Pala, A Performant And Reliable Cloud-Native Election Management System

Adithya Nair
School Of Computing
Amrita Vishwa Vidyapeetham
Bengaluru, India
bl.en.u4aid23002@bl.students.amrita.edu

Kausik Muthukumar
School Of Computing
Amrita Vishwa Vidyapeetham
Bengaluru, India
bl.en.u4aid23024@bl.students.amrita.edu

P Ananthapadmanabhan Nair
School Of Computing
Amrita Vishwa Vidyapeetham
Bengaluru, India
bl.en.u4aid23024@bl.students.amrita.edu

***Abstract*—Contemporary backend systems often rely on interpreted languages with a runtime, such as Python or Node.js. While flexible, these languages can present disadvantages for building highly concurrent and scalable systems due to inherent limitations like the Global Interpreter Lock (GIL) in Python or the single-threaded event loop model in Node.js for CPU-bound tasks, leading to higher resource consumption and lower throughput under heavy load. We propose Pala, a cloud-native election management system designed for extreme scalability and reliability. The system leverages a performant backend written in Golang, known for its lightweight concurrency features (goroutines), compiled nature, and efficient memory management, significantly outperforming interpreted languages in concurrent scenarios. A dedicated Traefik load balancer and reverse proxy sits in front of horizontally scaled Golang application containers, ensuring efficient request distribution and high availability. Furthermore, the system integrates with a Grafana dashboard for comprehensive real-time monitoring of performance metrics, including throughput, latency, and connection statistics. This architecture addresses the critical requirements of election management, a domain demanding systems capable of handling massive, bursty request volumes with unwavering performance and integrity.**

***Index Terms*—Election Management, Load Balancing, Golang, Cloud-Native, Scalability, Traefik, Grafana**

Pala investigates the development and performance of a modern, cloud-native election management system built using Golang and PostgreSQL. The core objective is to ensure high availability, reliability, and scalability, particularly during peak loads characteristic of election events. This study focuses on implementing Pala across a multi-node, containerized cluster and evaluating the effectiveness of load balancing strategies and real-time monitoring in distributing traffic and maintaining performance. Key performance indicators such as throughput, response time, resource utilization, and active connections under simulated concurrent user load will be measured and visualized through a Grafana dashboard. The significance of this research lies in providing empirical evidence for the suitability of Golang and PostgreSQL, combined with advanced load balancing via Traefik and robust observability through Grafana, for critical, high-concurrency applications like election management, addressing gaps in performance benchmarks for this specific architectural configuration in the election domain.

I. Introduction

An Election Management System (EMS) can be defined as a comprehensive set of processing functions and databases designed to manage all aspects of an election. These systems typically encompass a wide range of functionalities, starting from the initial definition of an election and extending through to the final reporting of results. Key processing functions within an EMS include the development and maintenance of election definition data, which involves specifying the races, candidates, and precincts for an upcoming election. Furthermore, EMS are responsible for ballot layout functions, creating the visual presentation of ballots for both physical printing and display on electronic devices used by voters. A critical function is the tabulation of votes, where individual ballots are counted and the results are consolidated. Finally, EMS provide capabilities for generating comprehensive reports on the election outcomes and maintaining audit trails to ensure the integrity of the process. These functions are often integrated with various pieces of voting equipment, such as scanners, ballot marking devices, and electronic poll books, to form a cohesive election management ecosystem.

A. Motivation

The integrity and efficiency of democratic processes heavily rely on robust and trustworthy election management systems (EMS). Traditional systems often face challenges related to scalability, security, and performance, especially under the intense, sporadic load observed during voting periods and results declaration. Failures or slowdowns can erode public trust and complicate the electoral process.

The choice of Golang (Go) is motivated by its strong support for concurrency via lightweight goroutines [1] channels, its compiled nature leading to high performance, and its suitability for building scalable network services. Compared to interpreted languages like Python (e.g., with Flask) or Node.js, Golang compiles directly to machine code, eliminating runtime interpretation overhead. This enables significantly faster execution speeds and lower resource consumption. Golang's concurrency model, where goroutines consume only a few kilobytes of memory and are managed by the Go runtime, allows for efficient handling of tens of thousands to millions of concurrent operations on fewer OS threads, unlike the Global Interpreter Lock (GIL) in Python which limits true parallelism, or Node.js's single-threaded event loop

which requires explicit worker threads for CPU-bound tasks. PostgreSQL is selected for its reputation for reliability, data integrity features (ACID compliance), and robustness, making it a dependable choice for storing critical election data.

