CSE 535: Distributed Systems
Lab 2
Due: November 7, 2024 (11:59 pm)

Abstract

The objective of the second project is to implement a variant of the PBFT consensus protocol. PBFT can achieve consensus with at least $3f + 1$ nodes where f is the maximum number of concurrent faulty (Byzantine) nodes. You are supposed to implement the normal case operation and the view-change routine of a BFT protocol called linear-PBFT. In contrast to PBFT, the normal case operation of linear-PBFT achieves consensus with linear communication complexity. The view-change routine, however, is the same as PBFT. Similar to the previous project, we use a basic distributed banking application. However, all client requests are processed by all nodes.

1 Project Description

We discuss the project in three steps. We first, explain the application followed by a brief overview of the PBFT protocol. Finally, the linear PBFT protocol is discussed.

1.1 Banking Application

In this project, similar to the first project, you are supposed to deploy a simple banking application where clients send their transfer transactions in the form of (S, R, amt) to the servers where S is the sender, R is the receiver, and amt is the amount of money to transfer. Since no single server is trusted, in contrast to project 1, all transactions need to be maintained by all servers, i.e., consensus is needed for every single transaction. Both clients and servers 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 overview of the system. The outgoing line form the client to the primary server 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 [1]. Please carefully read the 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.

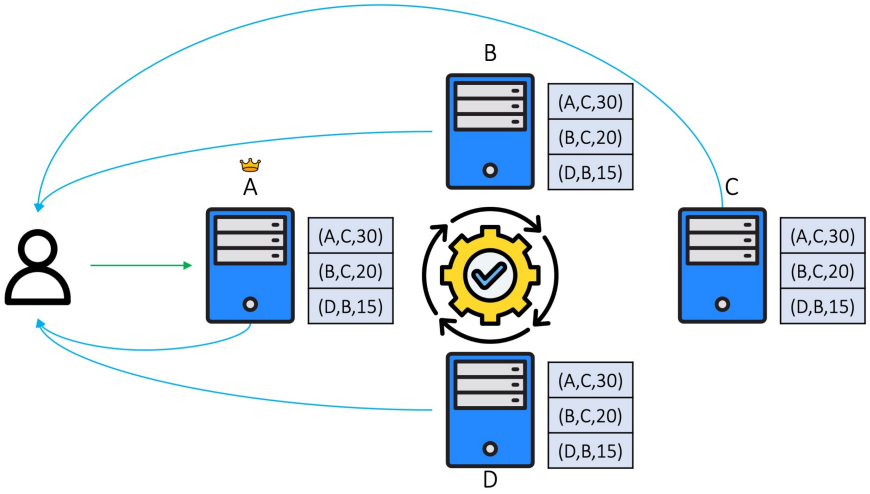

Figure 1: System overview

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 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 timeouts, 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.

In PBFT, replicas periodically generate checkpoint messages and send them to all replicas.

