if you just use several processes in 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 Made Simple** paper 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. We will assess the performance of all submissions against each other. Projects are ranked according to their performance, and those in the lower quarter of the ranking may face point deductions (loss of performance points).

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.

- Receiving transactions from clients and appending them to the local log
- Initiating leader election upon receiving a client request without having sufficient balance
- Basic agreement between servers resulting in appending a block of transactions to the datastore of each server and updating the log of all (involved) servers
- Failure of the leader right before sending the decide message and its impact on the next consensus instance
- Concurrently started agreements
- No agreement if too many servers 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 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 server 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 *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 Servers**: A list of servers 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 Servers
1	(S1, S3, amt1)	[S1, S2, S3, S4, S5]
	(S3, S5, amt2)	
	(S2, S4, amt3)	
	(S5, S1, amt4)	
	(S5, S3, amt5)	
2	(S1, S5, amt6)	[S1, S3, S5]
	(S3, S1, amt7)	
	(S1, S3, amt8)	

Table 1: Example Test Input File

Explanation:

- The first set (Set Number 1) contains five transactions `(S1, S3, amt1)`, `(S3, S5, amt2)`, `(S2, S4, amt3)`, `(S5, S1, amt4)`, and `(S5, S3, amt5)`. These transactions are processed with all servers active, as indicated by the list `[S1, S2, S3, S4, S5]`.
- The second set (Set Number 2) contains three transactions `(S1, S5, amt6)`, `(S3, S1, amt7)`, and `(S1, S3, amt8)`. For these transactions, only servers `[S1, S3, S5]` are active, meaning servers S2 and S4 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. 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.