International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Scalable approach for Distributed File Processing using Spring, Zookeeper, and Docker

Arjun Reddy Lingala

arjunreddy.lingala@gmail.com

Abstract

In modern distributed systems, handling large-scale data efficiently is a key challenge, especially when dealing with structured and unstructured files stored in the Hadoop Distributed File System (HDFS) [1]. This paper presents an API- based solution using the Spring framework to process files in distributed file system, transforming them based on specific busi- ness requirements and storing the results back into distributed storage. The proposed architecture ensures high availability, fault tolerance, and efficient workload distribution through the integration of Apache Zookeeper [2] for consensus management and Docker [6] for containerized execution. We have distributed processing frameworks like Spark [8], which cannot be used in some cases where a certain process requires installing a software which cannot be done in distributed file system for security reasons. Approach discussed in this paper leverages the parallel execution of multiple Spring-based microservices, each deployed as independent Docker [6] containers, allowing for scalable and efficient processing. More instances of the Spring application can run simultaneously, ensuring that files are processed in a distributed manner to maximize throughput. The API facilitates seamless interaction with the HDFS [1] cluster, enabling efficient read, transformation, and write operations. To ensure coordination among instances, Apache Zookeeper [2] is used to manage leader election, task allocation, and synchronization, preventing conflicts and ensuring load balancing across nodes. The parallel processing workflow significantly improves the performance and resilience of the system. By running multiple instances in a containerized environment, our solution dynamically scales based on workload demands. Additionally, Zookeeper [2] ensures that processing tasks are distributed optimally, preventing redundant operations and maintaining system consistency. The paper provides a solution that demonstrates reduced processing time and improved fault tolerance compared to traditional single- instance processing methods. Through this paper, we highlight the benefits of combining Spring Boot [3], HDFS [1], Docker [6], and Zookeeper [2] for scalable and efficient distributed file processing.

Keywords: Spring, Docker, Distributed processing, distributed storage, HDFS, Zookeeper, Consensus, Coordination, Containerization, REST API, Monitoring, Logging

I. INTRODUCTION

With the rapid expansion of data in modern distributed systems, efficiently processing large-scale

datasets with mini- mal latency has become a challenge. The HDFS [1] provides a reliable and scalable storage solution for vast amounts of structured and unstructured data, but efficiently processing these files while ensuring high availability, fault tolerance, and scalability is key to any successful application and remains complex. Traditional single system architectures often face performance bottlenecks, not using resources effectively, and can cause single points of failure. To overcome these challenges, this paper proposes a distributed API-based solution using the Spring framework to read, transform, and write files in HDFS [1] while leveraging containerization and consensus mechanisms for efficient parallel execution.

The system discussed in this paper utilizes Spring Boot [3] to develop a RESTful [9] API that enables interaction with HDFS [1], facilitating data retrieval, transformation, and stor- age. To gain scalability and performance, multiple instances of the Spring Boot microservices run in parallel as Docker [6] containers, ensuring dynamic scaling, and fault isolation. This containerized deployment strategy enables high availability and resilience, preventing failures from impacting the overall processing workflow. A critical component of this architecture is Apache Zookeeper [2], which manages distributed consensus, coordination, and synchronization among multiple Spring Boot [3] instances. In a highly distributed environment, maintaining consistency and preventing redundant file processing is essential. Zookeeper [2] ensures leader election, task allocation, and workload balancing, preventing duplicate operations and improving fault tolerance. By distributing processing tasks dynamically across instances, the system optimizes resource utilization and ensures uninterrupted execution. Parallel ex- ecution using multiple instances is the main theme of this approach, significantly improving throughput and reducing execution time. Running multiple Spring Boot instances in Docker [6] containers allows for distributed task execution, maximizing efficiency compared to traditional single-instance application, which lack scalability and resource utilization. Docker enhances system adaptability by providing isolated environments, rapid deployment, and streamlined microservice management, making the system highly flexible in handling variable workloads.

