CS 171 : Introduction to Distributed Systems

Final Project

Final Demos: Monday June 12th & Tuesday June 13th, 2023

1 Overview

In this project, you will be building a simple distributed blog post application that is replicated on 5 servers/nodes to ensure fault tolerance. Each server will have its own storage of blog users and their corresponding blog posts as well as a replicated log of all the blog operations in the form of a blockchain. You will be using Multi-Paxos as the protocol for reaching agreement on the next block to be appended to this replicated blockchain.

None of the nodes are assumed to be malicious, but some might crash, also known as a *fail-stop* failure. The network may also exhibit failures. In particular, the network might partition.

2 Blog Post Application

The blog post application consists of a blog data structure for storage and a blockchain that acts as the log for blog operations. The application's main features allow users to make blog posts, comment on other users' posts, and view posts and comments.

2.1 Replicated Blog

The blog is a data structure for storing all blog users, and associating them with the blog posts and comments they make.

Every node will maintain its own copy of the blog. These local copies should be kept consistent with each other (i.e. replicated) through a concensus protocol, which will be covered in more details in Section 3. Implementation for this data structure is flexible as long as it is able to support the following *writing* operations:

- **Make a new post**: given a username, a title, and the content, create a new blog post authored under the username with the title and the content
- **Comment on a post**: given a username, a title, and a comment, if a blog post with the given title exists, create a new comment authored under the username for that post. Otherwise, the creation of the comment fails.

And the following *reading* operations:

- **View all posts**: list the title and the author for all blog posts in chronological order
- **View all posts made by a user**: given a username, list the title and content of all blog posts *made by this user* in chronological order

- **View all comments on a post**: given the title of a blog post, get its content, and list all the comments on the post and their authors

2.2 Replicated Blockchain

Instead of a ledger for transactions, this blockchain acts as a log for all *writing* operations that have been applied to the blog. *Reading* operations should be applied to the blog immediately and do *not* need to be replicated. Similar to previous projects, the blockchain should be implemented as a chain of blocks connected via hash pointers. A block consists of 3 fields:

1. A hash pointer (H)
2. Writing operation details (T) in the clear, i.e. not hashed or compressed.
3. Nonce (N)

Hash pointer is the hash value of the previous block, i.e.

$$\text{Hash}(\text{H}_{Prev} \| \text{T}_{Prev} \| \text{N}_{Prev})$$

where H_{Prev}, T_{Prev}, and N_{Prev} refer to the hash value, operation, and nonce from the **previous** block. It should be stored in the block as a hexdigest.

Operation refers to the **post** and **comment** operations $\langle OP, username, title, content\rangle$:

- **Post**: OP is `POST`, *username* is the username of the author of the post, *title* is the title of the post being authored, and *content* is the content of the post
- **Comment**: OP is `COMMENT`, *username* is the username of the author of the comment, *title* is the title of the post being commented on, and *content* is the content of the comment

Nonce, associated with the current block, is a value $\text{N}_{Current}$ such that

$$\text{Hash}(\text{H}_{Current} \| \text{T}_{Current} \| \text{N}_{Current})$$

where $\text{H}_{Current}$ and $\text{T}_{Current}$ refer to the current block, will have at least THREE leading zero bits, i.e. the output of the hash function should be "000" followed by the rest of the 253 bits.

3 Consensus Protocol

In this project, you will be implementing Multi-Paxos as the consensus protocol for replicating the blog and the blockchain across 5 servers/nodes.

3.1 Nodes

Every node is connected to all other nodes and directly communicates with them. Each node should keep track of the current *leader* of the system. If an *acceptor* receives an operation on its command line interface, it should forward the operation to the *leader*. If this *acceptor* doesn't receive a decision on the operation from the *leader* after a reasonable period of time, **it should time out** and become a *proposer* to propose the operation. If an *acceptor* receives an operation and doesn't know who the *leader* is, it should also become a *proposer*. If another *acceptor* receives a forwarded operation, it should forward the operation to its known *leader*

3.2 Multi-Paxos

Multi-Paxos is an extension of Paxos where an existing *leader* who have already decided on a single log entry will stay as the *leader* for subsequent entries instead of starting a leader election for every new entry.

A `ballot_number` in Multi-Paxos should be a 3-tuple: $\langle seq\_num, pid, depth \rangle$, where *depth* captures the length/depth of the blockchain at the *proposer*/*leader*. The flow for your implementation of the protocol should look similar to the following:

1. **Election Phase**: A node intending to become the leader becomes a *proposer* and broadcasts `PREPARE` messages with its `ballot_number` to all *acceptors*.
2. A *proposer* must obtain a majority of `PROMISE` messages from *acceptors* for its `ballot_number` to become the *leader*
3. An *acceptor* should NOT reply `PROMISE` to a `PREPARE` if the *acceptor*'s blockchain is deeper than the *depth* from the `ballot_number`
4. **Replication/Normal Phase**: Once a node becomes the *leader*, it should maintain a queue of pending operations waiting for consensus from the system
5. The *leader* needs to compute a passing nonce for the next block in the blockchain based on the next pending operation from its queue
6. The *leader* then broadcasts `ACCEPT` messages with the next pending operation as a block to all *acceptors*
7. An *acceptor* should NOT reply `ACCEPTED` to an `ACCEPT` if the *acceptor*'s blockchain is deeper than the *depth* from the `ballot_number`
8. **Decision Phase**: After a majority of *acceptors* reply `ACCEPTED` to the *leader*, the *leader* will:
    - Append the proposed block to its blockchain
    - Apply the corresponding operation to its blog
    - Remove the operation from its queue

    - Broadcast `DECIDE` messages to all nodes

9. Upon receiving the `DECIDE` message from the *leader*, an *acceptor* will similarly append the proposed block to its blockchain, and apply the corresponding operation to its local blog

10. The *leader* should start a new normal/replication phase for the next pending operation in its queue with an updated `ballot_number`

4 Failures and Recovery

Your system should handle crash failures and network failures. It should still be able to make progress as long as a majority of the nodes are alive and connected. A crashed node should be able to restart, restore to its previous state from disk, and reconnect to the system. A partitioned node should be able to reconnect with the rest of the system and resume normal operation. A leader failure should be handled as described in the Paxos lectures.

