LIBR: A Moderated, Censorship-Resilient Digital Forum

Aradhya Mahajan, Lakshya Jain, Devesh Bhardwaj, Diwanshu Yadav, Arpan Jain

Abstract

Digital public forums are central to fostering open dialogue and building inclusive communities, but existing centralized and decentralized models each face critical limitations. Centralized forums are prone to censorship, often undermining free expression, while decentralized forums lack adequate moderation, exposing users to harmful or disruptive content. Striking a balance between censorship resistance and community-driven moderation is essential to address these challenges.

In this paper, we present LIBR*, a novel framework for creating a censorship-resilient yet moderated public forum. Leveraging a hybrid approach,* LIBR *combines concepts from decentralized networks, distributed hash tables (DHTs), and consensus protocols such as proof-of-work blockchains. It employs a role-based architecture with distinct types of nodes—clients, database nodes, and moderators—to ensure efficient message storage, community-led moderation, and robust governance. Key innovations include a mechanism for partial immutability of stored messages, moderation certificates for content validation, and a dynamic global state that adapts to network participation.*

By addressing issues of censorship, moderation, and governance through innovative use of decentralized technologies, LIBR *demonstrates the potential for a balanced platform that upholds both free expression and constructive dialogue. This work provides a blueprint for designing next-generation public forums that prioritize inclusivity, resilience, and community trust.*

1. Introduction

Public forums are meant to foster open dialogue, encourage diverse perspectives, and empower communities to thrive through collaboration and shared knowledge. However, the traditional centralized model of digital forums often falls short of this ideal. Centralized platforms are susceptible to censorship, where content deemed unfavorable by authorities or platform owners can be removed arbitrarily. This undermines the principles of free expression and community-driven engagement, eroding trust in the platform and its governance.

On the other hand, fully decentralized forums pose their own set of challenges. While they resist centralized censorship, their unmoderated nature can enable the proliferation of harmful, offensive, or malicious content. This creates a risk of anti-social elements disrupting the sense of community, turning forums into chaotic or even hostile spaces rather than safe havens for public discourse.

Balancing these opposing challenges—preventing undue censorship while maintaining a healthy and inclusive community—requires a new approach. A moderated yet censorship-resilient digital forum could preserve the spirit of open dialogue while ensuring the platform remains conducive to constructive interactions.

In this paper, we present LIBR, a framework that aims to solve the inherent issues with centralized forums while providing a community-moderated environment for discussions.

1.1. Addressed Issues

The proposed solution aims to bridge the gap between centralized and decentralized forum models by addressing the following key issues:

- **Censorship in Centralized Forums:** Ensuring that user-generated content is not arbitrarily removed or suppressed due to the biases or control of centralized authorities. This preserves the core principles of free speech and transparency in public discourse.

- **Lack of Moderation in Decentralized Forums:** Mitigating the risk of harmful, illegal, or disruptive content that can arise in fully decentralized systems. Such content can degrade the community experience and undermine the forum's purpose as a safe and inclusive space for discussion.

- **Balancing Resilience and Responsibility:** Striking a balance where the platform is censorship-resilient yet maintains community-driven moderation to safeguard its integrity and ensure productive engagement.

1.2. Challenges

Building a platform that balances censorship resistance with effective moderation poses several challenges, particularly when compared to conventional centralized systems. These include:

- **Message Storage:** Traditional platforms rely on centralized databases, which are not suitable for a censorship-resilient system. While a Distributed Hash Table (DHT) offers a trivial solution for decentralized storage, building a fault-tolerant system that ensures partial immutability of forum messages requires additional mechanisms and safeguards to maintain data integrity and availability.

- **Partial Immutability of Message Storage:** Pure immutability can be trivially achieved using a hash chain-based implementation, as seen in blockchain systems like Bitcoin. However, this approach is cost-inefficient and impractical for a forum, where every message would function as a transaction. Since forum messages do not alter the system state as significantly as blockchain transactions, total immutability is unnecessary. Instead, a replicated DHT setup provides a more efficient and suitable solution to achieve partial immutability.

- **Proof of State Transaction:** Due to lack of a centralized figure, or a reliant governance or Proof of Stake mechanism, it is difficult to validate whether a resource like Storage Provided, promised through a State Transaction, is actually available for use.

- **Network Dynamics:** The platform must be resilient to dynamic network participation, where nodes frequently join or leave. Addressing the challenges of churn while ensuring data consistency, availability, and seamless functionality across the network is critical for system reliability.

- **Dynamic Global State:** The platform must respond dynamically upon changes in the network. As more resource intensive nodes join (or leave) the network, the overall factors like replication and fault tolerance (for moderation and storage) should change in correspondence. This would require a consensus mechanism among different nodes to access and modify the global state.

- **Moderation:** Unlike centralized systems, this platform requires a decentralized, weighted moderation mechanism. Such a system must fairly evaluate and incentivize user contributions, aligning moderation practices with community interests while minimizing the risk of misuse or biased actions.

- **Governance Model for Node Behavior:** A decentralized governance scheme is essential to monitor and regulate node behavior. This involves incorporating user feedback to assess node reliability and performance, enabling adaptive governance. The design must balance automated processes with community-driven input while safeguarding against malicious actors and ensuring network integrity.

2. Overview

We propose the creation of a public forum that upholds community-driven moderation guidelines while being resilient to censorship risks associated with centralization. Our approach leverages concepts from Decentralized Networks, Peer-to-Peer Architecture, Blockchain, and related technologies. LIBR employs a consensus protocol, such as a Proof-of-Work Blockchain, as a black-box mechanism to compute a globally agreed-upon state. This section provides a brief overview of the underlying concepts and technologies, along with the rationale for their inclusion in our protocol.

3. Underlying Theoretical Concepts

Several core components of the LIBR protocol rely on well-established distributed systems and cryptographic primitives. This section summarizes two key underlying theoretical concepts that enable censorship resistance, decentralized storage, and consistency despite adversarial behavior: Distributed Hash Tables (DHTs) and Byzantine Consistent Broadcast (BCB).

3.1. Distributed Hash Tables (DHTs)

DHTs are decentralized systems that provide efficient key-based lookup by distributing data across a set of nodes. Each node is responsible for a portion of the keyspace, determined by a consistent hashing function. The LIBR protocol uses a replicated DHT setup to achieve:

- Deterministic and decentralized mapping of user messages to database nodes.

- Redundant storage via the replication factor R, enabling fault tolerance.

- Efficient discovery and retrieval of data without a central coordinator.