II. SYSTEM ARCHITECTURE

The proposed system is designed to enable efficient, scalable, and fault-tolerant file processing within the HDFS [1] using microservices based approach. The architecture leverages a Spring Boot [3]-based API that facilitates file reading, trans- formation, and writing operations in HDFS [1]. Additionally, it employs Docker [6] for containerized execution and Apache Zookeeper [2] for consensus management, ensuring seamless parallel execution of multiple Spring Boot instances.

A. Components

The architecture consists of multiple key components that work together:

1) Spring based API: A RESTful API [9] developed using Spring Boot [3] serves as the core interface for handling file processing requests providing endpoints for initiating file processing, retrieving status, and monitoring execution. Spring boot applicaiton manages interactions with HDFS [1], including file retrieval, processing, and storage by supporting sync and async API calls optimizing request handling.

2) File System: Hadoop distributed file system or any object oriented storage like Amazon S3 can be used to store raw input files, and final output files from the API providing high throughput data access across all distributed nodes. The system reads files from distributed storage, processes them, and writes

the results back to distributed storage HDFS [1] or S3 or any similar distributed system.

3) Containerization: Multiple Spring Boot [3] instances run in separate docker containers enabling parallel processing and each instance is responsible for handling a subset of file processing requests. The containerized approach ensures portability, fault isolation, and efficient resource utilization and containers can be dynamically scaled based on workload demand.

4) Consensus: Consensus is very important in any distributed application which ensures proper coordination among multiple Spring Boot [3] instances and manages leader election to designate a primary instance for task distribution. Apache Zookeeper [2] is most commonly used consensus tool which synchronizes task execution to prevent duplicate processing and ensures fault tolerance by detecting failures and redistributing tasks as needed.

5) Load Balancing: Load Balancer dynamically assigns tasks across multiple Spring Boot instances to ensure balanced processing and tracks instance workloads and optimally distributes file-processing jobs. It also prevents overloading of individual instances, ensuring consistent system performance.

B. Execution Workflow

The system follows a structured workflow to ensure efficient and parallelized file processing in HDFS [1] and it is important to have these steps execute in order for file processing. Few examples of scalable file processing include converting Word documents to PDFs or appending summary content to the files etc.

1) File Retrieval: The Spring Boot [3] API exposes REST [9] endpoints for users or automated services to initiate file processing requests. When a request is received, an API instance connects to HDFS [1], retrieves the specified file, and temporarily stores it in memory or local cache for further processing.

2) Task Allocation: To avoid redundant processing and ensure an even workload distribution, Apache Zookeeper [2] handles task coordination. Few key aspects involved in task allocation and coordination process are *Leader Election* – one Spring Boot instance is designated as the primary task manager, *Task Distribution* – the leader assigns file processing tasks to available instances based on workload and resource capacity, *Failure Handling* – If an instance crashes, Zookeeper[2] automatically redistributes the task to another active in- stance. Instances communicate through API calls to fetch tasks from the leader.

3) Parallel Processing: Once tasks are assigned, multiple Spring Boot instances running as Docker [6] containers execute processing in parallel where each instance reads its assigned file from distributed storage, and the file undergoes appropriate transformations and changes (converting Word to PDF, appending Summary of the document etc.) and the file is written to temporary storage before writing back to distributed storage. After the file is written to temporary storage it is copied over to destination path on HDFS [1]

