Log Sanitization and Analysis System Design

Author : Suresh Bysani

Background

IRCTC operates a server infrastructure using a mixed model of forked and threaded processes to handle user requests. However, the logging system used by IRCTC is rudimentary and results in jumbled and garbled logs, making it challenging to analyze and troubleshoot issues. The task is to design a Log Sanitization and Analysis System that can cluster logs of corresponding user requests together in chronological order. The system should provide log statistics and insights, such as the number of active threads, maximum concurrent threads, and thread lifetimes. By improving log readability and offering valuable insights, the system aims to enhance troubleshooting and performance monitoring for IRCTC's server infrastructure.

Goals:

- Sanitization: The system should sanitize the garbled log files and cluster logs of corresponding user requests together in chronological order for easier analysis and troubleshooting.
- Log Statistics: Provide comprehensive log statistics, including the number of active threads, thread IDs, process IDs, and references to the log files, to identify patterns, performance bottlenecks, and potential issues within the server infrastructure.
- Basic API: Expose an API to retrieve basic log statistics within a specified time range, including the number of active threads, their IDs, process IDs, and references to the log files.
- Bonus APIs: Provide additional APIs to retrieve advanced log statistics, such as the highest count of concurrent threads and the corresponding timestamp, and the average and standard deviation of thread lifetimes.
- Scalability: Design a scalable system capable of handling a large volume of log files, efficiently processing them to provide timely log statistics, and scaling horizontally to accommodate increasing log file sizes and user traffic.

Non-Goals:

- Real-time Log Streaming: The system does not aim to provide real-time log streaming capabilities. It focuses on processing and analyzing existing log files.
- Log Filtering: The system does not include advanced log filtering capabilities based on log content or specific criteria.
- Log Retention: The system does not handle long-term log retention. It focuses on log sanitization and analysis rather than long-term storage.

- Extensive Log Search: The system does not provide an extensive log search functionality. It primarily focuses on log clustering and basic log statistics retrieval.

We will talk about how to address the non goals if needed at the end of the document.

