Design Lab - Distributed Logger

Akshat Singh Rathore (19CS30003)

November 2023

1 Introduction

The Distributed Logger is designed to streamline log management and sharing within an organization. Its primary goal is to centralize and simplify the process of log management and sharing, making it more accessible and manageable. By capturing log outputs from a variety of applications, the system ensures a comprehensive collection of logging data. The use of Kafka as a data transfer layer not only boosts efficiency but also guarantees reliability and speed in log dissemination. This system is particularly beneficial for debugging and monitoring, offering a streamlined, unified view of logs from diverse sources. The emphasis on user-friendliness ensures that the tool can be easily adopted by different teams within an organization, enhancing collaborative troubleshooting and analysis efforts. Ultimately, the Distributed Logger stands as a solution to the complexities and scattered nature of traditional log management approaches, offering a consolidated, efficient, and agile logging ecosystem.

2 Architecure

The logger maintains a modular architecture with the following components:

Bytecode Logger

Purpose: To intrusively capture and log output statements directly from the application bytecode. **Implementation:** Uses Javassist for real-time bytecode manipulation. **Trigger:** Activates upon encountering ‘System.out.println‘ or similar output statements, logging the content and line number.

FileWatcher

Purpose: Monitors for new log entries, facilitating the transfer of log data. **Implementation:** Employs Java’s ‘WatchService‘ API to observe the ‘.dlogger‘ directory. **Trigger:** Responds to file creation or modification events in the monitored directory.

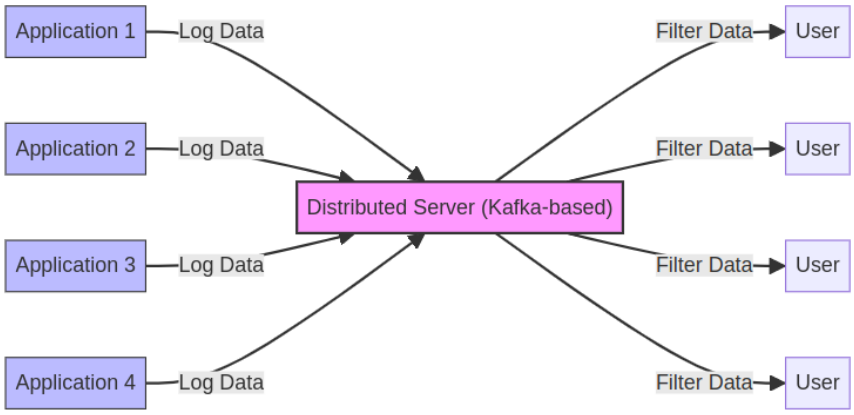

Figure 1: Basic illustration of working of the logger.

KafkaLogger

Purpose: Responsible for the efficient transmission of log data to Kafka topics. **Implementation:** Integrates with Kafka's producer API to send log messages. **Trigger:** Engages upon receiving new log data from the FileWatcher.

MainController

Purpose: Orchestrates the initialization and management of the logging environment. **Implementation:** Sets up KafkaLogger and FileWatcher, managing the logging lifecycle. **Trigger:** Initiates at system startup, configuring and starting the log monitoring process.

LogConsumer

Purpose: Enables users to retrieve specific log data from Kafka. **Implementation:** Utilizes Kafka's consumer API for filtered log retrieval. **Trigger:** User-driven, based on specified criteria like username and classname.

3 Flow of Logs

3.1 Log Generation

Applications generate logs through standard output mechanisms like `System.out.println` in Java.