This mechanism draws inspiration from systems like Chord and Kademlia, which offer logarithmic lookup times and resilience to node churn.

3.2. Byzantine Consistent Broadcast (BCB)

To ensure that messages are correctly validated and stored even in the presence of malicious nodes, LIBR employs Byzantine Consistent Broadcast. In this protocol:

- A message broadcast by a client is received consistently by all honest moderator nodes.
- If any honest node accepts a message, all others eventually do as well.
- Messages not conforming to community moderation policy are still consistently rejected.

This guarantees consensus on which messages receive valid moderation certificates, even under the assumption that some moderators may act arbitrarily or maliciously. The concept is derived from classical results in Byzantine agreement and reliable broadcast protocols.

3.3. Hashchains for State Reconstruction

LIBR nodes maintain a local hashchain that records state transitions such as moderator joins/leaves and database participation. Each transaction is hashed with a link to the previous one, forming an immutable ledger. This structure:

- Enables reconstruction of the global community state at any point in time.
- Provides tamper-evidence and ordering for control operations.
- Supports auditability of participation and moderation history.

These hashchains form the foundation for global configuration resolution and are compatible with block-based consensus protocols such as Tendermint or Bitcoin.

4. The LIBR System Architecture

The core protocol is fully described by the overall connections between different participating nodes. Unlike conventional peer-to-peer distributed systems, or traditional stake-based implementations, the protocol consists of several types of participating nodes. This section describes the architectural implementation of the LIBR protocol in detail.

4.1. Roles of Different Participating Nodes

- **Clients:** Client Nodes (Clients) act as the primary interface for users to interact with the protocol. They, and only they, allow the users to create and read messages on the forum. They are also crucial to facilitate governance actions that require user input. Clients are also responsible to broadcast messages, collect moderation signatures, create moderation certificates and interact with database nodes to store and fetch messages.
- **Database:** Database Nodes (DBs) are responsible for storing forum messages. They participate as nodes of the Replicated DHT Setup to manage storage while ensuring partial immutability and data availability.
- **Moderators:** Moderator nodes (Mods) are tasked with evaluating and moderating messages based on the community's moderation scheme. They validate the content of messages received from Clients if they find it suitable with the community guidelines.

The LIBR protocol uses a Consensus Mechanism, such as a Proof-of-Work Blockchain, as a Blackbox and thus may require some parties to participates as nodes of the used consensus mechanism.

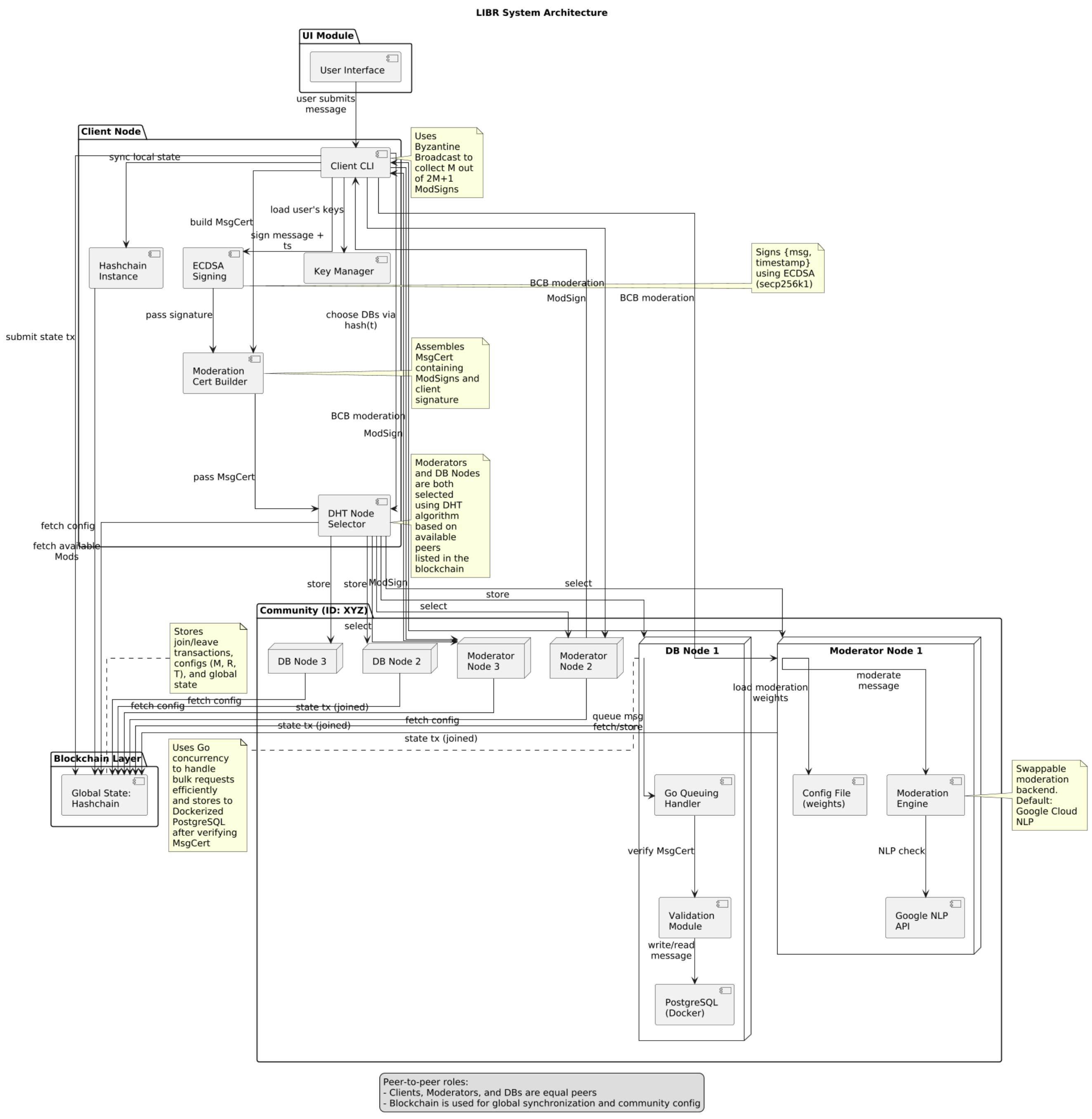


Figure 1. Detailed System Architecture of LIBR

4.2. What is a Community?

A **Community** is a group of Moderators and Database Nodes that collectively contribute their hardware and network resources to support a given set of Clients. These participants help maintain message integrity, moderation consistency, and data availability within their respective Communities.