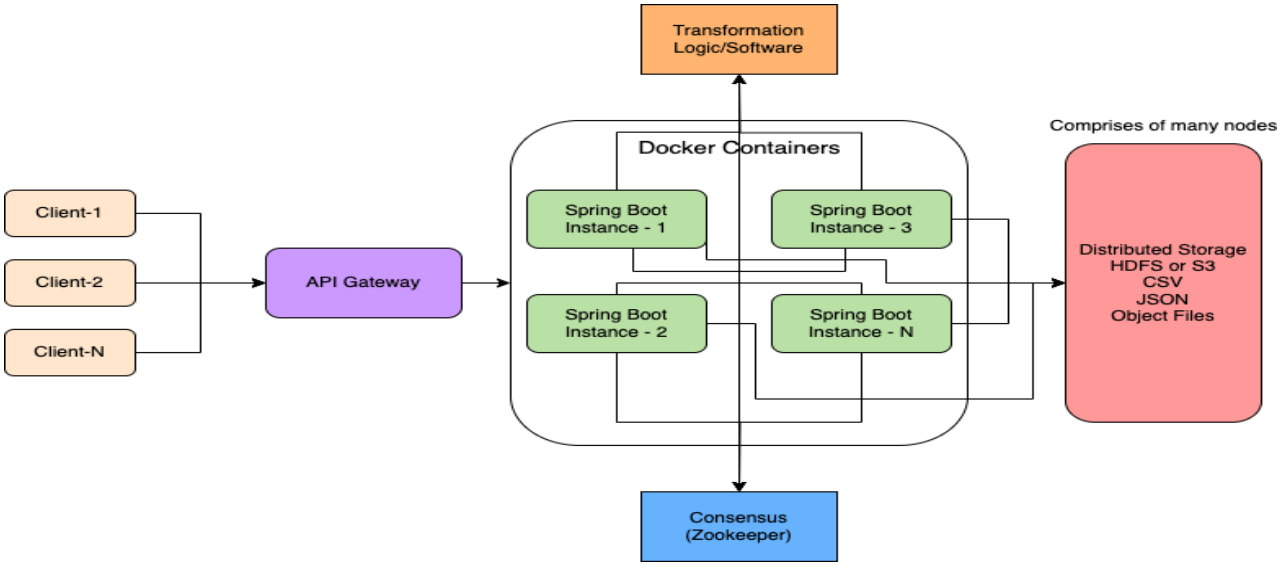

Fig. 1. File Processing Workflow

III. IMPLEMENTATION

The implementation of the proposed system is structured to achieve efficient, scalable, and fault-tolerant file processing within a distributed environment. It leverages Spring Boot [3] to develop a RESTful API [9] for interacting with Hadoop Distributed File System (HDFS), ensuring seamless read, trans- formation, and write operations. To manage task coordination and consensus across multiple parallel processing instances, Apache Zookeeper [2] is utilized, enabling synchronization and failure recovery. The deployment of the system is facilitated through Docker [6], which encapsulates each processing instance, allowing dynamic scaling and load balancing. The system architecture ensures high availability and efficiency by executing multiple processing tasks concurrently across more than ten Spring Boot instances, with API-driven interactions governing workflow execution.

The system exposes a RESTful API [9] developed using Spring Boot to initiate file processing tasks. These API end- points facilitate the reception of client requests, parameter validation, and the delegation of tasks to the processing layer. Apache Zookeeper [2] is employed to manage leader election and task distribution among processing instances. One instance is dynamically selected as a leader, which assigns tasks to worker instances based on availability and load conditions. Zookeeper [2] watches are configured to detect instance fail- ures, ensuring automatic task reassignment and fault tolerance. To achieve parallel execution, multiple Spring Boot instances operate as independent processing nodes, each executing a subset of the file transformation workflow. Each instance retrieves a designated file segment from HDFS [1], applies the necessary transformations, and writes the processed data back to HDFS. The use of dockerized instances ensures efficient workload distribution and fault isolation. Once processing is complete, the transformed data is stored back into HDFS. The system ensures that each processed file is properly indexed and versioned to maintain data integrity. Concurrent write operations are managed through Zookeeper [2] coordination, preventing conflicts and ensuring consistency.

Each instance of the Spring Boot [3] application is deployed as a Docker container, encapsulating dependencies and run- time configurations. The application can be deployed across different

environments without compatibility issues achieving portability. Any instance failure does not affect the overall system, as failed containers can be restarted independently achieving fault tolerance. The system utilizes Docker Compose or Kubernetes [7] to manage multiple Spring Boot instances. Additional containers can be instantiated dynamically based on processing demand, ensuring optimal resource utilization. New containers are deployed when processing demand increases achieving automatic scaling, tasks are evenly distributed across instances to prevent bottlenecks achieving load balancing.

IV. MONITORING AND LOGGING

To maintain operational efficiency, Prometheus and Grafana[5] are integrated for continuous monitoring of system performance metrics. These tools enable the collection and visualization of data related to API request throughput and latency, instance health and availability, resource computation that includes CPU, memory etc. The system employs the ELK Stack (Elasticsearch, Logstash, Kibana) [4] to aggregate and analyze logs from multiple processing instances which ensures provides visibility into the execution of file processing tasks for real time log retrieval use cases, facilitates rapid debugging of failures and anomalies for error detection and analyzing historical data.