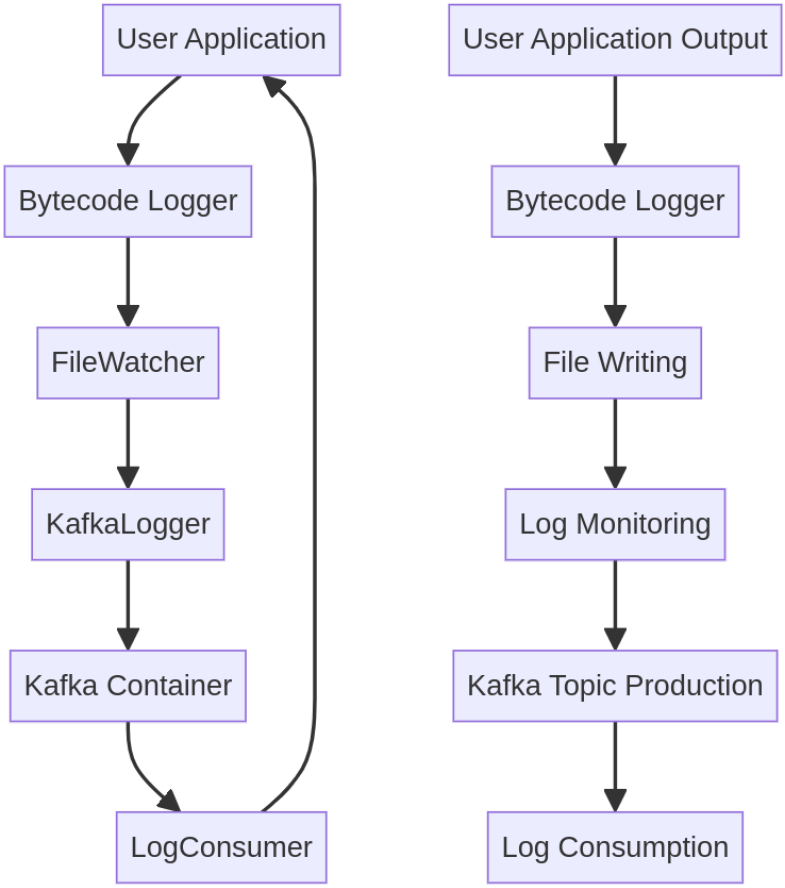

Figure 2: The flow of control among the components of the logger and their related activities are shown in the figure above.

3.2 Capture

The Bytecode Logger, once injected into the application, captures these print statements. It uses bytecode manipulation to intercept log messages and enriches them with metadata like line numbers.

3.3 Monitoring

The captured logs are written to hidden files in the `.dlogger` directory. FileWatcher continuously monitors this directory for new files or changes to existing files.

3.4 Kafka Production

Once FileWatcher detects changes, KafkaLogger takes over. It reads the new log data and produces it to specific Kafka topics, typically named after the user and class for easy identification.

3.5 Consumption

Logs are now available in Kafka topics and can be consumed. Users employ LogConsumer, specify the username and classname, and retrieve the relevant logs.

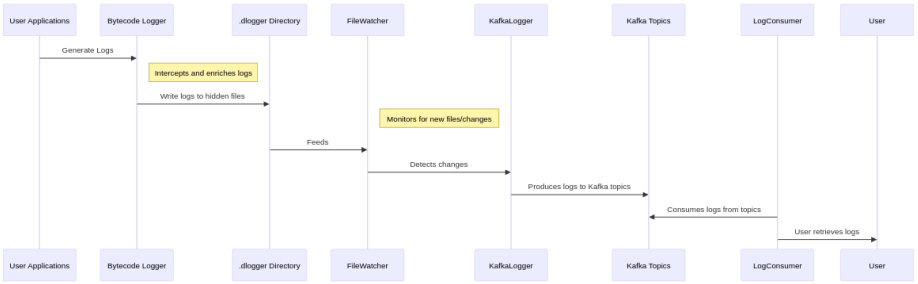

Figure 3: Sequence of messages is illustrated in the image above.

4 User Interactions

The user leverages the tool to have a global view of the logs of the entire organization queried through the username and the class name. Registering with the tool requires the user to have a unique username. This is later used to partition the logs. Now the user can view the logs of any user in the organization provided that the agent loaded. When viewing the logs, the user is asked for the username and the class whose logs the user wishes to view.

5 Design Choices

5.1 Why Kafka?

Kafka is selected for its exceptional ability to handle high-throughput, distributed data streaming, which is essential for reliable log transfer and management. Its advantages include:

- **Scalability:** Efficiently manages large volumes of data.
- **Fault Tolerance:** High availability and resilience to node failures.
- **Real-Time Processing:** Offers low-latency log delivery.