Communities are the fundamental administrative units within the protocol. Each Community operates independently but adheres to the same base protocol, and may coexist with others on the same blockchain.

Incentivization: Contributors (Mods and DBs) may be incentivized through off-chain or blockchain-native reward mechanisms to offer their services to the protocol.

Synchronization and Configuration: The blockchain layer is used to globally synchronize Community configurations and store key metadata required to validate and interact with that Community.

Global Parameters for a Community: Each Community is uniquely identified and governed by a set of global parameters stored on-chain:

- **Community Public Key:** A cryptographic identifier for the Community used to validate transactions and associate metadata.
- **Traffic** (T)**:**

  $$T = f\left(\frac{\text{number of messages}}{\text{time}}\right)$$

  This variable helps normalize Unix timestamps across all nodes. All timestamps are divided by T to standardize event processing.
- **Moderator Fault Tolerance** (M)**:**

  $M =$ Minimum number of moderator signatures required

  The message must be sent to $2M + 1$ moderators to tolerate up to M faulty or malicious nodes. If the total number of moderators is less than $2M+1$, the message is sent to all.
- **DB Replication Factor** (R)**:**

  $R =$ Number of DBs where each message is initially stored

  Messages are sent to R database nodes for redundancy. If the number of available DB nodes is less than R, the message is sent to all.

4.3. System Model

- **Moderation Signature (*ModSign*):** A LIBR moderator node, upon validating a User Message, returns a Moderation Signature. Each *ModSign* consists of:
  - *public_key*: The public key of the moderator who issued the signature.
  - *sign*: A digital signature over the message content and timestamp, created using the moderator's private key.

  This signature binds a moderator's validation to a specific User Message and its timestamp. All moderators signing the same message will generate identical signatures if honest. The ModSign structure is:

Listing 1. Structure of a Moderation Signature (ModSign)

```
{
  "public_key": "pubkey",
  "sign": "sign"
}
```

- **Message Certificate (*MsgCert*):** A *MsgCert* binds a User Message to its validation record. It is a cryptographic envelope that proves the message has been validated by at least $2f + 1$ moderators (in a system with a fault tolerance threshold f). The structure includes:
  - *sender*: Public key of the user sending the message.
  - *msg*: The actual message content.
  - *ts*: Timestamp of when the message was sent.
  - *mod_cert*: A linear array of ModSign entries.
  - *sign*: A user-generated signature over `msg` and `ts`.

Listing 2. Structure of a Message Certificate (MsgCert)

```
{
  "sender": "senderpk",
  "msg": "msg",
  "ts": "timestamp",
  "mod_cert": [
    {
      "public_key": "pubkey",
      "sign": "sign"
    }
  ],
  "sign": "sign"
}
```

- **Stored Messages (*StoredMsg*):** These are the persistently stored objects in the database nodes. Each entry consists of:

- *sender*: The public key of the message originator.
- *content*: The message body.
- *timestamp*: A 64-bit integer representing the time of submission.

Stored messages are extracted from valid MsgCerts after verification.

4.4. Protocol Messages

- **User Messages (*Msg*):** User Messages are generated by clients and form the core data unit in the system. A User Message is sent to moderators for validation. It includes:
  - *message*: The content submitted by the user.
  - *timestamp*: A Unix timestamp representing the client's local time of submission.

Listing 3. User Message sent to moderators

```
{
  "message": "hello",
  "timestamp": "1744219507"
}
```

- **Moderation Response (*ModSign*):** Once a moderator approves a User Message, it returns a cryptographic signature over the message and timestamp. This constitutes a Moderation Signature.

Listing 4. Moderation Response from a moderator

```
{
  "public_key": "02227547a108c40745cf479c54e4
      30da76a75f2350772370020c10832d4de14409"
      ,
  "sign": "3045022100b7caecd571414dad964fcf41
      aeac03a9be2eb56d0c30f24107db2713e2763f9
      b0220786ee3321cddced6d49cf96f2436f55bba
      657cd6f16481abdec462f0f2869fea"
}
```

- **State Transactions:** These are broadcasted by participating nodes to initiate or reflect changes in the global system state. All transactions share the following general structure:
  - *sender, recipient, amt*: Base fields, where `amt` is kept zero for control messages.
  - *data*: A nested object containing the transaction type and any relevant metadata.
  - *nonce*: A monotonically increasing number from the sender to ensure uniqueness.
  - *sign*: Signature by the sender over the transaction body.

Listing 5. Genesis of a Community

```
{
  "sender": "creatorPublicKey",
  "recipient": "0x0000000000000",
  "amt": 0,
  "data": {
    "type": "GENESIS",
    "name": "communityName",
    "metadata": "",
    "traffic": "",
    "modFaultTolerance": "",
    "dbReplicationFactor": ""
  },
  "nonce": senderNonce,
  "sign": ""
}
```

Listing 6. Database Node Joined

```
{
  "sender": "dbPublicKey",
  "recipient": "communityPublicKey",
  "amt": 0,
  "data": {
    "type": "DB_JOINED",
    "metadata": {
      "ip": "",
      "port": "",
      "other": ""
    }
  },
  "nonce": senderNonce,
  "sign": ""
}
```

Listing 7. Database Node Left

```
{
  "sender": "discovererPK",
  "recipient": "communityPublicKey",
  "amt": 0,
  "data": {
    "type": "DB_LEFT",
    "leaver": "dbPublicKey",
    "metadata": {
      "ip": "",
      "port": "",
      "other": ""
    }
  },
  "nonce": senderNonce,
  "sign": ""
}
```

Listing 8. Moderator Node Joined

```
{
  "sender": "modPublicKey",
  "recipient": "communityPublicKey",
  "amt": 0,
  "data": {
    "type": "MOD_JOINED",
    "metadata": {
      "ip": "",
      "port": "",
      "other": ""
    }
  },
  "nonce": senderNonce,
  "sign": ""
}
```

Listing 9. Moderator Node Left

```
{
  "sender": "discovererPK",
  "recipient": "communityPublicKey",
  "amt": 0,
  "data": {
    "type": "MOD_LEFT",
    "leaver": "modPublicKey",
    "metadata": {
      "ip": "",
      "port": "",
      "other": ""
    }
  },
  "nonce": senderNonce,
  "sign": ""
}
```

4.5. Data Structures

- **Message Database:** The *Message Database* (MessageDB) is the primary data structure used to store *Stored Messages* in the databases. It is abstracted as a key-value pair structure, where `MessageDB[timestamp]` maps a *Timestamp* (key) to an array of *Stored Messages* (value). The LIBR protocol does not place significant emphasis on the absolute ordering of messages and treats all messages received within the same second equivalently. However, to ensure consistency, correct databases store the array of *Stored Messages* in a deterministic manner, such as sorting them alphabetically.

