**Distributed Log Analyzer - Full Technical Documentation**

**Overview**

The **Distributed Log Analyzer** is a scalable, fault-tolerant system designed for analysing large log files in real-time. It leverages a distributed architecture with a **Coordinator** managing multiple **Worker Nodes** that parse, process, and compute various log metrics.

**Key Features:**

* **Efficient Log Parsing**: The system reads and processes log files efficiently by breaking them into smaller chunks and assigning them to workers.
* **Real-Time Metric Calculation**: Key metrics such as error rate, average response time, and request count are computed in real time.
* **Fault Tolerance**: If a worker node fails, its work is reassigned to other workers without interrupting the system.
* **Scalability**: The system can scale by adding more workers to handle larger log files, ensuring high availability and throughput.

**High-Level Diagram:**

**System Architecture**

Coordinator

(Manages Workers)

Worker 3

(Log Chunk)

Worker 2

(Log Chunk)

Worker 1

(Log Chunk)

**Key Components:**

1. **Coordinator**: The main controller of the system, which is responsible for:
   * Splitting log files into chunks.
   * Assigning tasks to workers.
   * Aggregating results from workers.
   * Handling worker failures and reassigning tasks if needed.
2. **Worker Node**: These are the distributed units that process chunks of log files. Each worker:
   * Reads a specific portion of the log file.
   * Extracts necessary data (timestamps, error levels, etc.).
   * Calculates relevant metrics.
   * Sends results to the Coordinator.
3. **Analyzer**: The real-time analysis engine that collects data from workers and computes:
   * Error rates
   * Average response times
   * Request counts
   * Resource usage patterns

**Installation Guide**

**Requirements:**

* Python 3.9 or higher
* Required Python packages (requirements.txt)
* Postman (for testing APIs)
* pytest (for unit testing)

**Setup Instructions:**

1. **Clone the repository**:
2. git clone <repository\_url>
3. cd log-analyzer-assessment
4. **Create a virtual environment**:
5. python -m venv venv
6. source venv/bin/activate # On Windows: venv\Scripts\activate
7. **Install dependencies**:
8. pip install -r requirements.txt
9. **Start the system**:
   * Launch the **Coordinator** on a specified port.
   * Start multiple **Worker Nodes** for processing log files.
10. **Run the system**:
    * Send log files to the Coordinator for processing.
    * The workers will process chunks of the file and send back results to the Coordinator.
11. **Run Tests**:
12. pytest test\_cases.py

**Design Decisions**

**1. Asynchronous Processing:**

* The system uses asyncio to perform asynchronous log processing. This enables concurrent processing of multiple log chunks, which improves efficiency and throughput.

**2. Distributed Workload:**

* Log files are split into manageable chunks, which are distributed to worker nodes. This design allows the system to scale efficiently, especially when dealing with large log files.

**3. Failure Handling:**

* If a worker node fails, the Coordinator detects this and reassigns the tasks of the failed worker to available workers. This minimizes downtime and ensures that the system remains operational even in the face of failures.

**4. Real-Time Analysis:**

* Real-time metrics are calculated based on the log entries as they are processed by workers. This ensures that the system provides up-to-date insights into system performance.

**Components**

**Worker Node**

class Worker:

def \_\_init\_\_(self, worker\_id: str, coordinator\_url: str):

self.worker\_id = worker\_id

self.coordinator\_url = coordinator\_url

async def process\_chunk(self, filepath: str, start: int, size: int) -> Dict:

# Process a chunk of the log file and return metrics

pass

async def report\_health(self) -> None:

# Send heartbeat to the coordinator to report worker status

Pass

**Responsibilities:**

* Read and parse the log file in chunks.
* Calculate metrics based on log entries.
* Send processed results to the Coordinator.

**Coordinator**

class Coordinator:

def \_\_init\_\_(self, port: int):

self.workers = {}

self.results = {}

async def distribute\_work(self, filepath: str) -> None:

# Split the log file and assign chunks to workers

pass

async def handle\_worker\_failure(self, worker\_id: str) -> None:

# Reassign work from failed worker

pass

**Responsibilities:**

* Manage worker nodes.
* Split log files into chunks and distribute them.
* Aggregate results from workers.
* Handle worker failure and reassign tasks.

**Analyzer**

class Analyzer:

def \_\_init\_\_(self):

self.metrics = {}

def update\_metrics(self, new\_data: Dict) -> None:

# Update the metrics with new data from workers

pass

def get\_current\_metrics(self) -> Dict:

# Return the current metrics calculated from processed logs

pass

**Responsibilities:**

* Calculate real-time metrics like error rate, response time, and request count.
* Update metrics as new data is processed.

**How It Works**

1. **Log File Processing**:
   * The Coordinator receives a log file, splits it into smaller chunks, and assigns these chunks to worker nodes for processing.
2. **Worker Node Processing**:
   * Each worker reads its assigned chunk, parses it for relevant information (e.g., timestamp, log level, metrics), and calculates performance metrics.
   * After processing, each worker sends the computed metrics to the Coordinator.
3. **Real-Time Analysis**:
   * The **Analyzer** class continuously updates the real-time metrics using the data provided by workers. These metrics include error rate, average response time, and resource usage patterns.
4. **Failure Recovery**:
   * If a worker fails, the Coordinator detects this failure, reassigns its tasks to other available workers, and ensures that log processing continues without interruption.

**Testing**

**Unit Tests:**

* **Test 1: Log Parsing**:
  + Verifies that logs are parsed correctly into the required fields (timestamp, level, message, metrics).
* **Test 2: Metric Calculation**:
  + Ensures that workers correctly compute the real-time metrics, such as error rates and average response times.

**Integration Tests:**

* **Test 3: Worker-Coordination**:
  + Verifies that workers can communicate with the Coordinator and report results correctly.
* **Test 4: Worker Failure Recovery**:
  + Simulates worker failure and checks that the Coordinator reassigns tasks appropriately.

**Testing Tools:**

* **Pytest** for unit and integration testing.
* **Postman** for testing API endpoints like log file uploads and processing requests.

**Real-Time Metrics**

**Key Metrics:**

1. **Error Rate**:
   * Measures the percentage of log entries marked as errors out of the total log entries processed.
2. **Average Response Time**:
   * Calculates the average time taken to process each log entry.
3. **Request Count**:
   * Tracks the number of requests logged per second.
4. **Resource Usage Patterns**:
   * Monitors CPU and memory usage while processing logs.

**Performance Results**

**Metrics Tracked:**

* **Processing Speed**: The system can process large log files quickly by distributing the workload across multiple worker nodes.
* **Memory Usage**: The memory consumption is minimized by processing log files in chunks.
* **Failure Recovery**: The system can recover from worker node failures in under a few seconds by redistributing tasks to healthy workers.

**Conclusion**

The **Distributed Log Analyzer** is a robust, scalable system capable of processing large volumes of log data efficiently while providing real-time insights and ensuring fault tolerance. The system can handle various operational challenges, including worker failures and large log file sizes, while delivering accurate metrics that are critical for monitoring system health.