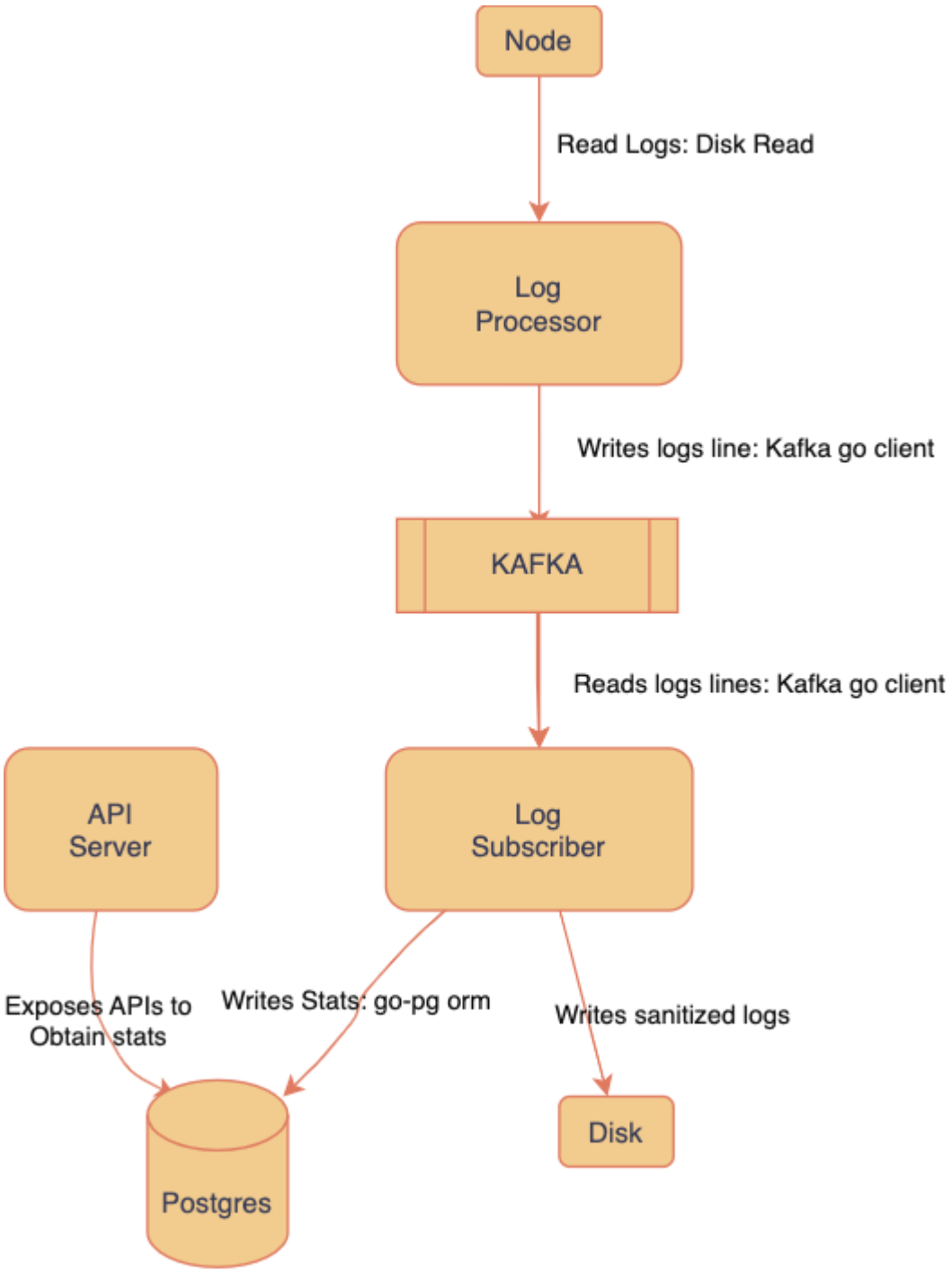
High Level Design

Architecture

The high-level design of the system follows a microservices architecture, consisting of several loosely coupled services that work together to achieve the desired functionality.

The system is composed of the following key components:

1. Log Processor Microservice: The Log Processor microservice is responsible for reading the log files, parsing the log entries, and sanitizing the logs by clustering them based on the corresponding user requests. It processes the logs in a distributed manner, using parallel processing techniques to handle large volumes of log data efficiently. The Log Processor can scale horizontally by adding more instances to handle increased workloads.
2. Log Subscriber Microservice: The Log Subscriber microservice consumes the sanitized logs produced by the Log Processor. It further processes the logs to extract relevant information and aggregates the log statistics, such as the number of active threads, thread IDs, process IDs, and references to log files. The Log Subscriber stores the processed log statistics in a persistent storage system, such as a database, for easy retrieval and analysis.
3. API Server: The API Server provides a RESTful API interface to expose the log statistics and enable clients to retrieve the desired information. It acts as a gateway for external systems to interact with the log data and retrieve specific log statistics based on various queries. The API Server communicates with the Log Subscriber microservice to fetch the relevant log data and transforms it into a suitable response format for the clients.
4. Database: The system utilizes a database to store the sanitized logs and log statistics. The Log Subscriber microservice persists the processed log statistics in the database, which allows for efficient querying and retrieval of log data. The choice of the database technology depends on the specific requirements, scalability needs, and performance considerations. For now we propose to use Postgres SQL for simplicity of this assignment. In the real world, we will probably choose OLAP.
5. Kafka: This is used as a message queue to capture the processed log lines from log processor service. The log subscriber service will consume them and prepare the sanitized logs and stats.


Node
Read Logs: Disk Read
Log Processor
Writes logs line: Kafka go client
KAFKA
Reads logs lines: Kafka go client
API Server
Log Subscriber
Exposes APIs to Obtain stats
Writes Stats: go-pg orm
Writes sanitized logs
Postgres
Disk

Key Design Decisions

1. Microservices Architecture: The decision to adopt a microservices architecture allows for independent scaling, deployment, and maintenance of the different components of the system. It promotes loose coupling, flexibility, and ease of development for each microservice.
2. Distributed Processing: The use of parallel processing techniques in the Log Processor microservice enables efficient handling of large log file volumes by leveraging multiple threads or processes to process log entries concurrently. This approach improves performance and reduces processing time.
3. Asynchronous Communication: The integration of a message queue, such as Kafka, enables asynchronous communication between the Log Processor and Log Subscriber microservices. This decoupling ensures fault tolerance, scalability, and resilience in the system.
4. Database Choice: The selection of an appropriate database technology, such as PostgreSQL, ensures efficient storage and retrieval of log data and log statistics. The choice of the database depends on factors like data volume, performance requirements, and scalability considerations.

