Blockchain-Based Voting System for Secure Distributed Elections

Waseem Ahmad
Harvard College
waseemahmad@college.harvard.edu

Max Peng
Harvard College
mpeng@college.harvard.edu

05/04/25

Abstract

We present a prototype blockchain-based voting system that combines classical distributed-systems principles with modern Web3 tools to deliver secure, transparent, and decentralized elections. Our design builds on an in-class exploration of consensus algorithms, extending our Paxos/Raft/BFT exercises to a Proof-of-Authority (PoA) scheme, and integrates off-chain storage via IPFS alongside Ethereum-style wallet authentication. The system comprises a Next.js client for election management and vote casting, a Node.js/Express server exposing REST endpoints, a custom PoA consensus module in JavaScript, and IPFS for candidate metadata persistence. We validated our approach with end-to-end tests across multiple nodes. Our evaluation demonstrates that PoA reduces computational overhead while ensuring tamper-resistant vote ordering, IPFS guarantees data availability without prohibitive gas costs, and Web3 wallets enable strong voter authentication. This work illustrates how distributed-systems theory can be translated into a real-world application for trustworthy elections, and provides a foundation for further enhancements in scalability, privacy, and more complicated models.

The GitHub repository for this project is available at: `https://github.com/waseemahmad1/cs2620_final`, which also includes presentation slides.

1 Introduction

In an era where digital technology has taken over aspects of daily life, the persistence of largely physical and centralized voting systems stands in stark contrast. Most modern democracies continue to rely on in-person polling places and rudimentary practices. While these traditional approaches offer familiarity, they also present persistent concerns regarding accessibility, scalability, transparency, and public trust in electoral outcomes.

Recent debates surrounding the integrity and inclusivity of elections (i.e. the past few presidential elections) demonstrate the need to re-imagine how voting systems can leverage modern computational advances. Distributed systems, and blockchain technologies in particular, offer the potential to fundamentally reshape how elections are conducted by introducing tamper-resistance, decentralization, and verifiable auditability. These technical breakthroughs, alongside the potential social implications, inspired our project: as discussed in the motivation subsection below.

1.1 Motivation

The primary motivation behind our project is to demonstrate the feasibility and value of a blockchain-based voting system that addresses the challenges in traditional election infrastructure. Having enjoyed CS 2620's deep dive into consensus and replication, we were particularly excited by Design Exercise 4 (`cs2620_hw4`), where we built a leadership election logic and learned about algorithms like Paxos, Raft, etc, alongside

simplified Byzantine-fault-tolerant replicas across multiple nodes. That exercise taught us how subtle protocol choices affect safety and liveness under node failures.
For our final project, we wanted to take those lessons further:

- **Explore alternative consensus schemes**
- **Leverage IPFS storage** for further data persistence
- **Integrate Web3 wallets** to decentralize voter authentication, and explore blockchain technologies

By building a (very nontrivial) distributed application that combines both academic concepts from class and real-world blockchain technologies, we hope to solidify our understanding of how distributed systems theory scales into practice, a key theme throughout the semester.

1.2 Problem Formulation

More explicitly, for this project, we aim to design and implement a secure, transparent, and distributed election platform that meets the following goals originally in our proposal:

1. Ensures **security** via vote integrity. Each vote, once cast, is immutable and tamper-proof through decentralized ledger storage. We will evaluate against edge cases (ie. adversarial attempts and simulated node failures)
2. Promotes **transparency** while preserving anonymity. All recorded votes are publicly verifiable on the chain and through an independent audit mechanism
3. Eliminates central dependence via **fault tolerance**. The system does not rely on a single trusted authority for validation or storage. Instead, vote blocks are pinned to IPFS across multiple peers, and block inclusion is controlled by a distributed consensus protocol rather than by a central server.

Our approach centers on building a prototype that mirrors the core components of a real-world election system:

- A **web-based voting client** interacting with the voter's Web3 wallet.
- A **distributed backend** based on Ethereum smart contracts (or a private PoA chain) to record vote transactions.
- A **consensus protocol module** inspired by both classical algorithms (Paxos/Raft) and blockchain PoA/PoS variants to validate and order votes.
- **IPFS-based storage** for candidate metadata and ballot definitions, ensuring decentralization.
- An **audit mechanism**, exposing a read-only endpoint that anyone can use to reconstruct and verify the vote history.