- **Hashchain Instance:** The LIBR protocol employs a *Hashchain* data structure to store consensus-based data, particularly *State Transactions*. The immutability and widespread utility of Hashchains in known and implemented consensus protocols make them an ideal choice for this purpose. Each node, irrespective of its contribution or participation type, maintains a local instance of the *Hashchain*. This Hashchain is used to reconstruct the *Global State* when required.

The UML Class Diagram shows the System Model in detail

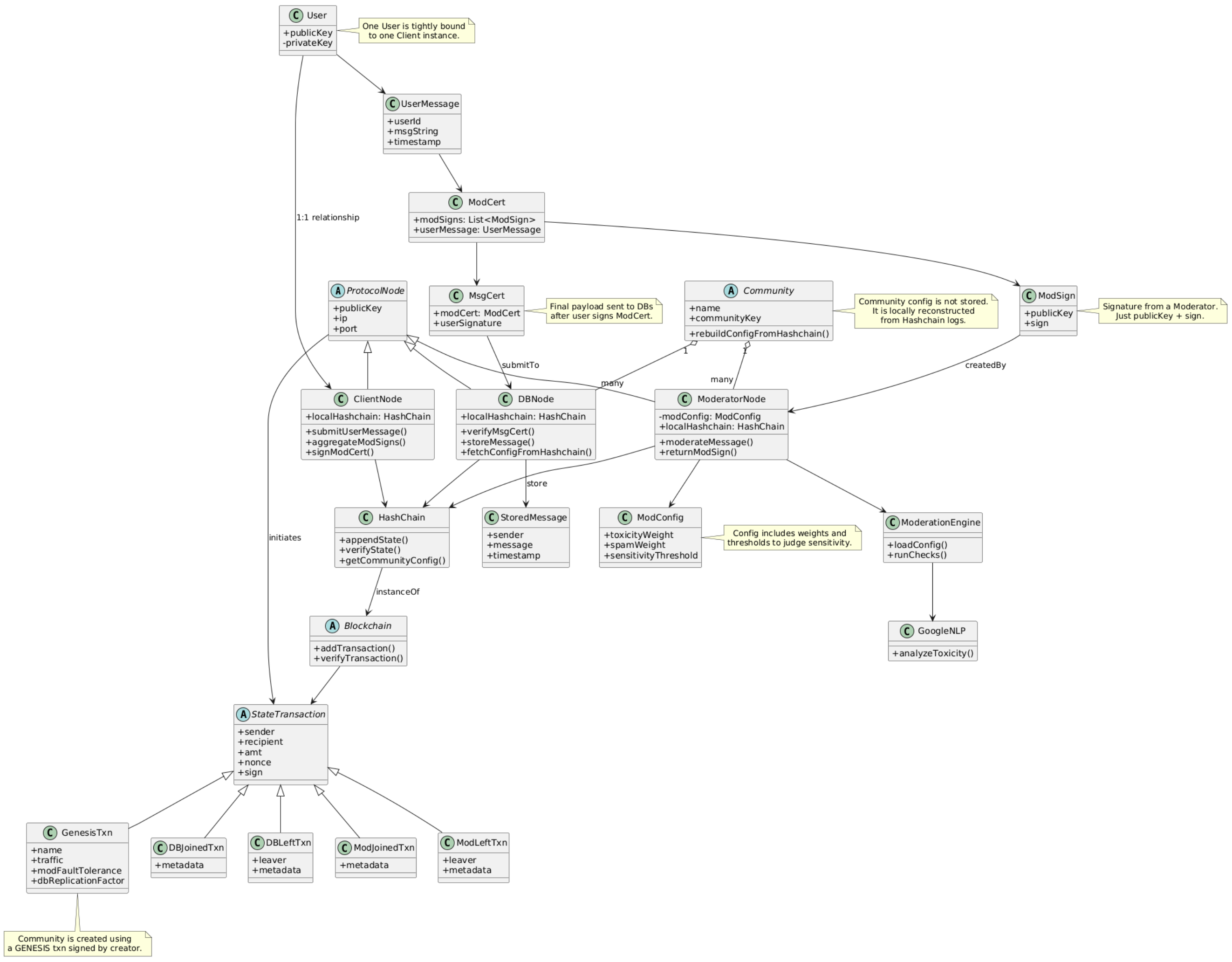


Figure 2. UML Class diagram illustrating the system model of LIBR.

