Distributed DICOM Processing System (Design and Architecture)

Introduction

This document outlines the design and architecture of a distributed system for processing large batches of medical images (e.g., DICOM images) across multiple machines. The system is designed to ensure efficient task distribution, fault tolerance, logging, and scalability. It should handle multiple concurrent image uploads, perform distributed processing, and provide a way for users to retrieve the results.

1. Main Entry Point

1.1. Upload API

- **Upload Endpoint**: The system will feature a RESTful endpoint, `/upload`, allowing users to upload DICOM images. This endpoint will be accessible through a secure web portal.
- **File Validation**: The system will validate uploaded DICOM files. The validation will include:
    - Verification of the file format and structure.
    - Data integrity checks using checksums.
    - Validation of DICOM metadata to ensure it meets processing requirements.
- **Task Queue**: Once validated, the files will be queued for processing using a distributed queue system like **RabbitMQ** or **Kafka**. This mechanism ensures that tasks are processed in the order they are received and provides a buffer between the workload and the processing servers.

1.2. Task Queuing and Prioritization

- **Task Queuing**: Tasks will be queued based on defined priority (e.g., urgent images might have higher priority). The tasks will include the DICOM file information and necessary metadata for processing.
- **Prioritization**: Urgent tasks will be placed in a high-priority queue, while regular tasks will be queued normally. The system may also allow dynamic reordering of tasks based on user needs or system policies.

2. Distributed Processing System

2.1. Processing Server Design

- **Group of Workers**: The system will consist of a group of processing servers, or workers, responsible for processing the DICOM images. Each worker will run an instance of the processing service, capable of handling multiple tasks in parallel.
- **Load Balancing**: A load balancer (e.g., **NGINX** or **HAProxy**) will distribute tasks among the workers based on their current load (CPU, memory) and processing capacity. The load balancer will monitor the health of the workers and redirect tasks to the least loaded servers.

2.2. Scalability

- **Horizontal Scalability**: The system will be designed to allow the addition or removal of workers according to system demands. Using tools like **Kubernetes** for container orchestration, the system can automatically scale the number of workers based on current load.
- **Auto-scaling**: Kubernetes will automatically manage the scaling of workers in response to resource usage metrics such as CPU, memory, and queue latency.

3. Usage Limits and Resource Management

3.1. Resource Monitoring and Management

- **Resource Monitoring**: The system will use **Prometheus** to monitor resource usage on each worker, including CPU, memory, disk, and network. The collected metrics will be visualized in **Grafana**, with alerts configured to notify about potential overloads or failures in workers.
- **Usage Limit Management**: A resource management policy will prevent workers from being overloaded. If a worker exceeds a predefined resource usage threshold, the system will stop assigning new tasks to it until its load decreases.

3.2. Task Assignment Algorithm

- **Load-based Task Assignment**: The system will use a task assignment algorithm based on the current load of the workers. The algorithm will consider CPU usage, memory usage, and the number of ongoing tasks to assign new tasks to the most suitable worker.
- **Task Redistribution**: If a worker fails during task processing, the system will redistribute the task to another available worker, ensuring service continuity.

4. Database Integration

4.1. Result Storage

- **Distributed Database**: The results of the image processing will be stored in a distributed database, such as **Cassandra** or **MongoDB**. These databases are designed to handle large volumes of data and offer high availability and scalability.
- **Database Schema**: The database schema will be optimized for quick access to processed results, using data partitioning and replication to improve performance.

4.2. Worker Integration

- **Concurrent Writing**: Each worker will be responsible for writing the processing results to the distributed database. The database must handle concurrent writes from multiple workers without compromising data consistency.
- **Activity Logging**: All processing events, including failures and successes, will be logged in a centralized logging system. This logging will allow for monitoring system performance and conducting audits.

5. Information System for Users

5.1. User Interface

- **Secure Web Portal**: Users will access a web portal where they can upload images, check the processing status, and download the results. This portal will be designed to provide an intuitive and secure user experience.

5.2. Authentication and Access Control

- **Authentication**: A robust authentication system will be implemented, supporting various authentication methods, such as passwords, OAuth, and two-factor authentication (2FA).
- **Access Control**: The system will ensure that users can only access their data. Roles and permissions will be implemented to manage access to different functionalities within the system.

5.3. Database Interaction

- **Real-time Querying**: The web portal will provide real-time access to processing results, interacting directly with the distributed database to retrieve and display data efficiently.

6. Additional Considerations

6.1. Security and Privacy of Medical Data

- **HIPAA/GDPR Compliance**: The system will be designed to comply with medical data privacy regulations, such as HIPAA in the U.S. and GDPR in Europe.
- **Data Encryption**: All medical data will be encrypted both in transit and at rest using standards like AES-256 for at-rest encryption and TLS for in-transit encryption.

6.2. Logging Implementation

- **Centralized Logging**: Using the ELK Stack (Elasticsearch, Logstash, Kibana), a centralized logging system will be implemented to capture logs from all parts of the system, including the upload API, workers, and database.
- **System Health Monitoring**: The logging system will be configured to monitor the health of each system component and generate alerts in case of anomalies or failures.

6.3. Error Handling and Availability

- **Fault Tolerance**: The system will be designed to handle failures in individual servers without compromising overall availability. If a worker fails, its assigned tasks will be automatically redistributed to other workers.
- **Maintaining Availability**: The system will use redundancy in all critical components (workers, database, load balancers) to ensure high availability and avoid single points of failure.