Through this project, we will explore how the foundational principles of distributed computing we've learned throughout the semester can be applied to one of society's most foundational processes: ensuring secure, transparent, and fair elections.

2 Methodology

We will provide a very brief overview of our system, then discuss specific mechanisms of certain meaningful aspects, namely the, consensus algorithm, use of IPFS, transactions of ethereum blockchain, and audit mechanism.

2.1 Brief Overview of System

Let us first provide a brief overview of our system, then dig deeper into specific components. Our platform comprises a Web3-enabled Next.js client and multiple Node.js/Express server instances. The client connects to users' Web3 wallets (via MetaMask), presents a voting interface, and submits ballots to the backend. Behind the scenes, each server instance validates and orders incoming votes using our PoA consensus module; servers stay in sync via Redis Pub/Sub and are fronted by a simple HTTP load balancer. Once a block is sealed, the server pins it to IPFS for decentralized storage and records its header in an Ethereum-style smart contract. Finally, an audit API allows anyone to fetch the full chain of IPFS CIDs and contract events to independently verify election integrity.

Now we will delve into the four key mechanisms of the system: the consensus algorithm, IPFS storage, Ethereum blockchain, and audit mechanism.

2.2 Consensus Algorithm

Early in the project we considered using proof of work to stay true to classical blockchain ideas, but it quickly became clear that it was impractical. Proof of work is slow, computationally expensive, and in a permissioned setting like an election it is unnecessary. Instead, we built a Proof of Authority (PoA) model where a fixed set of validator nodes take turns signing blocks in a round-robin fashion. These validators are hardcoded into the system and could represent university-run servers or trusted election officials. Because the validator set is known and pre-approved, there is no mining competition and consensus is fast and predictable.

To coordinate agreement among these authority nodes we implement a simplified PoA algorithm. A validator is deterministically selected from a predefined list in round-robin order. Each new block bundles one or more signed vote transactions along with the election ID and voter signature. The consensus module first verifies each vote by checking that the signature matches its vote data and then ensures that the block's parent hash matches the most recent chain tip. Finally, it confirms that the block was produced by the correct validator for that turn before appending it to the chain.

This lightweight, deterministic approach gives us fault tolerance without the high resource costs of proof of work or the complex staking requirements of PoS, making it pretty efficient for our project. It mostly mirrors actual blockchain deployments where the priority is secure agreement among a known group of participants rather than global, permissionless trust. By eliminating mining delays and heavy computation, our PoA consensus ensures that vote inclusion is efficient, and well suited for our election use case.

2.3 Decentralized Storage with IPFS

In our blockchain-based voting system we employ the InterPlanetary File System (IPFS) to achieve decentralized and immutable storage of all election-related data. IPFS is a peer-to-peer distributed file system that eliminates reliance on centralized servers by distributing data across multiple nodes. We realized early that storing full ballot objects or candidate metadata directly on-chain would be prohibitively expensive and inefficient. Instead, we wrote a set of helper functions in `ipfs.js` that pin JSON blobs—such as ballot definitions, vote payloads, and candidate information—to a local IPFS daemon.

Each uploaded file returns a unique content identifier (CID) based on its cryptographic hash. We then embed these CIDs in our block headers stored on the blockchain, so that the ledger only needs to store compact pointers rather than bulky data. This design allows us to leverage IPFS for off-chain storage while preserving blockchain immutability and consensus. By separating data storage from the ledger, we keep on-chain transactions lightweight and focused on vote ordering.

This approach offers two key benefits. First, data availability is ensured even if one server node crashes or drops out, since any IPFS peer holding the CID can provide the original JSON. Second, tamper resistance is enforced by design: any modification to the stored content would change its CID and break the reference in the block header. Together, the hybrid of IPFS and blockchain creates a robust, scalable voting infrastructure that gives us effectiveness and transparency without incurring high gas fees.