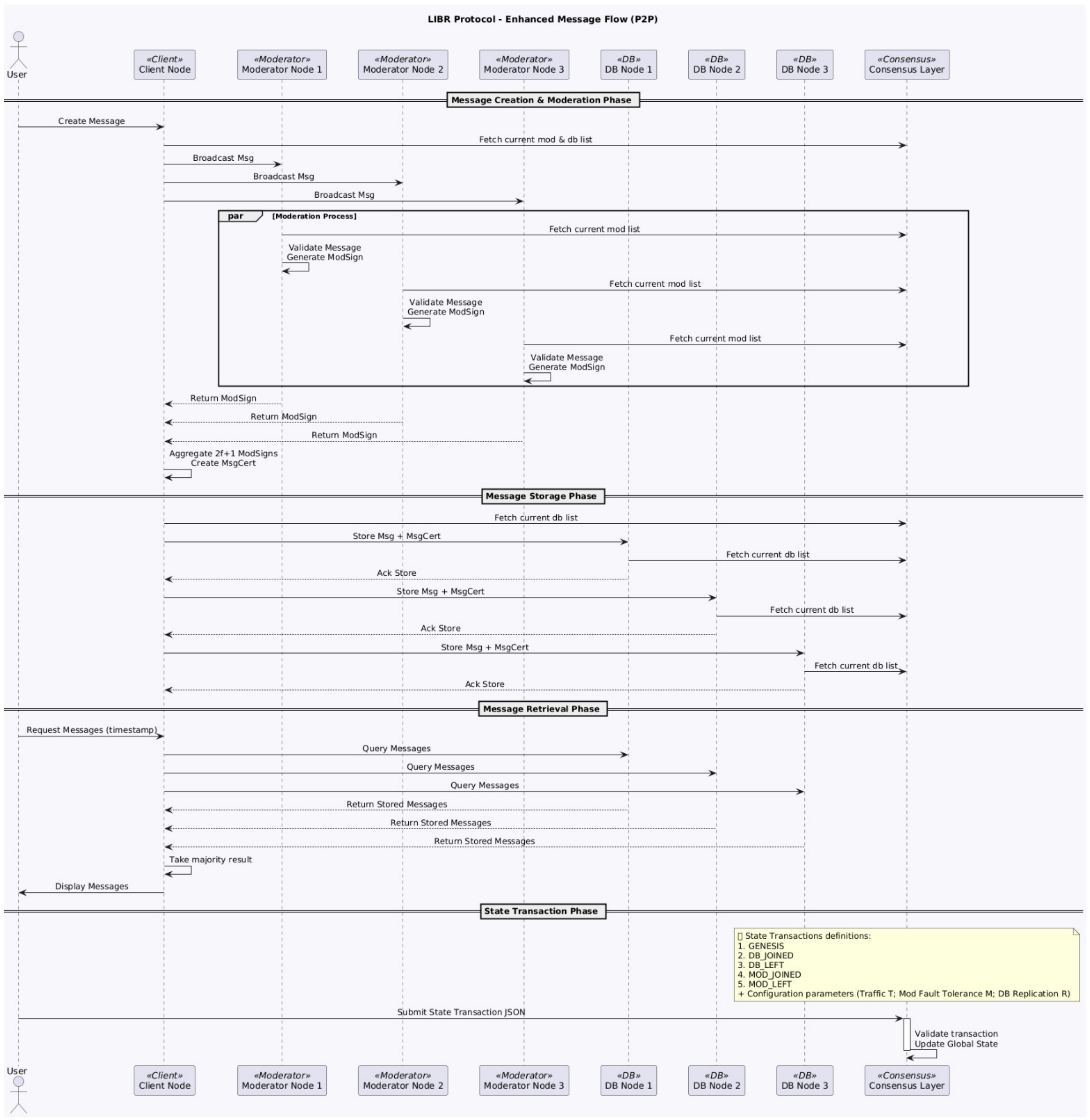


Figure 3. Sequence diagram illustrating the message storage and fetching process in LIBR.

4.6. Core Protocol

The core protocol of LIBR defines the foundational steps and mechanisms required to achieve its objectives. This subsection outlines the process of message storage and fetching in a structured, step-by-step manner.

4.6.1 Message Storage

- ***Step 1: Message Submission:*** The process begins when the user submits a message to the forum through a client. The user may act as their own client or rely on a trusted third party to submit the message on their behalf.
- ***Step 2: Message Moderation:*** The client employs a Byzantine Consistent Broadcast protocol to broadcast the submitted message to a quorum of Moderator Nodes. Honest Moderators validate messages that adhere to community guidelines and reject those that do not. In either case, the Moderator generates a *Moderation Signature* (MODSIGN), which is returned to the client as proof of validation.
- ***Step 3: Identifying Database Nodes:*** Based on the community's defined replication factor (*k*), the user message is redundantly stored on *k* Database Nodes (DB nodes). The LIBR client uses a Distributed Hash Table (DHT) to deterministically locate the appropriate *k* DB nodes. Each DB node is then approached by the client with a *Stored Message*, which is a tuple comprising the user message and its corresponding Moderation Certificate.

  The Algorithm to locate the appropriate k DB nodes from the Hashchain Instance is given here.
- ***Step 4: Storing the Message:*** When a DB node receives a *Stored Message*, it performs several verification steps. First, it validates the attached Moderation Certificate to ensure its authenticity. Then, it checks whether the message is assigned to its storage based on the community's allocation rules. If both verifications succeed, the message is stored in the local *Message Database* (*MessageDB*), keyed by its timestamp.

4.6.2 Message Retrieval

- **Step 1:** ***Query Submission:*** The process begins when the user queries a timestamp to the forum through a client. The user may act as their own client or rely on a trusted party to query on their behalf.
- **Step 2:** ***Identifying Database Nodes:*** Based on the timestamp queried, the client finds the k nodes to retrieve the message. The Client then queries all the k DB nodes for the messages

Algorithm 1 Storage Support (Generic)

```
Input: Blockchain transaction history $\mathcal{T}$, timestamp $ts$, replication factor $k$
Output: List of $k$ selected database nodes

// Step 1: Preprocessing — compute currently active DB nodes
1: Initialize empty map nodeStatus
2: for all $tx \in \mathcal{T}$ do
3:     if $tx$.Type = DB_JOINED then
4:         nodeStatus[$tx.Sender$] ← true
5:     else if $tx$.Type = DB_LEFT and $tx.Data.Leaver \neq \emptyset$ then
6:         nodeStatus[$tx.Data.Leaver$] ← false
7:     end if
8: end for
9: Let activeNodes ← all keys in nodeStatus with value true, sorted

// Step 2: Select $k$ nodes deterministically using timestamp
10: Compute SHA256 hash of timestamp $ts$
11: Extract 64-bit integer seed from first 8 bytes of hash
12: Initialize PRNG with extracted seed
13: Initialize empty set selected
14: while size of selected < min($k$, |activeNodes|) do
15:     Sample index $i$ from PRNG in [0, |activeNodes| − 1]
16:     if $i \notin$ selected then
17:         Add $i$ to selected
18:     end if
19: end while
20: Let selectedPublicKeys ← values in activeNodes at selected indices

// Step 3: Construct node metadata from join transactions
21: Initialize empty map nodeMeta
22: for all $tx \in \mathcal{T}$ where $tx$.Type = DB_JOINED do
23:     if "metadata" in $tx.Data.Metadata$ then
24:         Parse JSON from $tx.Data.Metadata$["metadata"] into map $parsed$
25:         nodeMeta[$tx.Sender$] ← (parsed["ip"], parsed["port"])
26:     end if
27: end for

// Step 4: Return selected database node metadata
28: Initialize empty list selectedNodes
29: for all $pk \in$ selectedPublicKeys do
30:     if $pk \in$ nodeMeta then
31:         Append nodeMeta[$pk$] to selectedNodes
32:     end if
33: end for
34: return selectedNodes
```

The Algorithm to locate the k DB nodes from the Hashchain Instance is same as the Node Identification Algorithm to store messages and is given here.

- **Step 3:** ***Fetching the messages:*** When a correct DB node receives a timestamp query from a client, it fetches and responds with all the Stored Messages keyed by the queried timestamp. The client, after receiving enough responses can then choose to deliver the user with the majority response.

5. Implementation

The LIBR protocol is implemented entirely in the Go programming language. Go's strong support for concurrency, lightweight goroutines, and native channels allows for efficient and scalable network communication between nodes, while retaining code simplicity. Critical components such as broadcasting, message validation, and DHT coordination extensively use `sync.WaitGroups` and Go channels to handle asynchronous behavior, minimizing latency and maximizing throughput.

