CSE 535: Distributed Systems
Project 2
Due: November 9, 2025 (11:59 pm)

Abstract

The goal of the second project is to implement a variant of the Practical Byzantine Fault Tolerance (PBFT) consensus protocol. PBFT is designed to achieve consensus in a distributed system, requiring a minimum of $3f + 1$ nodes, where f represents the maximum number of concurrently faulty (Byzantine) nodes that the system can tolerate without compromising its integrity. In this project, you will focus on implementing two key components of the linear variant of the PBFT protocol: the normal case operation and the view-change routine. The normal case operation in linear-PBFT is distinct from traditional PBFT as it achieves consensus with linear communication complexity, making it more efficient in terms of the overhead associated with message passing between nodes. On the other hand, the view-change routine in linear-PBFT remains the same as the original PBFT protocol. As in the previous project, we will utilize a basic distributed banking application to facilitate the implementation and testing of this protocol variant.

1 Project Description

We first explain the application, followed by a brief overview of the PBFT protocol and the linear PBFT protocol. We then discuss different types of requests and malicious behaviors that the system needs to support.

1.1 Banking Application

In this project, similar to the first project, you will deploy a simple banking application that allows clients to submit their requests. This time, we support two types of requests: (a) balance (read-only) transactions (s), where the system 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). Both clients and replicas can be faulty, so all messages need to be checked to ensure validity. For simplicity, we do not implement any access control or blocking mechanism to restrict clients' malicious behavior. However, all messages need to be signed. Figure 1 shows the workflow of the system to process transfer transactions. The outgoing line from the client to the primary replica is the request message, and all incoming lines to the client show the reply messages.

1.2 PBFT Protocol

PBFT, as discussed in the class, is a *leader-based* consensus protocol. To perform this project, you need to be fully aware of all details of the PBFT protocol [2]. Please carefully read the

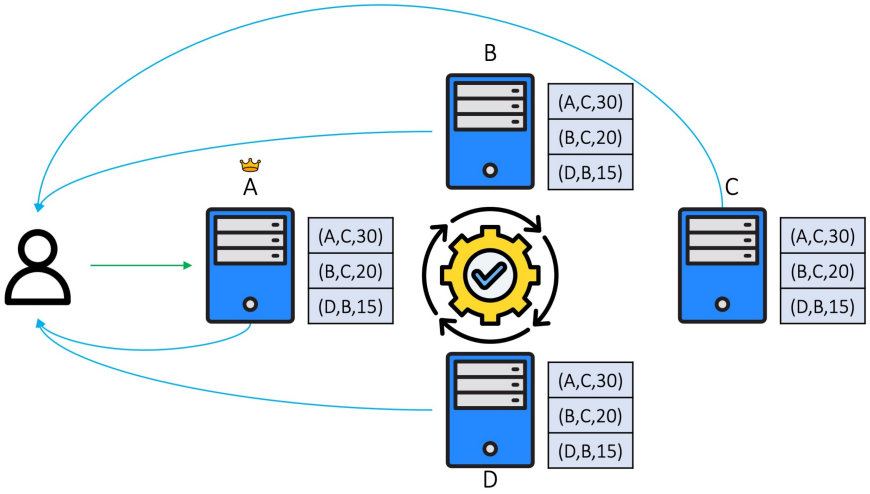

Figure 1: System overview

paper before starting the project. Here, we briefly give a high-level overview of the normal operation and view-change routine of the protocol.

PBFT assumes the partial synchrony model. In PBFT, while there is no upper bound on the number of faulty clients, the maximum number of concurrent malicious replicas is assumed to be f. Replicas are connected via an unreliable network that might drop, corrupt, or delay messages, and the network uses point-to-point bi-directional communication channels to connect replicas. In PBFT, a strong adversary can coordinate malicious replicas and delay communication. However, the adversary cannot subvert cryptographic assumptions.