2.4 Ethereum Blockchain

Next up is how we write our vote blocks to a blockchain that remains compatible with Ethereum tools while avoiding traditional mining overhead (as mentioned in our presentation, we do not have the funds to do mining!). We deployed a simple smart contract featuring a single method called `appendBlock`, which validators invoke after they gather and finalize a batch of votes. This method takes three parameters: the parent block's content identifier, the list of new vote CIDs, and the validator's digital signature produced with an election specific wallet key. In our JavaScript modules `Block.js` and `Blockchain.js`, each block header is constructed to include only these fields, which keeps the on-chain logic minimal and focused. By restricting the contract to this one function we ensure that state changes remain straightforward and auditable (with our audit mechanism).

Our consensus process runs entirely off-chain in the file `PoAConsensus.js`, so we do not perform proof of work or compete to mine blocks. When a validator's turn arrives it packages the votes into a block object, pins the block payload to IPFS, and collects the resulting CIDs. It then submits those CIDs and the parent identifier to the smart contract in a single transaction. This atomic call records the block header without any extra computation or gas heavy operations. The off-chain consensus combined with this on-chain header commits block history at essentially no cost.

Despite the lightweight design the ledger works seamlessly with standard Ethereum clients and libraries. Anyone can call JSON RPC methods such as `eth_getLogs` to fetch all `appendBlock` events for a particular election. Each event contains the data needed to reconstruct the chain exactly as defined in `Blockchain.js`. With those logs and the corresponding IPFS payloads it is possible to rebuild the entire vote history. From the perspective of an external observer the system behaves like a normal Ethereum based ledger but without the high fees and long confirmation times normally associated with on-chain mining.

2.5 Audit Mechanism

We built a single REST endpoint, `/auditProof`, that automates end-to-end verification of any election. When you call `/auditProof?electionId=XYZ`, the server first queries our Ethereum-style smart contract for every `appendBlock` event associated with that election. For each event it gathers the list of IPFS content identifiers, then fetches the corresponding JSON files to reconstruct the original vote payloads.

The response is a comprehensive JSON array in which each entry contains the parent CID, the array of vote CIDs, and the validator's anonymized signature. This means anyone can run a simple script to confirm that the recorded blocks form an unbroken chain, that each signature validates against the known validator keys, and that every cast vote appears exactly once. This endpoint consolidates all verification steps—contract log retrieval, IPFS fetches, and chain reconstruction—into a single web call. It provides transparency as anyone is able to independently confirm the integrity and completeness of election results without relying on a central authority.

3 Development Log

Aside from initial brainstorming for the proposal and waiting for proposal feedback, we primarily completed the core implementation over a nine-day period (more or less), which we divide into three consecutive three-day phases.

Days 1–3: Blockchain Core and Consensus
Day 1. We first read up to review the blockchain lecture (the 3blue1brown video on youtube was quite helpful as well). We initialized the repository and sketched out our blockchain data structures in `Block.js`

and `Blockchain.js`. After defining the block format (prev-hash pointer, vote list) and in-memory chain operations, we committed foundational tests for block linking.

Day 2. With the data structures in place, we turned to consensus. In `PoAConsensus.js` we implemented a simple Proof-of-Authority scheme: a fixed list of validator addresses in `config/validators.json` take turns in round-robin order to sign and seal new blocks. We wrote unit tests to simulate validator failures and message reordering, to help us verify that only the designated authority at each height could produce a valid seal.

Day 3. Next, we built the Express API in `index.js`, exposing endpoints like `/castVote` and `/getChain`. We wired up vote submission to the consensus module and block-sealing logic, ensuring that new blocks were created when vote batches reached a threshold or after a timeout. By the end of Day 3, clients could post signed ballots and receive back block identifiers.

Days 4–6: Decentralized Storage and Authentication
Day 4. In terms of storage, we turned to usind IPFS. We added helper functions in `ipfs.js` to pin JSON blobs (candidate lists, ballots) and retrieve CIDs. We refactored the API so that each new block's payload was pinned to IPFS before being sealed on the blockchain.