5.1. Modular Architecture

The architecture is deliberately modular, with a clear separation of concerns between Clients, Moderators, and Database Nodes. Each component is implemented as an independent Go module, capable of being compiled and run separately. Shared logic—such as message schemas, signature verification, and blockchain communication—is maintained in remote, versioned Go modules and imported using `go get`. This promotes code reuse and enforces loose coupling between system components, allowing upgrades or reconfiguration of any module without impacting the others.

5.2. Client Interface

The Client module offers a Command Line Interface (CLI) built using the `cobra` library. This design provides users with a simple and powerful tool for interacting with the protocol. Users can post messages, request moderation certificates, query data, or trigger governance actions—all through terminal commands. The CLI is well-suited for early-stage experimentation and scripting, while being extendable to GUI or web clients in the future.

5.3. Database Layer

For storage, the system currently uses a Dockerized instance of PostgreSQL. This choice provides a balance between SQL-based reliability and ease of deployment across environments. Each Database Node in the network operates independently but conforms to the replication strategy defined by the community parameters. The containerized setup ensures that nodes can be deployed consistently across platforms and updated in a controlled manner.

5.4. Moderation Engine

The moderation layer is powered by Google Cloud Natural Language Processing (NLP) services, which analyze and validate messages based on configurable guidelines. This allows rapid prototyping and access to state-of-the-art language models without requiring heavy local computation. However, the moderation logic is abstracted through an interface, allowing any third-party or custom NLP engine to be substituted by simply changing the configuration.

5.5. Pluggability and Reconfiguration

Every component of the LIBR protocol is designed to be reconfigurable. Communities or advanced users may:

- Replace PostgreSQL with another SQL or NoSQL database.
- Swap out Google Cloud NLP with a local LLM or another moderation API.
- Modify the CLI or add a new interface (e.g., Web UI, Mobile App).
- Add new client-side features using custom plug-ins or forks.

This approach empowers each Community to tailor the system according to its performance needs, regulatory requirements, or infrastructure preferences—without deviating from the core protocol.

The source code is organized using idiomatic Go conventions, with each major component (Client, Moderator, Database Node) implemented in a separate module. Code-level documentation, including API details, file structure, and developer setup instructions, is available in the project's GitHub repository and supporting documentation files.

6. Challenges and Mitigation Strategies

During the development and deployment of the LIBR protocol, several challenges emerged due to the novel nature of the system and its underlying goals of censorship resistance, modularity, and decentralization. This section outlines the major technical and operational hurdles and how they were addressed.

6.1. 1. Validity of State Transactions

Ensuring the authenticity and integrity of state transactions—such as moderator join/leave requests, message approvals, and DB updates—was a non-trivial problem in a decentralized and permissionless setup. Traditional distributed systems often rely on a trusted server or consensus framework to validate such transitions.

To overcome this, we developed a custom fork, Blockshare-LIBR of a lightweight blockchain framework named **Blockshare** (github.com/Aradhya2708/Blockshare). Blockshare-LIBR provides a minimal Proof-of-Work consensus layer to securely log and synchronize all critical state changes across community nodes. Each transaction is cryptographically signed and verified on-chain, ensuring tamper-proof community coordination.

6.2. 2. Hardware Compatibility

Although Go provides cross-platform support and excellent binary portability, differences in hardware environments (e.g., ARM vs x86, OS-specific syscall behavior, etc.) may introduce unforeseen bugs during real-world deployment. To mitigate this:

- We statically compile binaries to maximize portability.
- Docker is used wherever applicable (especially for DB nodes) to abstract away environmental differences.
- Community contributors are encouraged to test the system on diverse hardware setups, and automated CI pipelines are used to test builds across popular platforms.

6.3. 3. Cost and Time Efficiency

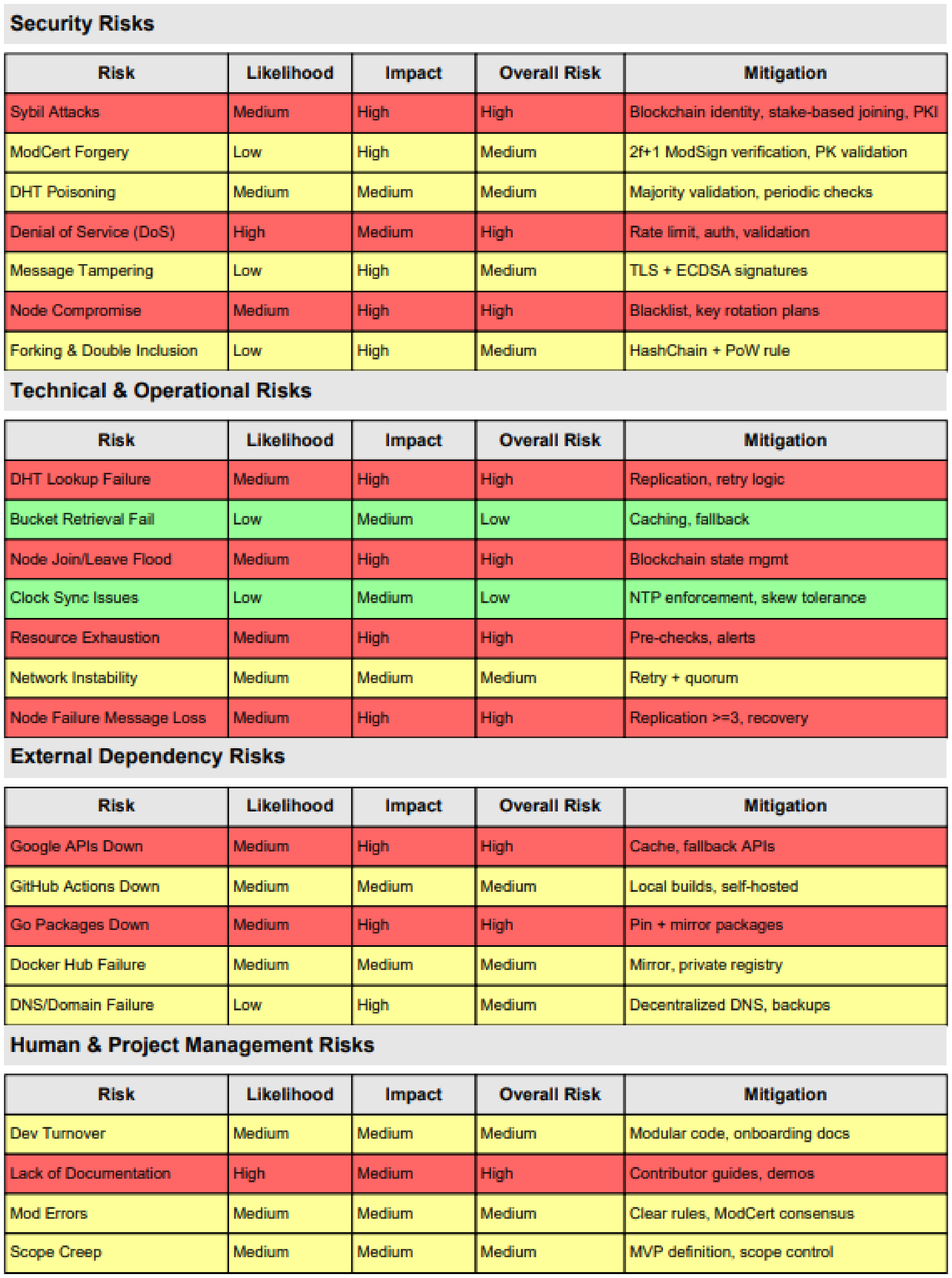
Compared to traditional digital forums that operate on centralized infrastructure, LIBR's distributed nature introduces overhead—both in terms of latency and resource usage. Message validation by multiple moderators, signature generation, and multi-node replication add to system load.