4.1 Restore from Disk

To ensure data persistence in the presence of crash failures, you should periodically save a node's blog and blockchain to a file on disk. When a crashed node restarts, it should read from this file and restore its blog and blockchain to the state before it crashed.

If a node is the *leader*, it should write a proposed block to file once all *acceptors* have accepted.

If a node is an *acceptor* that receives a proposed block from the *leader*, it should initially write the block to file and tag the block as *tentative*. Once it receives the `DECIDE` message on a block from the *leader*, it then changes the tag to *decided*. Once a block is *decided* on disk, its operation should be applied to the blog in memory. Any changes to the in-memory blog should be reflected in the on-disk blog.

4.2 Extra Credit (10%): Repair from Other Nodes

A node that is reconnecting with the system will be missing any blocks that were replicated while the node was down or partitioned from the system. You need to implement a way for this node to repair its blockchain with the blocks that it missed and update its blog accordingly to be consistent with the rest of the system.

One suggestion is to have every node maintain information about its first uncommitted index. This index is then attached to `ACCEPT`, `ACCEPTED`, and `DECIDE` messages. For instance, when the *leader* receives an `ACCEPTED` message from an *acceptor*, if the attached index is smaller than what the *leader* has, the *leader* can send all necessary information to repair that *acceptor*'s log with the `DECIDE` message. On the other hand, if the index of the *leader* is smaller than that of the *acceptor*, the *leader* will request all necessary information to repair its log from that particular *acceptor*.

This is just a hint to a possible approach. You still need to devise the rest of the implementation details and handle any edge cases that may arise.

5 Testing

You should implement a **3-second message-passing delay** when receiving a message to allow for testing concurrency. You are highly encouraged to implement and test your program in the following order of priority:

1. Multi-Paxos normal operation with replicated log (no failure)
2. Crash failure & recovery from disk
3. Network partition & reconnection
4. Blog post application

You demo will also be evaluated with more weight towards items with higher priorities.

6 User Interface

Your program should always be able to ingest new user inputs and shouldn't ever block user input.

6.1 Protocol-Related Input Interface

- `crash`: terminate the process
- `failLink(dest)`: fail the connection between the issuing process and the `dest` node. Links are assumed to be **bidirectional**, i.e. once P1 runs `failLink(P2)`, P1 can neither send to or receive from P2
- `fixLink(dest)`: fix the connection between the issuing process and the `dest` node
- `blockchain`: print all conent of every block in the blockchain
- `queue`: print all pending operations in the queue

6.2 Application-Related Input Interface

- `post(username, title, content)`: issue a Post operation for creating a new blog post authored under `username` with the given `title` and `content`
- `comment(username, title, content)`: issue a Comment operation for creating a new comment authored under `username` with `content` for the blog post with `title`
- `blog`: print a list of the title of all blog posts in chronological order. If there are no blog posts, print "BLOG EMPTY" instead
- `view(username)`: print a list of the title and content of all blog posts authored under `username` in chronological order. If there are no posts authored under `username`, print "NO POST" instead

- `read(title)`: print the content of the blog post with `title`, and the list of comments for the post with their respective authors in chronological order. If a post with `title` doesn't exist, print "POST NOT FOUND" instead

6.3 Output Interface

These are the minimum output we are looking for during demo to reflect the state of your system. You are free to change the wording or formatting as long as these output are still identifiable. You are also encouraged to include additional logging as long as it doesn't clutter the console.

- "NEW POST ⟨*title*⟩ from ⟨*username*⟩": output when a node successfully applies a decided Post operation to its blog. If a blog post with `title` already exists, the node should print "DUPLICATE TITLE" instead without creating any post
- "NEW COMMENT on ⟨*title*⟩ from ⟨*username*⟩": output when a node applies a Comment operation to its blog. if a blog post with `title` does *not* exist, the node should print "CANNOT COMMENT" instead without creating any comment
- "PREPARE/PROMISE/ACCEPT/ACCEPTED/DECIDE ⟨*ballot_num*⟩": clearly indicate in console output whenever a protocol-related message is **sent or received**. You should include sender/recipient information and any additional information associated with these messages
- "TIMEOUT": output when an *acceptor* times out on waiting for the *leader* to `decide` on the operation it forwarded

7 Teams

This project can be done solo or in a team of 2.

8 Deadlines and Deployment

This project will be due on 06/12/2023 by 9 AM. This is the deadline to submit your codebase to Gradescope. We will have a short demo over Zoom for everyone's work on Monday 06/12 and Tuesday 6/13/2023. Late demos are scheduled for Wednesday 06/14/2023. **Keep in mind that late submissions / demos will result in a 10% deduction in points**.

You are required to use TCP/UDP sockets for communication. You can demo on different machines i.e, run 1 instance of the node on a different machine than yours. It is also acceptable if you just use several processes on the same machine to simulate the distributed environment.

More information on signing up for teams and time slots will be released on Piazza closer to the demo date.