Day 5. We shifted focus to voter authentication. In the client code, we scaffolded a Next.js interface and wired in `ethers.js` calls to MetaMask. Voters now sign their vote objects locally; the server validates signatures against the public key before accepting ballots.

Day 6. With most of our core functionality correct, we polished the frontend UI. We implemented the "Create Election," "Quick Vote," "End Election," and "View Ledger" buttons, connecting each to the appropriate API calls. We also added basic error handling for invalid election IDs and signature failures.

Days 7–9: Testing, Audit, and Writing Paper
Day 7. We designed our audit mechanism: an `/auditProof` endpoint that returns the list of block CIDs and on-chain events. We then wrote a small client script to fetch and locally verify each block's parent pointer generated in `Blockchain.js`.

Day 8. This day was primarily for more testing and starting the notebook. We conducted end-to-end tests: spinning up three PoA nodes, creating elections, casting votes, and simulating node crashes. We wrote up the skeleton of our draft, thinking about how we wanted to split our notebook.

Day 9. In our final day (mostly), we updated the README with setup instructions and double-checked that all tests pass. By the end of Day 9, our prototype was stable and met the goals of integrity, transparency, and decentralization that we had outlined in the beginning.

3.1 Challenges

During development, we encountered several challenges while building and debugging our distributed system. Below, we highlight the main issues and how we addressed them:

- **Cross-Machine Redis and IPFS Connectivity.** Ensuring seamless communication between nodes using Redis and IPFS was a significant challenge. Specifically:
  - **Redis:** One node was configured to use a remote Redis server, but initial attempts to connect failed due to `protected-mode` settings and firewall restrictions. Debugging involved reconfiguring the Redis server to allow external connections, setting up authentication, and ensuring the correct port was open.

  - **IPFS:** The IPFS daemon on one machine was not accessible externally due to incorrect API binding (`127.0.0.1` instead of `0.0.0.0`). This required modifying the IPFS configuration and restarting the daemon. These connectivity issues underscored the importance of proper network configuration and testing in distributed systems.

- **Client-Server Port Mismatch.** A recurring issue during testing was the mismatch between the client and server configurations. The client was hardcoded to send requests to `localhost:4850`, while the server was running on `localhost:3003`. This led to repeated `ECONNREFUSED` errors in the client. Resolving this required:
  - Standardizing the server port across all configurations.
  - Ensuring the client dynamically used the correct server URL, especially in multi-node setups where one server might run locally and another remotely.

- **Blockchain Synchronization Across Nodes.** Synchronizing the blockchain state across multiple nodes proved to be a challenge. While Redis was used to broadcast new blocks, there were edge cases where:
  - A node would miss a block broadcast if it was temporarily disconnected or restarted.
  - The order of block processing could differ between nodes, leading to temporary inconsistencies.

  To address this, we implemented a periodic blockchain sync mechanism where nodes could request the latest state from their peers if they detected a discrepancy.

- **Load Balancer Health Check Sensitivity.** The custom load balancer in `load-balancer.js` periodically probes the `/health` endpoint of each server. During testing, we observed that transient delays (e.g., due to IPFS pinning or garbage collection) caused the balancer to incorrectly mark healthy nodes as down. This led to uneven traffic distribution and occasional single points of failure. To mitigate this:
  - We increased the health-check interval and added retries before marking a node as unhealthy.
  - We logged health-check failures for offline debugging, which helped identify and address performance bottlenecks.

- **Eventual Consistency in Blockchain State.** When implementing the `/createElection` and `/castVote` endpoints, we assumed that all nodes would immediately reflect the latest blockchain state. However, due to the distributed nature of IPFS and Redis, there were occasional delays in propagating new blocks or votes. This required:
  - Adding retries and backoff mechanisms for critical operations.
  - Designing the system to tolerate temporary inconsistencies, ensuring eventual consistency without blocking user actions.