Task allocation among worker instances is dynamically managed to ensure even distribution of workload. Apache Zookeeper [2] monitors instance health and redistributes tasks accordingly to prevent overloading any single node. Concur- rent data access can be implemented to reduce file retrieval time and frequently accessed files can be cached to reduce redundant disk I/O operations. System ensures automatic failure detection and recovery by leveraging Zookeeper [2] for failure detection and Kubernetes [7] for containerized self healing.

V. CONCLUSION

The proposed system presents a scalable, efficient, and fault- tolerant approach for distributed file processing in large-scale computing environments. By integrating a Spring Boot [3]- based API with Hadoop Distributed File System (HDFS), the system ensures seamless read, transformation, and write operations on distributed data. This type of system is helpful in cases where transformations require installation of software which cannot be done in distributed environments like HDFS or S3 for security and licesing reasons. Few examples of such transformations include converting Word documents into PDFs, adding summary based on the file content etc. The parallel execution of file processing tasks is achieved through Dockerized Spring Boot instances, enabling dynamic scaling based on workload demands. The usage of Apache Zookeeper [2] plays a pivotal role in ensuring task coordination, leader election, and fault tolerance, effectively preventing resource contention and guaranteeing high availability of processing nodes. The system architecture is designed to efficiently handle large datasets by executing multiple file-processing tasks concurrently across more than ten Spring Boot instances. The RESTful API framework enables seamless integration with external applications, allowing on-demand file processing through API calls. Each instance functions as a worker node that independently executes processing steps while remaining synchronized through Zookeeper [2] watches. The use of container orchestration tools such as Docker Compose or Kubernetes [7] further improves load balancing, automatic scaling, and instance recovery mechanisms, ensuring uninterrupted system operation.

REFERENCES

[1] K. Shvachko, H. Kuang, S. Radia and R. Chansler, "The Hadoop Distributed File System," 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), Incline Village, NV, USA, 2010, pp. 1-10, doi: 10.1109/MSST.2010.5496972.

[2] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait- free coordination for Internet-scale systems," 2010 USENIX Annual Technical Conference (USENIX ATC), Boston, MA, USA, 2010, pp. 145-158

[3] Pivotal Software, Inc., "Spring Boot Reference Documentation," Spring, 2020. [Online]. Available: https://docs.spring.io/spring- boot/docs/current/reference/html/.

[4] Elastic, "The Elastic Stack: Elasticsearch, Kibana, Beats, and Logstash," Elastic, 2020. [Online]. Available: https://www.elastic.co/guide/index.html.

[5] M. S. Aslan, M. A. Salahuddin, and R. H. Glitho, "Towards a Framework for Monitoring and Analyzing High Performance Computing Environ- ments Using Kubernetes and Prometheus," 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Sydney, NSW, Australia, 2019, pp. 1-6

[6] S. Singh and N. Singh, "Containers and Docker: Emerging roles and future of Cloud technology," 2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), Bangalore, India, 2016, pp. 804-807.

[7] P. J. Koopman and A. Chakravarty, "Kubernetes Architecture, Best Practices, and Patterns," 2022 IEEE International Conference on Cloud Engineering (IC2E), San Francisco, CA, USA, 2022, pp. 1-10.

[8] M. Zaharia et al., "Spark: Cluster Computing with Working Sets," Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10), Boston, MA, USA, 2010, pp. 10-10.

[9] M. Baez et al., "REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices," Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), Austin, TX, USA, 2016, pp. 778-789

[10] Amazon Web Services, Inc., "Amazon Simple Storage Service Documentation," Spring, 2021. [Online]. Available: https://docs.aws.amazon.com/s3/.

[11] S. C. Cahng and C. K. Lo, "A Consensus-Based Leader Election Algorithm for Wireless Ad Hoc Networks," 2012 IEEE International Conference on Communications (ICC), Ottawa, ON, Canada, 2012, pp. 1-5.