PBFT, as shown in Figure 2, operates in a succession of configurations called *views*. Each view is coordinated by a *stable* leader (primary). In PBFT, the number of replicas, n, is assumed to be $3f + 1$ and the ordering stage consists of pre-prepare, prepare, and commit phases. The pre-prepare phase assigns an order to the request, the prepare phase guarantees the uniqueness of the assigned order and the commit phase guarantees that the next leader can safely assign the order.

During a normal case execution of PBFT, clients send their signed request messages to the leader. In the pre-prepare phase, the leader assigns a sequence number to the request to determine the execution order of the request and multicasts a pre-prepare message to all *backups*. Upon receiving a valid pre-prepare message from the leader, each backup replica multicasts a prepare message to all replicas and waits for prepare messages from $2f$ different replicas (including the replica itself) that match the pre-prepare message. The goal of the prepare phase is to guarantee safety within the view, i.e., $2f$ replicas received matching pre-prepare messages from the leader replica and agree with the order of the request.

Each replica then multicasts a commit message to all replicas. Once a replica receives $2f + 1$ valid commit messages from different replicas, including itself, that match the pre-prepare message, it commits the request. The goal of the commit phase is to ensure safety across views, i.e., the request has been replicated on a majority of non-faulty replicas and can be recovered after (leader) failures. The second and third phases of PBFT follow the *clique* topology, i.e., have $O(n^2)$ message complexity. If the replica has executed all requests with lower sequence numbers, it executes the request and sends a reply to the client. The

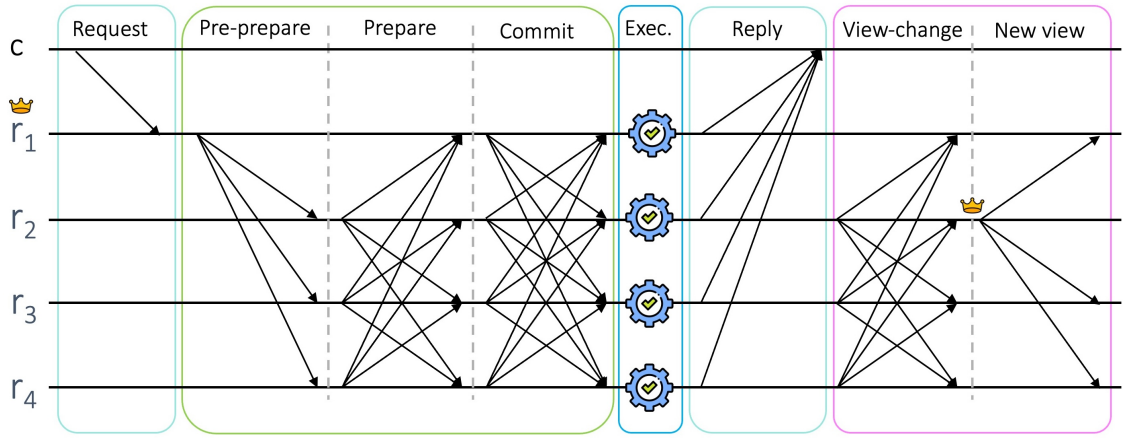

Figure 2: Different stages of PBFT protocol

client waits for $f+1$ matching results from different replicas.

In the view change stage, upon detecting the failure of the leader of view v (being suspicious that the leader is faulty) using execution timers, backups exchange view-change messages including their latest stable checkpoints and the later requests that the replicas have *prepared*. After receiving $2f + 1$ view-change messages, the designated stable leader of view $v + 1$ (the replica with $\texttt{ID} = v + 1 \bmod n$) proposes a new-view message, including a pre-prepare message for each request that should be processed in the new view. Since checkpointing is not part of the project (it is a bonus), you do not need to include any checkpointing certificate in the view-change messages, and the min-s in the new-view message is equal to 0.

In PBFT, replicas periodically generate checkpoint messages and send them to all replicas. If a replica receives $2f + 1$ matching checkpoint messages, the checkpoint is stable. PBFT uses either signatures [2] or MACs [3] for authentication. Using MACs, replicas need to send view-change-ack messages to the leader after receiving view-change messages. Since new-view messages are not signed, these view-change-ack messages enable replicas to verify the authenticity of new-view messages.