4 Results

4.1 General System Design

Previously, in the methodology section, we gave a brief introduction to our system design. Now, we discuss it more deeply. Our voting platform is organized into a layered architecture that ensures secure vote collection, decentralized storage, and high availability. At the front end, a Next.js web client connects to users' Web3 wallets via MetaMask to handle authentication and ballot signing. The client submits election creation, voting, and ledger queries to one of several Node.js/Express server instances running the code in `server/index.js`. These servers are synchronized through Redis Pub/Sub (configured in `index.js`) so that any vote received by one node is immediately broadcast to all peers. A simple HTTP load balancer (`load-balancer.js`) sits in front of the servers and routes requests in round-robin fashion, with health checks to remove unresponsive nodes from rotation.

Beneath the API layer is our custom blockchain module, implemented in `server/blockchain/Block.js`, `server/blockchain/Blockchain.js`, and `server/blockchain/PoAConsensus.js`. Votes are batched into blocks by the PoA consensus algorithm, where a fixed set of validator addresses takes turns signing new blocks. Once a block is sealed, its full vote payload is pinned to IPFS via helper functions in `ipfs.js`, returning content identifiers (CIDs). The block header—consisting of the parent CID, array of new CIDs, and validator signature—is then recorded in an off-chain Ethereum-style ledger by invoking the `appendBlock` method on a smart contract.

This design separates concerns and theemes cleanly: the web client handles user interaction and signature generation, the API layer manages request validation, synchronization, and smart-contract calls, the PoA module enforces consensus, IPFS provides durable off-chain storage, and the load balancer together with Redis ensures horizontal scalability and resilience. By combining these components, we achieve a system that can tolerate node failures, prevent data loss, and allow any participant to verify election integrity through our audit API.

4.2 System Design Choices

Our design choices were guided by the three core goals of vote integrity, transparency, and decentralization (noted in our proposal). Specifically, here are some choices we made:

- **Redis for Real-Time Communication.** To enable efficient communication between nodes, we used Redis as a lightweight pub/sub mechanism. This allowed nodes to broadcast new blocks and votes in near real-time, ensuring that all participants maintained a consistent view of the blockchain.
- **Custom Load Balancer.** Instead of relying on a third-party solution, we implemented a custom load balancer (`load-balancer.js`) to distribute client requests across multiple servers. This design gave us fine-grained control over health checks, failover behavior, and traffic routing, ensuring high availability and fault tolerance.
- **Dynamic IPFS Integration.** By dynamically importing the IPFS client (`ipfs-http-client`) and connecting to a remote IPFS daemon, we ensured that the system could scale across multiple machines. This approach also allowed us to decouple blockchain operations from local storage, improving modularity.
- **Validator-Based Election Creation.** Election creation is restricted to validators listed in the Proof-of-Authority consensus. This ensures that only trusted entities can initialize elections, preserving the integrity of the voting process while maintaining decentralization.

Together, these design choices emphasize scalability, modularity, and fault tolerance (some of the main themes we highlighted in our original proposal). Redis ensures efficient communication, the custom load balancer improves reliability, and IPFS integration supports decentralized data storage.

4.3 Example Election Walkthrough

Below is a step-by-step illustration of both the user experience in the Next.js client and the underlying technical flow:

1. **Create an Election.** In the client, a user enters a unique `Election ID` (e.g., `spring2025_senate`) and clicks "Create Election." The frontend sends a POST to `/createElection` in `index.js`, where the server checks Redis for an existing key, initializes an empty vote list in memory, and pins the candidate metadata JSON to IPFS via `ipfs.js`. The resulting CID is stored in Redis under `election:<ID>:meta` and returned to the client as confirmation.
2. **Cast a Vote.** Voters switch to the "Quick Vote" panel, enter the same `Election ID`, type their choice, and approve the transaction in MetaMask. The client submits the signed vote object to `/castVote`. The server verifies the signature with `ethers.js`, publishes the vote to the Redis channel, and the PoA consensus module in `PoAConsensus.js` batches it into the next block. Once the block is sealed,

`ipfs.js` pins the block payload and the server calls the smart contract's `appendBlock` method with the parent CID, new vote CIDs, and validator signature.

3. **View Ledger (Ongoing).** At any time, anyone can click "View Ledger," enter the `Election ID`, and the client issues a GET to `/viewLedger`. The server retrieves the list of block CIDs from Redis and for each CID fetches the corresponding JSON from IPFS. It returns an array of vote entries (each containing voter address and choice) without exposing wallet private keys. The client renders this anonymized list in real time.