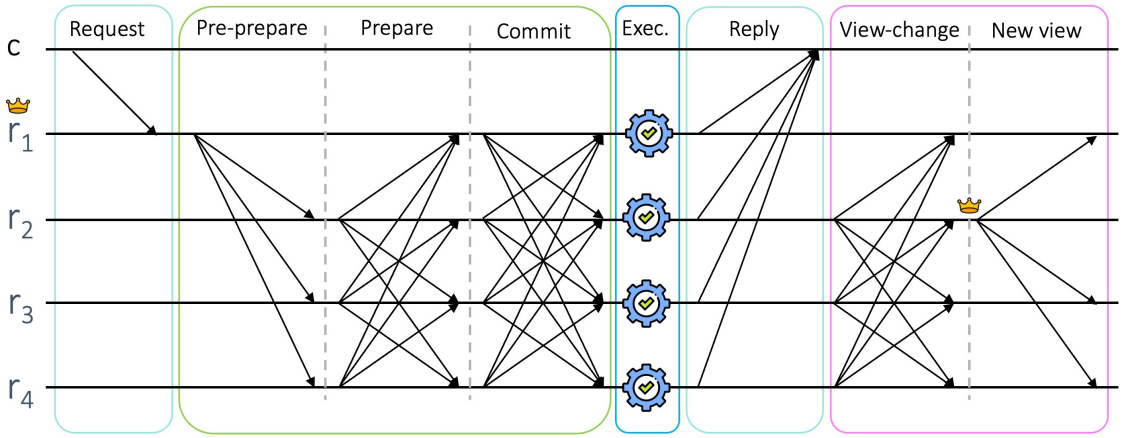

Figure 2: Different stages of PBFT protocol

If a replica receives $2f + 1$ matching checkpoint messages, the checkpoint is stable. PBFT uses either signatures [1] or MACs [2] 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 nodes 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 nodes). 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. The collector collects a quorum of $n - f$ *signatures* from replicas and broadcasts its message including the signatures, as a certificate of having received the required signatures. Figure 3 presents different stages of the linear PBFT protocol.

2 Implementation Details

In this project, you are supposed to implement the linear-BFT 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, 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

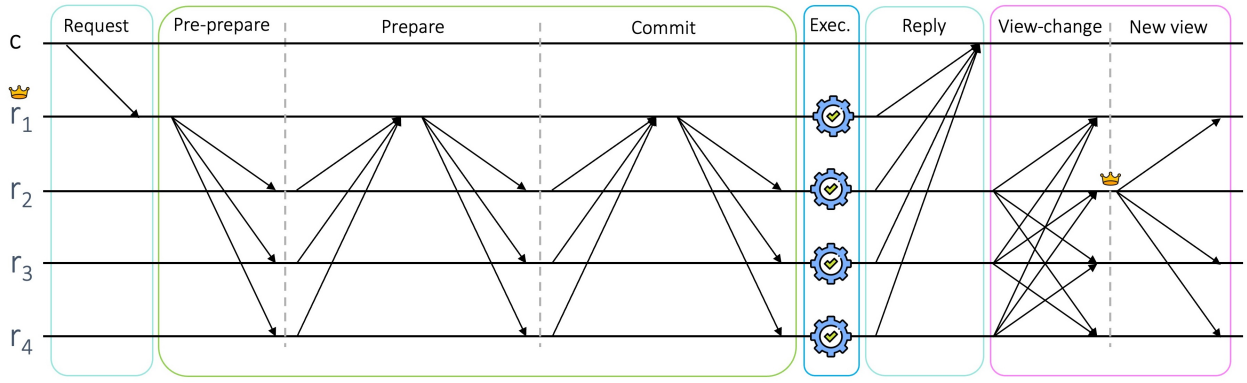

Figure 3: Different stages of Linear PBFT protocol

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. Using RPC 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.

2.1 Prescribed Conditions

Your implementation should support 7 servers ($3f + 1$ servers where $f = 2$) where clients communicate with the server they assumed to be the leader. As explained in PBFT, if the node that receives a request 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.

Your implementation should be able to support 10 clients. Each client sends its requests to the primary server. Each request is a transfer (S,R,amt) from account S to R.

The database can be represented as a key-value store that keeps the record of all clients.

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,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 PrintBalance function which prints the balance of a given client.
- Your program should have a PrintLog function which prints the local log of a given server including all exchanged messages.
- 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. **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 leader maliciousness) 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.
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-share holders 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.
3. **Optimistic phase reduction.** Given a *linear* BFT protocol, you can optimistically eliminates two linear phases (i.e., the equivalence of a single quadratic prepare pahse) assuming all replicas are non-faulty, e.g., SBFT [3]. The leader (collector) waits for signed messages from all $3f + 1$ replicas in the second phase of ordering, combines signatures and sends a signed message to all replicas. Upon receiving the signed message from the leader, each replica ensures that all non-faulty 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 fallbacks to its slow path and runs the third phase of ordering.

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 2, please include an explicit commit with the message **submit lab2**. 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 November 7. 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

- Again, similar to your first project, this is a difficult project! Start early!
- Read and understand the PBFT paper [1] 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 servers resulting in committing a client transaction
- Failure of a follower nodes
- Failure of the leader node and initiating view-change when $2f + 1$ nodes send view-change message
- 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.

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.

References

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

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

[3] 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.