- **Durability:** Ensures reliable log storage.

These features establish Kafka as a robust and efficient backbone for the Distributed Logger system.

5.2 Advantages of Bytecode Manipulation

Bytecode manipulation provides significant advantages in logging:

- **Flexibility:** Enables dynamic modification of classes at runtime.
- **Granularity:** Captures detailed information, including specific method calls and line numbers.
- **Low Overhead:** Minimal impact on application performance.
- **Versatility:** Applicable across diverse Java applications.

This approach allows the Distributed Logger to capture detailed logs unobtrusively, maintaining application integrity and performance.

6 Challenges and Solutions

6.1 Scheduling

Challenge: Efficiently monitoring the `.dlogger` directory for new log entries.
Solution: Implementation of an optimized FileWatcher to periodically check the log directory, coupled with KafkaLogger for efficient log processing.

6.2 Duplication

Challenge: Avoiding duplication of log messages.
Solution: Utilization of unique consumer groups per user, ensuring singular processing of each log entry.

6.3 Scalability and Performance

Challenge: Handling large-scale log data without performance issues.
Solution: Kafka's distributed architecture underpins scalability, with potential for dynamic load balancing and data partitioning enhancements.

6.4 Locality Mapping

Challenge: Correctly mapping log messages to appropriate channels based on their origin.
Solution: Hashing the combination of user and class names to accurately direct logs to the correct Kafka topics.

7 Discussions

7.1 Effectiveness in Diverse Environments

The Distributed Logger is designed to adapt to various application environments, from small-scale applications to large, distributed systems. Its scalability, facilitated by Kafka, allows it to handle varying log volumes effectively.

7.2 Impact on Developer Workflow

Integrating the Distributed Logger into existing workflows offers developers a more streamlined approach to log management. However, the need for understanding bytecode manipulation and Kafka might require initial upskilling.

7.3 Security and Data Privacy

While the Distributed Logger centralizes log data, it raises questions about data security and privacy. Ensuring secure transmission and storage of logs, especially in sensitive applications, is a vital area for future development.

7.4 Comparison with Traditional Logging Systems

Traditional logging systems often lack the real-time processing and scalability offered by the Distributed Logger. However, they might be simpler to set up and use in less complex environments.

7.5 Future Enhancements

Future enhancements could include advanced data analytics capabilities, integration with AI for predictive analysis, and expanded customization options for different use cases.

8 Codebase

The link to the source code for the project can be found at: https://github.com/Akshat-Rathore/distributed-logger.git.

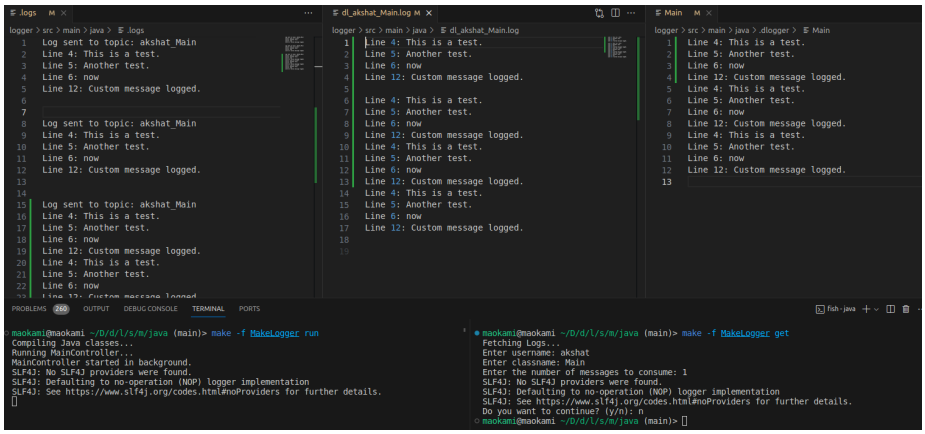

Figure 4: The figure above shows a user interacting with our logger. Content of the different forms of log is shown. The file .logs contains the logs of the logger itself, while the log in the middle is the end product generated from the query. The final log is the intermediary to produce the logs to the kafka brokers.