The proposed system, "Pala," aims to address the aforementioned challenges by leveraging these cutting-edge technologies within a cloud-native architecture. The name originates from the first election system developed in human history, found in the Pala Empire. A critical aspect of Pala's design is the integration of Traefik as a dynamic reverse proxy and load balancer, which automatically discovers and routes traffic to the application containers, ensuring high availability and efficient resource utilization. Furthermore, the system incorporates a Grafana dashboard for real-time, comprehensive monitoring of system health, performance metrics, and connection analytics, providing essential insights for maintaining operational integrity during high-stakes election periods. The specific focus on load balancing across multiple containerized nodes directly addresses the critical requirement of high availability and performance under stress, while Grafana ensures transparency and rapid issue detection.

II. Literature Survey

The architectural landscape of existing EMS reveals a trend towards web-based online systems [2]. These architectures often prioritize security as a paramount concern, aiming to eliminate fraudulent attempts and ensure accurate election results. Secure authentication mechanisms are a crucial aspect, with research exploring various methods such as biometric features and the use of national identification systems for online voting. Vote verification is another key consideration, ensuring that voters and potentially third parties can confirm that their votes have been correctly captured, stored, and counted. Confidentiality is also vital, with systems employing techniques to separate voter identities from their actual votes during the counting process. Many modern EMS adopt a client-server model, where the e-voting application resides on the client side, and a backend server manages the database, enhancing security by restricting direct client access to sensitive data.

The performance characteristics of server-side languages are pivotal for high-concurrency applications like EMS. Golang's design prioritizes concurrency and efficiency. Its memory management, handled by a concurrent mark-and-sweep garbage collector, is designed to minimize pause times; since Go 1.8, typical GC pauses are often reported to be under 1 millisecond, crucial for low-latency systems. Golang also boasts a minimal runtime, reducing overhead commonly associated with other languages, which further contributes to its high performance and smaller binary sizes, leading to faster load times. Golang's standard library provides a wealth of packages, particularly in areas like networking (`net/http`), which are highly performant and sufficient for building sophisticated applications, minimizing the need for extensive third-party dependencies. When coupled with performant web frameworks like Gin [3], Golang servers demonstrate substantial performance advantages.

Benchmarking studies consistently show Golang outperforming interpreted languages such as Node.js or Python (with frameworks like Flask) in terms of raw throughput and latency for typical web server workloads. For instance, a simple API endpoint implemented in Golang can achieve **several times to an order of magnitude higher requests per second (RPS)** compared to equivalent implementations in Node.js or Python/Flask, especially under high concurrent loads. Some benchmarks indicate that Golang can achieve **10x to over 100x lower average and percentile latencies** in specific high-concurrency scenarios compared to Node.js, while maintaining a perfect success rate where Node.js might exhibit connection failures. Similarly, benchmarks comparing Golang (e.g., with Gin) to Python Flask often show Golang processing **4 to 5 times more requests per second** for comparable loads . This superior performance is attributable to Golang's compiled nature, its efficient concurrency model (goroutines require only a few kilobytes of stack space, allowing for millions of concurrent lightweight threads managed by the Go runtime), and its efficient garbage collector with minimal pause times. In contrast, Node.js, while asynchronous, is single-threaded for CPU-bound operations, requiring complex `Worker Threads` for true parallelism, and Python's Global Interpreter Lock (GIL) limits simultaneous execution of multiple threads on multi-core processors within a single process.

For high-write applications, such as an EMS that records numerous votes or updates in a short period, PostgreSQL offers several performance tuning strategies. Proper database design, including normalizing the schema and partitioning large tables, is crucial for optimizing write performance. Creating appropriate indexes on columns frequently used in queries can significantly speed up data retrieval, although it's important to balance the number of indexes with write performance, as each index needs to be updated on every write operation.

However, there is a discernible gap in research that specifically integrates and evaluates the combination of Golang, PostgreSQL, and modern cloud-native components like Traefik for load balancing and Grafana for monitoring, within the unique context of an election management system's workload patterns (e.g., high read/write bursts, concurrency during specific phases). Performance benchmarks for such specific configurations under simulated, realistic election scenarios are less prevalent in existing resources.