Alternatives Considered

During the design process, several alternative approaches were considered for different components of the system. These alternatives were evaluated based on their compatibility with the requirements of the assignment, as well as their suitability for future scalability and maintainability. Some of the key alternatives that were considered include:

Database Alternatives:

While PostgreSQL was chosen as the primary database for this assignment due to its support for SQL queries and ease of integration with the chosen technology stack, alternative databases such as Cassandra and data warehouses like Redshift, BigQuery, or Druid were also evaluated.
However, considering the volume and nature of the log data in this scenario, PostgreSQL was deemed sufficient for handling the data and executing the required SQL queries.
The decision to use PostgreSQL allowed for leveraging existing SQL knowledge and tools without the need for additional complex setups.
For a real world scenario a Data warehouse solution like (Redshift, Snowflake) is probably preferred if complex queries are a requirement, Casandra can be preferred if there is a high ingestion rate. Casandra can scale well for ingestion since its distributed.

Message Queue Alternatives:

While Google Pub/Sub and Amazon Kinesis are both viable alternatives for message queueing, Kafka was chosen as the message queue for log processing in this assignment.

Kafka offers several advantages that make it a better fit for the given requirements.Firstly, Kafka is an open-source distributed streaming platform designed for high throughput and fault-tolerance. It provides durability and reliability in handling large volumes of data, which is crucial for processing log data efficiently.

Language Choice:

Golang was chosen as the primary programming language for the microservices due to its efficiency, performance, and familiarity. I code in Golang on a day to day basis. While other languages like Java or Python could have been considered especially when spark is in picture. Golang's concurrency features and ease of building scalable systems made it a suitable choice for handling concurrent log processing and microservice development.

Batch Processing vs. Stream Processing:

Batch processing was chosen over stream processing for log sanitization due to the nature of the assignment and the volume of log data involved. While stream processing frameworks like Apache Flink or Apache Spark Streaming could provide real-time log processing capabilities, the current requirements focused on batch processing to sanitize logs and generate statistical information. However, the system is designed in a modular way that allows for easy extension to incorporate stream processing if future requirements demand real-time processing.

Detailed Design

Log Processor:

1. The Log Processor is responsible for reading log files from the input directory and writing them to Kafka. It follows a parallel processing approach by utilizing multiple goroutines, with each goroutine handling a specific file.
2. The number of goroutines is determined by the configured maximum number of files per batch, allowing for efficient parallel processing and scalability.
3. Each goroutine reads the log file line by line and writes the lines to a buffered channel. The size of the buffered channel controls the number of lines that can be processed concurrently across all goroutines.
4. Once all lines are written to the buffered channel, a separate goroutine is responsible for consuming the lines from the channel and writing them to the Kafka topic.
5. This service ensures that log lines from different files are partitioned by the key "processid-threadid" to maintain the order within each thread while allowing for parallelism across different threads.

Log Subscriber:

1. The Log Subscriber acts as a Kafka consumer, subscribing to the log topic and receiving log messages from Kafka.
2. It processes the log messages, performs sanitization if required, and generates log statistics needed for the API responses.
3. The Log Subscriber is responsible for managing the state of active threads and processing the log messages in a way that ensures the chronological order of logs for each thread.
4. Sanitized log files are generated, containing the log lines for each thread in sequential order, facilitating easy analysis and debugging.
5. The log statistics, such as the number of active threads, thread IDs, process IDs, and file references, are calculated and stored for efficient retrieval by the API Server.

API Server:

1. The API Server provides a RESTful API interface for accessing log statistics and information.
2. It handles incoming HTTP requests and interacts with the Postgres database to retrieve the required data for the requested endpoints.
3. The API Server exposes endpoints for retrieving basic statistics, such as the number of active threads, thread IDs, process IDs, and file references within a specified time range.
4. It also provides endpoints for retrieving the maximum count of concurrent threads observed and the corresponding timestamp, as well as the average and standard deviation of thread lifetimes.
5. The API Server is designed to be scalable and handles concurrent requests efficiently to provide timely responses.

Kafka:

1. Kafka is used as a message queue. It is used for process the log lines from log processor service.
2. A kafka image from confluent is used. The topic "processor-messages" is created from log processor service.

Schema

```
CREATE TABLE IF NOT EXISTS log_lines (
    process_id VARCHAR(255),
    thread_id VARCHAR(255),
    timestamp TIMESTAMPTZ,
    timestamp_seconds BIGINT,
    log_message TEXT,
    PRIMARY KEY (process_id, thread_id, timestamp, timestamp_seconds)
);
```

The log_lines table has the following columns:

- process_id: Represents the unique identifier of the process associated with the log line.
- thread_id: Represents the unique identifier of the thread associated with the log line.
- timestamp: A TIMESTAMPTZ (timestamp with time zone) column that stores the timestamp of the log line. This stores the complete timestamp with milliseconds precision.
- timestamp_seconds: A BIGINT column that stores the timestamp in seconds for easier querying and analysis.
- log_message: A TEXT column that contains the actual log message.