1.3 Linear PBFT Protocol

While PBFT is the gold standard for BFT protocol design, it suffers from its quadratic communication complexity, where in the prepare and commit phases, all replicas need to communicate with each other. This causes a significant overhead, especially in cases where the protocol is deployed on a large network (i.e., a large number of replicas). One way to reduce this high communication complexity is to linearize the protocol by splitting each all-to-all communication phase into two linear phases: one phase from all replicas to a collector (typically the leader) and one phase from the collector to all replicas. The output protocol requires signatures for authentication. In each phase, the collector collects a quorum of $n - f$ *signed messages* (either prepare or commit) from replicas and broadcasts these messages (within a single message) to all replicas. Figure 3 presents different stages of the linear-PBFT protocol.

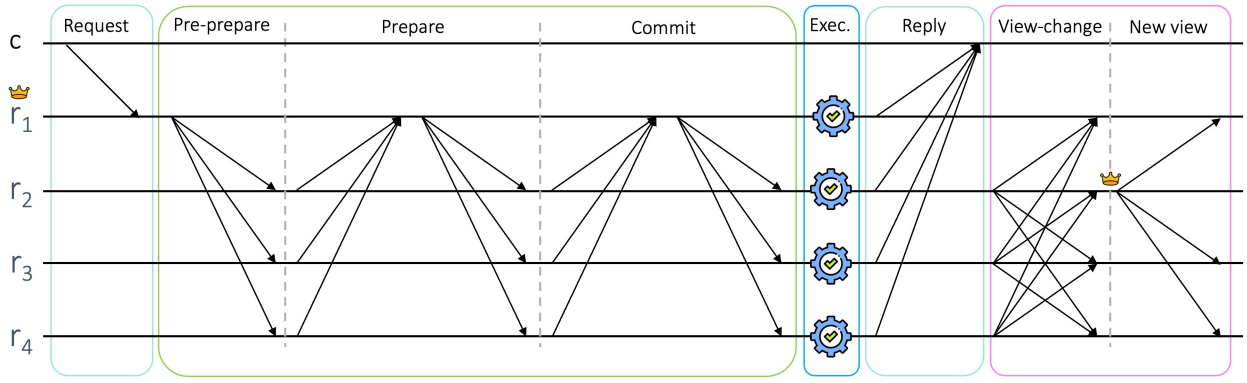

Figure 3: Different stages of Linear PBFT protocol

1.4 Request Types

Your system is supposed to support both read-write and read-only requests. Read-write requests, e.g., transfers in our context, are processed using the normal operation of the protocol, as discussed earlier, and update the state once executed. Read-only requests, e.g., reading the balance in our context, on the other hand, do not modify the state and hence can be processed more easily. A client multicasts a read-only request to all replicas. Replicas execute the request immediately in their state after checking that the request is properly authenticated and send the reply back to clients. The client waits for $2f + 1$ replies from different replicas with the same result. The client may be unable to collect $2f+1$ such replies if there are concurrent writes to data that affect the result; in this case, it retransmits the request to the leader as a regular read-write request after its timer expires.

1.5 Byzantine Failures

To make our project more realistic, we plan to implement several types of malicious failures.

1. **Invalid signature.** Malicious replicas do not sign their messages properly, resulting in validation errors on the recipient's end.

2. **Crash.** When a malicious replica serves as the leader, it (a) refrains from changing its status to prepared and neglects to send prepare messages to other replicas, despite meeting the necessary requirements (e.g., receiving and logging $2f$ valid prepare messages), (b) does not send a new-view message during a view change while still recording any received view-change messages, and (c) does not send reply messages back to clients (in case of read-only requests). When acting as a backup, the malicious replica (a) fails to send any prepare messages or update its status to prepared, even if it receives (and logs) valid prepare messages from the leader, and (b) does not send reply messages back to clients (in case of read-only requests).

3. **In-dark.** Malicious replicas intentionally avoid sending messages to certain honest replicas.