To mitigate these performance costs:

- The community can tune protocol parameters (e.g., replication factor R, fault tolerance M) to balance efficiency vs robustness.
- Caching mechanisms and local snapshotting are planned for future versions to reduce redundant reads.
- Batch-based message verification and moderation pipelines can help reduce processing time, especially under heavy load.

While the system is heavier than traditional forums, it provides trade-offs in censorship resistance, transparency, and resilience that are essential for high-trust or adversarial settings.

7. Risk Analysis



Security Risks

Risk	Likelihood	Impact	Overall Risk	Mitigation
Sybil Attacks	Medium	High	High	Blockchain identity, stake-based joining, PKI
ModCert Forgery	Low	High	Medium	2f+1 ModSign verification, PK validation
DHT Poisoning	Medium	Medium	Medium	Majority validation, periodic checks
Denial of Service (DoS)	High	Medium	High	Rate limit, auth, validation
Message Tampering	Low	High	Medium	TLS + ECDSA signatures
Node Compromise	Medium	High	High	Blacklist, key rotation plans
Forking & Double Inclusion	Low	High	Medium	HashChain + PoW rule

Technical & Operational Risks

Risk	Likelihood	Impact	Overall Risk	Mitigation
DHT Lookup Failure	Medium	High	High	Replication, retry logic
Bucket Retrieval Fail	Low	Medium	Low	Caching, fallback
Node Join/Leave Flood	Medium	High	High	Blockchain state mgmt
Clock Sync Issues	Low	Medium	Low	NTP enforcement, skew tolerance
Resource Exhaustion	Medium	High	High	Pre-checks, alerts
Network Instability	Medium	Medium	Medium	Retry + quorum
Node Failure Message Loss	Medium	High	High	Replication >=3, recovery

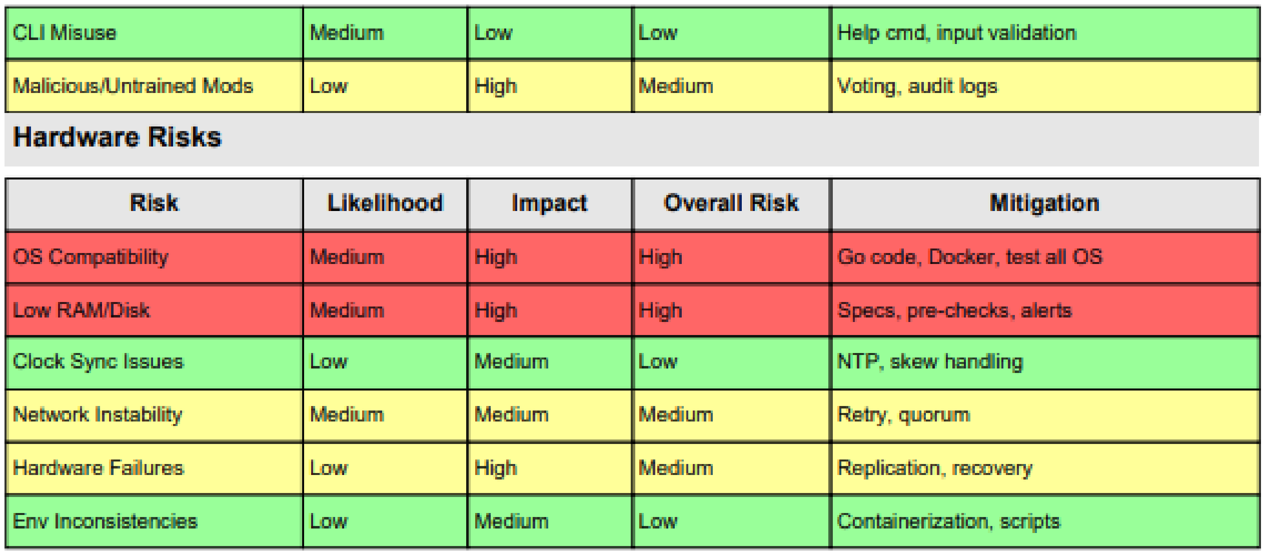
External Dependency Risks

Risk	Likelihood	Impact	Overall Risk	Mitigation
Google APIs Down	Medium	High	High	Cache, fallback APIs
GitHub Actions Down	Medium	Medium	Medium	Local builds, self-hosted
Go Packages Down	Medium	High	High	Pin + mirror packages
Docker Hub Failure	Medium	Medium	Medium	Mirror, private registry
DNS/Domain Failure	Low	High	Medium	Decentralized DNS, backups

Human & Project Management Risks

Risk	Likelihood	Impact	Overall Risk	Mitigation
Dev Turnover	Medium	Medium	Medium	Modular code, onboarding docs
Lack of Documentation	High	Medium	High	Contributor guides, demos
Mod Errors	Medium	Medium	Medium	Clear rules, ModCert consensus
Scope Creep	Medium	Medium	Medium	MVP definition, scope control

Figure 4. Risk Analysis of LIBR



CLI Misuse	Medium	Low	Low	Help cmd, input validation
Malicious/Untrained Mods	Low	High	Medium	Voting, audit logs

Hardware Risks

Risk	Likelihood	Impact	Overall Risk	Mitigation
OS Compatibility	Medium	High	High	Go code, Docker, test all OS
Low RAM/Disk	Medium	High	High	Specs, pre-checks, alerts
Clock Sync Issues	Low	Medium	Low	NTP, skew handling
Network Instability	Medium	Medium	Medium	Retry, quorum
Hardware Failures	Low	High	Medium	Replication, recovery
Env Inconsistencies	Low	Medium	Low	Containerization, scripts

Figure 5. Risk Analysis of LIBR (cont.)