4. **End the Election.** Only the election creator's wallet can call `/endElection`. After confirming a MetaMask signature, the server sets a Redis flag `election:<ID>:closed` and instructs the PoA module to seal the final block immediately. This ensures no further `/castVote` requests are accepted for that ID.

5. **View Results.** Once the election is closed, "View Ledger" shows the permanent closed ledger of votes, which can be tallied up to find the winner. Note that there are now disabled further voting UI elements. Meanwhile, auditors can call `/auditProof` to receive the full sequence of parent CIDs, vote CIDs, and signatures for independent verification.

5 Discussion

Our project demonstrates the feasibility of using a blockchain-based system for secure and decentralized voting. However, as with any prototype, there are areas for improvement and opportunities for theorizing how we can scale the system to handle larger, real-world elections. Below, we discuss some key considerations and potential enhancements.

5.1 Scaling for Larger Blockchain Voting Systems

While our prototype is designed for small-scale elections, scaling the system for larger, real-world use cases introduces several challenges:

- **Consensus Scalability.** The current Proof-of-Authority (PoA) consensus is efficient for a small number of validators but may become a bottleneck as the number of nodes increases. Transitioning to a more scalable consensus algorithm, such as Delegated Proof-of-Stake (DPoS) or actual PoW conesnsus algorithms typically used, could improve performance while maintaining security.
- **Data Storage.** As the number of voters and elections grows, the size of the blockchain and associated IPFS data will increase significantly. Implementing pruning mechanisms could help manage storage requirements, though we haven't seen any issues in our prototype yet.
- **Network Latency.** In a global election scenario, network latency could impact the speed of block propagation and vote confirmation. Optimizing the peer-to-peer network topology and introducing regional nodes could mitigate these delays. Our current computers are all in Cambridge, within a few hundred feet of one another, but real-world elections are statewide, national, or even global.

5.2 Future Work

To build on this prototype, we propose the following areas for future work. Despite the countless hours put into this system, we are only a two-person undergraduate group. Some of the limitations we encountered could be addressed in the future.

- **Enhanced Security Features.** Implementing advanced cryptographic techniques, such as threshold signatures or multi-party computation, to further secure the voting process.
- **Decentralized Identity Management.** Integrating decentralized identity solutions to streamline voter authentication while preserving privacy. Note that if someone gets ahold of your current Metamask, then they will know your signature and can vote as you (meaning potential fraud). Granted this is difficult to do since the user must create a login or sign in with Google for Metamask, and we assumed these login options (and Metamask itself) were secure, it could still be a possibility.

- **Scalability Testing.** Conducting stress tests with thousands of nodes and voters to evaluate the system's performance under real-world conditions.
- **User Experience Improvements.** Developing a more user-friendly client interface, including mobile support, to make the system accessible to a broader audience. The current frontend is lacking, as our paper and project focuses more on effective implementation.
- **Interoperability with Existing Systems.** Exploring ways to integrate the blockchain voting system with existing election infrastructure, such as voter registration databases and audit systems that already are implemented in various forms of government.

By addressing these areas, we aim to evolve our prototype into a more robust, scalable, and secure blockchain-based voting system capable of supporting real-world elections (think thousands of votes, across geographical and time bounds).

6 Conclusion

We have designed and implemented a functional blockchain-backed voting platform that meets our core proposal goals of vote integrity, transparency, and decentralization. By combining a lightweight Proof-of-Authority consensus protocol with IPFS-based off-chain storage and Ethereum-style wallet authentication, our system achieves publicly verifiable vote data without relying on a single trusted authority and is distributed. Throughout development, we navigated challenges in dealing with IPFS and making system choices, ultimately delivering a prototype that demonstrates the practical viability of distributed-ledger technology for elections. We hope our work inspires further research into scaling blockchain voting to larger electorates, integrating advanced privacy techniques, and exploring hybrid consensus approaches. Thank you to the CS 2620 Professor Waldo and all the teaching staff for helping make this class an incredible experience.