4. **Timing.** A malicious leader intentionally delays sending any messages for t_a ms (deliberately postponing its communications). The duration t_a is a fraction of the execution timers and must be selected carefully to avoid causing timer expiration.

5. **Equivocation.** The malicious leader disseminates conflicting pre-prepare messages by assigning two distinct (consecutive) sequence numbers to the same client request and sending that to two different subsets of backups.

2 Implementation Details

In this project, you are required to implement the linear-PBFT protocol. As discussed earlier, most details of the PBFT protocol remain the same, other than the communication pattern, which becomes linear. You can rely on the leader or any other node to play the role of collector. Note that the view-change routine of the protocol remains unchanged compared to PBFT.

As before, 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.

While the PBFT paper uses UDP and its explanation is more consistent with TCP/UDP communication style, using RPC helps in understanding important concepts like data serialization, marshaling, and inter-process communication over RPCs.

Your implementation should support 7 nodes ($3f + 1$ 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 view, 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 prepare or commit messages of the previous requests before sending the pre-prepare 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.

- All clients start with 10 units.

- 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.
- 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:
  - `PP`: Pre-prepared (when the leader sends or a backup receives a valid pre-prepare message from the leader)
  - `P`: Prepared
  - `C`: Committed
  - `E`: Executed
  - `X`: No Status
- Your program should include a PrintView function that outputs all new-view messages (including all its parameters, e.g., received view-change messages) exchanged since the start of the test case. If the system has undergone 3 view changes during the test case, the PrintView function should display all 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.
- Your implementation must reset the system state before processing the next set of transactions, clearing all data maintained on both the nodes and clients. Each transaction set is processed in isolation, with no retained state from previous sets (refer to section 6.4).
- We do not want 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 waits 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 honest 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, and n_1 is the leader of view 1.

3 Bonus!

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

1. **Checkpointing mechanism.** The checkpointing mechanism of PBFT is used to first, garbage-collect data of completed consensus instances to save space, and second, restore in-dark replicas (due to network unreliability or malicious leader) to ensure all non-faulty replicas are up-to-date. Checkpointing is typically initiated after a fixed window (e.g., every 100 request) in a decentralized manner without relying on a leader. If you choose to implement checkpointing, you need to include the checkpointing certificate in view-change messages. However, you should not discard the data of completed consensus instances.

2. **Threshold Signature.** The Threshold Signature Scheme (TSS) is a digital signature scheme in which a threshold must be met before a transaction can be authorized. The threshold refers to the number of key shareholders who can sign on behalf of the entire group. The general rule or access structure of TSS is often referred to as "t of n" (e.g., $2f + 1$ out of $3f + 1$). Using threshold signatures, the collector message size becomes constant, e.g., instead of sending $2f + 1$ commit messages to all backups to let them know the message has been committed, it sends a single commit message signed by $2f + 1$ nodes using a threshold signature.

3. **Optimistic phase reduction.** Given a *linear* BFT protocol, you can optimistically eliminate two linear phases (i.e., the equivalence of a single quadratic prepare phase) assuming all replicas are non-faulty, e.g., SBFT [4]. The leader (collector) waits for signed messages from all $3f + 1$ replicas in the second phase of ordering (prepare), combines signatures and sends a signed message to all replicas. Upon receiving the signed message from the leader, each replica ensures that all $3f+1$ replicas have received the request and agreed with the order. As a result, the third phase of communication can be omitted, and replicas can directly commit the request. If the leader has not received $3f + 1$ messages after a predefined time, the protocol falls back to its slow path and runs the third phase of ordering (commit).

4. **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.