8. Progress

The LIBR project is being developed over the course of the semester through a series of well-defined research, planning, and implementation milestones. This section outlines the chronological evolution of the project based on actual deliverables and team activities.

8.1. Milestone Timeline

- **Jan 10, 2025 – Team Finalization**
  Project team members were finalized and roles were loosely designated based on domain interest.

- **Jan 16, 2025 – Project Ideation Began**
  Initiated discussion and brainstorming sessions around forum architectures, decentralization, and censorship resilience.

- **Jan 20, 2025 – Submission of Core Project Idea**
  Finalized and submitted the core concept: a decentralized, censorship-resilient, yet moderated public forum framework.

- **Jan 21–Feb 2, 2025 – Background Research Phase**
  Team divided into sub-groups to study critical concepts:
  - Blockchain, decentralization, and consensus protocols.
  - Distributed Hash Tables (DHTs) and Byzantine Consistent Broadcast (BCB).
  - Peer-to-peer (P2P) architectures and forum moderation techniques.

- **Feb 3, 2025 – Exploration of Space-Based Architecture**
  Investigated alternatives like space-based coordination models and compared their feasibility with message-passing architectures.

- **Feb 5, 2025 – System Design and UML Drafting**
  Initiated system modeling using UML diagrams to define actor interactions, message flows, and modular responsibilities.

- **Feb 11, 2025 – SRS and Documentation Initiated**
  Formal documentation, including Software Requirements Specification (SRS), was started. A live collaborative draft was maintained.

- **Mar 5, 2025 – Initial Protocol Architecture Finalized**
  Defined the LIBR node roles (Client, Moderator, DB Node) and their interactions. Determined the core pipeline of moderation, certification, and storage.

- **Mar 6–10, 2025 – Tech Stack Finalization**
  Evaluated JavaScript, Python, C++, and Go for protocol implementation. Rejected Python (too high-level), C++ (less suitable for modular systems), and JavaScript (single-threaded runtime). Finalized Go for its concurrency support, static typing, and modular development model.

- **Mar 11–15, 2025 – Learning Go and Basics**
  Learned Go syntax and semantics. Built basic REST servers and clients. Familiarized the team with Go's tooling and compilation.

- **Mar 16–19, 2025 – Concurrency and Goroutines**
  Studied Go's concurrency primitives—goroutines, channels, and sync patterns—required for modular, non-blocking network programming.

- **Mar 20–23, 2025 – Cryptographic Module Prototyping**
  Implemented standalone test modules for cryptographic primitives such as hashing and digital signature verification in Go.

- **Mar 27, 2025 – Blockchain Layer Forked and Extended**
  Forked the Blockshare repository to serve as the test blockchain layer. Fixed bugs, added logging, and customized features to fit LIBR's governance and state transaction requirements.

- **Mar 31, 2025 – Project Development Kickoff**
  Officially began development in the main GitHub repository. Created separate modules for each node role.

- **Apr 1, 2025 – Shared Documentation Finalized**
  Completed internal documentation to ensure consistent protocol understanding and cross-module compatibility.

- **Apr 2, 2025 – Client Skeleton Completed**
  Developed foundational logic for client interaction: message creation, signature handling, and CLI scaffolding.

- **Apr 4, 2025 – Database Node Skeleton Completed**
  Created the core database module including logic for accepting and validating incoming messages.

- **Apr 6, 2025 – DHT-Based Node Selection Algorithm Implemented**
  Wrote and integrated a deterministic node selection mechanism using timestamp hashing and PRNG over active DB nodes.

- **Apr 8, 2025 – Initial Integration Tests**
  Successfully tested client and database modules independently and together using a dummy moderator.
- **Apr 11, 2025 – Moderator Module Completed**
  Finalized the moderator module to receive user messages, evaluate content, and return cryptographic ModSigns.
- **Apr 12, 2025 – Protocol Prototype Integration**
  Integrated Client, Moderator, and Database modules to achieve an end-to-end working prototype for message certification and storage.

8.2. Current Status

The LIBR system has evolved from an architectural concept to a functional prototype. A REPL client interface, modular back-end services, and verifiable message workflows have been developed. Final stages involve full system integration, stress testing, and preparing a complete test deployment across virtualized nodes.

9. Use Case

The LIBR protocol is designed for communities that require transparent, censorship-resilient, and community-governed communication. Below, we outline a core use case where the strengths of LIBR are most applicable.

9.1. Decentralized Public Forums with Moderation Transparency

Conventional online forums and discussion platforms rely on centralized infrastructure and opaque moderation policies. This makes them vulnerable to:

- Arbitrary censorship of user content.
- Lack of verifiability in moderation decisions.
- Single points of failure and control.

The LIBR protocol directly addresses these concerns by enabling communities to establish their own infrastructure and moderation rules, all verifiable on-chain.

Scenario

A group of researchers and activists wants to create a public forum to discuss politically sensitive topics. They are concerned about:

- Platform-level takedowns.
- Shadow banning or content suppression.
- Lack of clarity in moderation decisions.

Using LIBR, they can:

- Deploy their own community with specific moderator nodes and database resources.
- Allow users to post messages via CLI or browser-based clients.
- Ensure messages are validated by a quorum of moderators using configurable thresholds.
- Log all decisions and state changes (moderator activity, DB participation, etc.) immutably on Blockshare.
- Customize infrastructure (e.g., swap PostgreSQL with another DB, switch NLP validation providers) without altering the protocol's integrity.

9.2. Benefits Over Traditional Forums

LIBR offers several distinct advantages:

- **Censorship Resistance:** No single entity controls message flow or content approval.
- **Transparency:** All moderation actions are recorded as verifiable transactions.
- **Community Governance:** Users can influence the forum by running clients, proposing mods, or adjusting replication/moderation policies.
- **Modularity:** Components like the database, NLP validator, or UI interface are fully customizable.

This makes LIBR suitable not only for public forums but also for whistleblower platforms, academic discussions, open-source governance, or any environment where message integrity, traceability, and trust are vital.