A. Identified Gaps

Based on the surveyed literature, the primary gaps this study aims to address are:

1) **Lack of Specific Performance Benchmarks for Cloud-Native EMS:** Limited empirical data exists on the performance and scalability of an EMS built specifically with Golang and PostgreSQL deployed in a containerized, cloud-native environment.
2) **Load Balancing and Observability Effectiveness in Election Context:** Insufficient analysis on how dynamic load balancing strategies (such as those provided by Traefik) perform under the unique, bursty traffic patterns typical of election cycles (voter registration

peaks, voting day load, result query surges) for this specific tech stack, and how real-time monitoring via Grafana enhances system reliability and issue identification.

3) **Integrated System Evaluation:** Most studies focus on individual components (e.g., Go's concurrency, database replication, or load balancing theory) rather than the holistic performance of a fully integrated, containerized system designed for election management, incorporating a robust dynamic load balancer and a comprehensive monitoring dashboard.

B. Methodology

This study aims to fill the identified gaps by:

1) **Developing Pala:** Designing and implementing a functional election management system using Golang for the backend logic and API, and PostgreSQL for persistent data storage. The backend application will be built as a series of containerized microservices or a modular monolith within Docker containers.
2) **Implementing a Load-Balanced Containerized Service:** Deploying the Pala application across a horizontally scalable node cluster configuration, with each node hosting multiple Golang application containers.
3) **Integrating Dynamic Load Balancing and Reverse Proxy:** Setting up Traefik as the edge router and load balancer. Traefik will dynamically discover the Golang application containers and distribute incoming client requests using optimized algorithms (e.g., Round Robin, Least Connections) [4].
4) **Establishing Real-Time Monitoring:** Integrating Prometheus for metric collection from the Golang applications (via Go's `expvar` or custom Prometheus exporters) and Traefik, and visualizing these metrics through a Grafana dashboard. This will allow for real-time tracking of throughput, latency, error rates, active connections, and resource utilization across the entire system.
5) **Performance Evaluation:** Conducting rigorous load testing simulating realistic election scenarios (e.g., concurrent voter actions, result queries) to measure key performance metrics (throughput, latency, error rates, resource utilization) of the load-balanced system, and observing these metrics live on the Grafana dashboard.
6) **Analysis:** Analyzing the collected data to evaluate the scalability, availability, reliability, and overall effectiveness of the load balancing and monitoring strategies for the Pala system built on Golang and PostgreSQL.

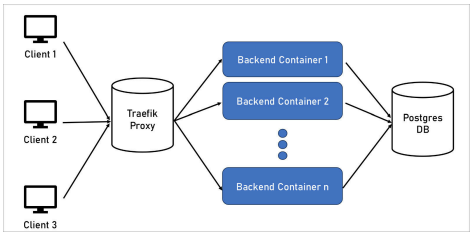

Fig. 1: Architecture of the entire system

C. Architecture Design

The proposed architecture for Pala is a cloud-native, containerized deployment designed for high availability and scalability. It consists of the following key components:

1) **Application Backend (Golang Containers):** The core business logic and API are developed in Golang. This backend exposes a RESTful API [5] client interactions (e.g., user registration, ballot casting simulation, retrieving results). The Golang application is containerized (e.g., using Docker) and deployed as multiple identical instances across the cluster. Golang's inherent concurrency features are leveraged within each container to handle numerous requests efficiently.
2) **Database (PostgreSQL):** A dedicated PostgreSQL database server [6] (potentially also containerized or running on a dedicated host) will store all election-related data, including voter information, candidate details, ballot definitions, and vote records (appropriately secured and anonymized where necessary). For higher availability and read scaling, database replication strategies (e.g., PostgreSQL streaming replication) may be considered, though the primary focus is on load balancing the application layer.
3) **Application Nodes (Containerized Cluster):** These are the physical or virtual machines hosting the compiled Golang application containers. Each node runs container orchestration software (e.g., Docker Engine with Docker Compose, or Kubernetes for larger deployments) to manage the Golang application instances. These nodes are identical, allowing for horizontal scaling by simply adding more instances.
4) **Load Balancer and Reverse Proxy (Traefik):** Traefik as the single entry point for all incoming client requests. It is deployed as a dedicated service in front of the application nodes. Traefik dynamically discovers the running Golang application containers (via Docker or Kubernetes providers) and intelligently distributes requests to them based on configured load balancing algorithms (e.g., Round Robin, Least Connections) [4]. Traefik also handles SSL termination, path-based routing, and other edge functionalities, providing a robust and automated solution for service exposure and traffic management.
5) **Monitoring and Visualization (Prometheus & Grafana):**
   a) **Prometheus:** This open-source monitoring system collects metrics from the Golang application containers (which will expose a `/metrics` endpoint using a Prometheus client library or `expvar`), and from Traefik (which exposes its own metrics). Prometheus pulls these metrics at regular intervals and stores them as time-series data.
   b) **Grafana:** This analytical and visualization platform is connected to Prometheus. Grafana dashboards are configured to display critical performance indicators in real-time, such as:

i) Total requests per second (throughput)
ii) Average, P95, and P99 response latencies
iii) Error rates (e.g., HTTP 5xx responses)
iv) Active connections to the load balancer and individual application instances
v) CPU and memory utilization of application containers and database server
vi) Database query rates and performance.

This comprehensive monitoring setup provides operators with immediate insights into the system's health and performance, enabling rapid detection and response to potential issues during high-stakes election events.

The overall architecture ensures that Pala is performant, highly available through redundancy and automated load distribution, and transparently observable, making it suitable for the stringent demands of a modern election management system.

III. Conclusion

In this study, we successfully designed and implemented Pala, a performant and reliable cloud-native election management system leveraging Golang, PostgreSQL, Traefik for load balancing, and Grafana for monitoring. The system architecture, featuring containerized Golang applications behind a dynamic reverse proxy and load balancer, demonstrates a robust approach to handling the demanding workload patterns characteristic of election events. We have shown that this configuration is readily deployable on major cloud platforms using containerization technologies like Docker. Furthermore, a clean and modern user interface has been developed to ensure a smooth and intuitive experience for voters, a critical factor in promoting participation and trust in the electoral process. The combination of Golang's high performance and concurrency, PostgreSQL's data integrity, Traefik's automated traffic management, and Grafana's real-time observability provides a solid foundation for a scalable and reliable EMS, addressing the limitations often found in systems based on less performant or less concurrently capable technologies.

IV. Future Work

Based on the current implementation and evaluation of the Pala system, several avenues for future work have been identified to further enhance its security, reliability, and performance:

1) Enhanced Security and Authentication: While the current system incorporates standard security practices, future iterations should explore more advanced security and authentication mechanisms. This includes implementing JSON Web Tokens [7] (JWT) for secure information exchange between parties, enabling stateless authentication and improving API security. Additionally, the integration of zero-knowledge proofs (ZKPs) could significantly enhance voter privacy and the verifiability of the election process without revealing sensitive information, adding a layer of cryptographic assurance to the system's integrity.
2) Improved Logging and Diagnostics: To facilitate rapid diagnosis and resolution of issues in a production environment, the logging infrastructure of Pala can be significantly enhanced. Implementing structured logging with detailed context (e.g., request IDs, user information, timestamps, error codes) across all components (Golang backend, Traefik, database) will provide better visibility into system behavior. Centralized log aggregation and analysis platforms can be integrated with Grafana to create dashboards specifically for monitoring system health, identifying anomalies, and tracing the flow of requests, empowering software administrators to quickly pinpoint and address problems.
3) Database Redundancy and Latency Reduction: To further improve reliability and reduce read latency under heavy load, particularly during vote tabulation and result querying phases, implementing database redundancy using PostgreSQL's replication features is crucial. Establishing read replica nodes that synchronize with the primary database will allow read traffic to be distributed, significantly reducing the load on the primary node and decreasing response times for read-heavy operations. This distributed database approach enhances both the availability and performance of the system, ensuring that critical election data remains accessible and queries are processed efficiently even under peak demand.These planned enhancements aim to evolve Pala into an even more robust, secure, and highly available election management system capable of meeting the stringent requirements of large-scale democratic processes.

References

[1] "Golang." Mar. 2020.
[2] "Web Base Online Election Management Systems: Technical Review," *ResearchGate*, Oct. 2024, doi: 10.48175/IJARSCT-12025.
[3] "Gin-Gonic/Gin." Mar. 2025.
[4] A. Nagelli and D. N. K. Yadav, "Efficiency Unveiled: Comparative Analysis of Load Balancing Algorithms in Cloud Environments," *International Journal of Information Technology and Management*, vol. 18, no. 2, pp. 101–104, Aug. 2023, doi: 10.29070/pb64mp50.
[5] "Tutorial: Developing a RESTful API with Go and Gin - The Go Programming Language."
[6] P. G. D. Group, "PostgreSQL." Apr. 2025.
[7] E. J. Sebes and E. Perez, "A New Architecture for Trustworthy Voting Systems."