The primary key of the log_lines table is defined as a combination of the process_id, thread_id, timestamp, and timestamp_seconds columns. This ensures that each log line is uniquely identified and allows for efficient retrieval and indexing of the log data.

Third Party Software (OSS)

Libraries used

Open-Source Library	**Purpose**	**Services Used**
go-pg	Database ORM for PostgreSQL	apiserver, logprocessor, logsubscriber
golang/glog	Logging library	apiserver, logprocessor, logsubscriber
echo/v4	HTTP framework for building web APIs	apiserver
viper	Configuration management and parsing	apiserver, logprocessor, logsubscriber
confluentinc/confluent-kafka-go	Kafka client library for Go	logprocessor, logsubscriber

Infra Components Used

1. Kafka
2. Postgres

Next Steps:

Scaling to 100x

Scaling the log sanitization project to handle a 100x increase in log volume requires careful consideration of various components within the system. The following aspects should be addressed to ensure the system can handle the increased load effectively:

Container Orchestration with Kubernetes

1. **Robust Container Management**: While Docker Compose simplifies container deployment on a single host or small cluster, Kubernetes offers more advanced container management capabilities. It provides features like self-healing, rolling updates, and declarative configuration management, ensuring the reliability and consistency of the log sanitization process.
2. **Automatic Scaling**: Kubernetes allows for automatic scaling of the log sanitization microservices based on resource utilization and demand. It leverages features such as Horizontal Pod Autoscaling (HPA) and Cluster Autoscaling to dynamically adjust the number of running instances, ensuring optimal resource utilization and performance.
3. **Load Balancing and Service Discovery**: Kubernetes provides built-in load balancing and service discovery mechanisms. It intelligently distributes incoming requests to the log sanitization microservices across multiple instances, preventing any single instance from becoming a bottleneck. This ensures high availability and efficient utilization of resources.
4. **Fault Tolerance and High Availability**: Kubernetes offers fault tolerance mechanisms such as replica sets and pod rescheduling. In the event of a node failure or pod crash, Kubernetes automatically restarts the affected containers on healthy nodes, ensuring the continuity of log sanitization without interruption.

Database improvements:

Table Partitioning:

In the current example we are not partitioning the table at all. As the volume of data increases, we must partition the SQL table for scalability and performance of the queries. The proposal is to add time range based partitioning on the column timestamp_seconds.

```
CREATE TABLE log_lines (
  id SERIAL PRIMARY KEY,
  thread_id INT,
  process_id INT,
  timestamp_seconds BIGINT,
```

```
  -- Other columns...
)
PARTITION BY RANGE (timestamp_seconds);

-- Define partitions for each month
CREATE TABLE log_lines_partition_202301 PARTITION OF log_lines FOR
VALUES FROM (1640995200) TO (1643673600);
CREATE TABLE log_lines_partition_202302 PARTITION OF log_lines FOR
VALUES FROM (1643673600) TO (1646092800);
-- Add more partitions for subsequent months as needed

-- Create a default partition for all other values
CREATE TABLE log_lines_partition_default PARTITION OF log_lines DEFAULT;
```

With this modified schema with partitions on timestamp_seconds, the data related to a specific month is stored in the specific partition table.

The typical user behavior is to search for logs or logs stats in a specific time range window. So dividing the data into such partitions will reduce the full table scan and it will resort the scan to specific partitions.

Index Optimization:

Our frequently executed queries has timestamp_seconds, thread_id, process_id columns used in the WHERE, JOIN, and ORDER BY clauses. Create indexes on timestamp_seconds, thread_id, and process_id columns.

```
CREATE INDEX idx_log_lines_timestamp ON log_lines (timestamp_seconds);
CREATE INDEX idx_log_lines_thread_id ON log_lines (thread_id);
CREATE INDEX idx_log_lines_process_id ON log_lines (process_id);
```

Data Archiving:

SQL does not scale well if the volume of data is too much. SQL is fundamentally not distributed. So its important to keep the size of our DB intact. Consider keeping data only for the last 1 year or 6 months. Archive older data to data warehouses or cheaper storage like S3.

Sharding:

If the volume of data continues to grow beyond the capacity of a single database server, consider implementing sharding. Distribute the data across multiple database servers based on a shard key, such as a hash of the process_id. Each shard would have its own log_lines table, and queries would need to be executed across multiple shards when necessary.

Executing such queries will bring in its own complexity. This is the case for using a distributed query engine as opposed to something like Postgresql.

Database Replication:

Set up database replication to create replicas of the log_lines table for read scalability and redundancy. Replication can be synchronous or asynchronous, depending on the requirements. Read queries can be directed to the replicas to distribute the read load and improve overall performance.