   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/bft-<YourGithubUsername>.git
$ cd bft-<YourGithubUsername>
```

This will create a directory named `bft-<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 Lab 2, 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.

NOTE: Please note that, unlike the previous assignment, where submission mistakes like incorrect commit messages or workflow file modifications were waived without penalties, this lab will not allow any exceptions. Follow all instructions carefully, as no waivers will be granted for submission errors. The verification link provided is there to help you confirm if you have submitted correctly.

5 Deadline, Demo, and Deployment

This project will be due on November 9. We will have a short demo for each project on November 14. 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.

6 Tips and Policies

6.1 General Tips

- Again, similar to your first project, this is a difficult project! Start early!
- Read and understand the PBFT paper [2] and the PBFT 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 (test cases) that your system should be able to handle.

- Client communication in both Request and Reply phases
- Basic PBFT agreement between nodes resulting in committing a client transaction
- Failure of a backup node, including different malicious behaviors
- Failure of the leader node, and probably initiating the view-change routine
- Consecutive view-change routines
- No agreement if too many nodes are faulty (disconnected or Byzantine)

6.4 Test Case Format

The testing process involves a CSV file (.csv) as the test input containing a set of transactions along with the corresponding live nodes and Byzantine 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.

After executing one set of transactions, your implementation should allow the use of functions from section 2.2 (such as printDB, printStatus, etc.). The implementation will be evaluated based on the output of these functions after each set of transactions is processed.

In contrast to the previous assignment, where the system's state was preserved after executing a set of transactions, the current 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.

The test input file will contain four columns:

1. **Set Number**: Set number corresponding to a set of transactions.
2. **Transactions**: A list of individual transactions, each on a separate row, in one of the two formats: (Sender, Receiver, Amount) or (Sender).
3. **Live**: A list of nodes that are active and available for all transactions in the corresponding set (including Byzantine nodes).
4. **Byzantine**: A list of nodes that exhibit Byzantine behavior for all transactions in the corresponding set.
5. **Attack**: Type of the attack performed by the Byzantine nodes. Each set might include more than one type of attack separated by ';'. The format is as follows:
   - sign: all malicious nodes perform invalid signature attack,
   - crash: all malicious nodes perform crash attack,
   - dark(n_i, n_j): all malicious nodes perform in-dark attack on nodes n_i and n_j. The number of target nodes could be between 1 to all non-Byzantine live nodes,
   - time: the malicious leader performs a timing attack, and
   - equivocation(n_i, n_j): the malicious leader performs an equivocation attack by sending n_i and n_j client request m with sequence number n while sending other nodes request m with sequence number $n + 1$.

An example of the test input file is shown below:

Explanation:

- The first set includes five transactions: `(A, C, 1)`, `(C, E, 2)`, `(B, F, 3)`, `(G)`, and `(J, H, 5)`. In transaction `(A, C, 1)`, client `A` sends 1 unit to client `C`, while in transaction `(G)`, client `G` reads its balance. All nodes are active for processing these transactions, as indicated by the list `[`n_1`, `n_2`, `n_3`, `n_4`, `n_5`, `n_6`, `n_7`]`, with nodes `[`n_4`, `n_6`]` being Byzantine. The Byzantine nodes perform a crash attack: they do not send any prepare messages or update their status to prepared, even if they receive valid prepare messages from the leader.

Set Number	Transactions	Live	Byzantine	Attack
1	(A, C, 1)	[n_1, n_2, n_3, n_4, n_5, n_6, n_7]	[n_4, n_6]	[crash]
	(C, E, 2)			
	(B, F, 3)			
	(G)			
	(J, H, 5)			
2	(A, E, 6)	[n_1, n_2, n_3, n_5, n_6, n_7]	[n_1, n_3]	[time; dark(n_2)]
	(C)			
	(D, I, 8)			

Table 1: Example Test Input File

- The second set contains three transactions `(A, E, 6)`, `(C)`, and `(D, I, 8)`. For these transactions, only nodes [n_1, n_2, n_3, n_5, n_6, n_7] are active, meaning node n_4 is disconnected and nodes n_1 (the leader) and n_3 exhibit malicious behavior. The leader n_1 performs a timing attack, and both n_1 and n_3 perform an in-dark attack on node n_2.

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) + 20% bonus. 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 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 PBFT 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.

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] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186. USENIX Association, 1999.

[3] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[4] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable decentralized trust infrastructure for blockchains. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 568–580. IEEE/IFIP, 2019.