Materialized views:

Most of the data that we need is stats. If the requirements are relaxed to make sure we are ok with stats of log data where the freshness is of yesterday, we can create required materialized views and refresh them once a day. The APIs can be directly answered from the MVs.

Data Compression:

Implement data compression techniques at the database level to reduce the storage requirements. Utilize database-specific compression features or external tools to compress the log data. Compressed data takes up less storage space, allowing you to store more data within the available resources. The requirement to store raw log lines along with stats db also seems very counter intuitive. We can revisit that.

Improve Durability and Reliability

To enhance the durability and reliability of the log sanitization system, we can implement several measures to ensure data integrity and accessibility, especially when scaling to handle a 100x increase in log volume. Consider utilizing cloud storage services like Amazon S3 or Google Cloud Storage for storing the log files. These services provide high durability and redundancy, ensuring that the log data is protected against hardware failures, disk corruption, or accidental deletion. Cloud storage services also offer built-in mechanisms for data replication and backup, increasing the overall reliability of the system.

Kafka HotSpots

We are using a hash key based partitioning scheme. The hash key is (processId-threadId). So even if certain processes are publishing too many logs because it is receiving too many API calls, we are covered because not all logs from a given process reach the same partition. But if there are too many logs for a given thread in the process, this can create a hotspot if the consumer is slow in processing. To address this, we need to understand the names of these processes, we can create special partitions/topics for these processes.

Supporting Advanced Analytics

Supporting advanced analytics requires a robust and efficient search mechanism to enable complex search operations on the log data. One approach to achieve this is by implementing an inverted index using a search engine like Elasticsearch.
The inverted index is a data structure that maps each unique term in the log data to the list of documents (log lines) that contain that term. It allows for fast and efficient keyword-based searching and retrieval of relevant log lines. By indexing the log data in Elasticsearch, we can leverage its powerful search capabilities, including full-text search, filtering, faceted search, and aggregations.

With Elasticsearch, we can enable advanced search functionalities such as:

1. Full-text search: Perform keyword-based searches across the log data, including support for fuzzy matching, stemming, and relevance scoring.
2. Filtering: Apply filters to narrow down the search results based on specific criteria, such as timestamps, thread IDs, process IDs, or custom fields.
3. Faceted search: Retrieve aggregated information about the log data, such as the distribution of log lines by various dimensions like timestamps, thread IDs, or severity levels.
4. Aggregations: Perform data summarization and analysis on the log data, such as computing statistical metrics (e.g., average, maximum, minimum) or generating histograms.

By integrating Elasticsearch into the log sanitization system, we can enhance the analytics capabilities and enable more advanced querying and exploration of the log data. This allows users to gain deeper insights, detect patterns, and troubleshoot issues effectively.
Additionally, Elasticsearch provides horizontal scalability and distributed data storage, making it suitable for handling large volumes of log data. It can be configured as a cluster of nodes to ensure high availability, fault tolerance, and improved query performance.
With the integration of Elasticsearch, the log sanitization system can support advanced analytics use cases, empower users with powerful search capabilities, and facilitate in-depth log analysis and exploration.

Integrate Observability

To ensure the effective monitoring and observability of the log sanitization system, we can integrate various tools and techniques that provide insights into system performance, availability, and issues. The following are some recommended approaches:

Monitoring with Prometheus and Grafana

Integrate Prometheus and Grafana to collect and visualize system metrics. Prometheus is a popular open-source monitoring system that gathers metrics from various sources, including microservices, infrastructure components, and databases. Grafana, on the other hand, provides a rich set of visualization tools and dashboards to display these metrics in real-time. By instrumenting our microservices with Prometheus exporters and setting up Grafana

dashboards, we can monitor key performance indicators, such as throughput, latency, and resource utilization, enabling us to identify and troubleshoot any performance bottlenecks or anomalies.

Distributed Tracing with OpenTelemetry

Implement distributed tracing using OpenTelemetry to gain visibility into the flow of requests and transactions across microservices. OpenTelemetry is a vendor-neutral observability framework that provides a standardized approach to capture and propagate trace data. By integrating OpenTelemetry into our microservices, we can generate trace spans that capture the execution details of individual requests, enabling end-to-end visibility and analysis of the log sanitization process. This allows us to pinpoint performance issues, track request flows, and optimize system performance. Integrate Jeager as tracing backend. Grafana can be the single source of the truth to view traces as well.

Log Aggregation and Analysis

Afterall we are building the log aggregation and analytics platform. We should dog food our platform for logging.