Introduction

In today's digital landscape, the demand for video content continues to rise exponentially. With this surge in demand comes the need for efficient video processing systems capable of handling large volumes of video transcoding tasks. This document provides a detailed overview of such a system that we built, outlining its architecture, technologies utilized, challenges encountered, and proposed solutions.

System Overview

The video transcoding system serves as a pivotal intermediary between distributed storage services and end clients. Its primary function is to facilitate the transcoding of video content into various resolutions and bitrates, thereby optimizing the viewing experience for users with different bandwidth constraints. The system operates in a highly distributed manner, allowing for concurrent processing of multiple video files while dynamically scaling resources to meet fluctuating workload demands.

The system's architecture is designed to harness the benefits of distributed computing, including scalability, fault tolerance, and efficient resource utilization. The system handles fault detection and recovery, autoscaling, load balancing, and efficient handling of video segments during transcoding, thereby ensuring seamless operation and optimal performance under varying conditions.

Technology Stack

The technology stack powering the video transcoding system is carefully chosen to meet the diverse requirements of our video processing system:

- Java serves as the primary programming language, offering a robust ecosystem and extensive libraries that support our needs.
- JUnit and Maven are used for automated testing and streamlined project management, respectively.
- Docker is used for containerization, ensuring smooth and consistent deployment across all environments.
- Docker Swarm is used for container orchestration, allowing us to manage a complex system consisting of multiple nodes each running multiple containers.
- RabbitMQ handles messaging between system components in a distributed and reliable manner.
- FFmpeg is used for much of the transcoding logic, as it provides comprehensive multimedia processing capabilities.
- MongoDB serves as our database for storing logs, as its storage model aligns

with the type of data that we store.
- HLS and H264 Codec are utilized for streaming and video encoding, respectively, as they are widely supported.

Key Technical Challenges and Proposed Solutions

Scaling

At the heart of distributed systems lies the challenge of managing resources effectively to meet the ever-changing demands of users. Scaling, the ability to accommodate increasing workloads and maintain performance as the system grows while also not wasting resources when usage decreases, is paramount for ensuring reliability, availability, and performance. Without proper scaling strategies in place, distributed systems can quickly become overwhelmed, leading to downtime, degraded performance, and ultimately, dissatisfied users. Additionally, there can be lots of wasted processing power and associated costs.

Scaling takes on various forms, including vertical scaling (increasing the resources of individual nodes) and horizontal scaling (adding more nodes to the system). Each approach comes with its own set of trade-offs, and striking the right balance between them is essential for optimizing performance and resource utilization.

One of the primary challenges we had to address when designing our system was how to scale our system. In the context of our system, vertical scaling means altering the number of running containers within a node while horizontal scaling is changing the number of nodes. We needed some sort of container orchestration service to accomplish this. Docker Swarm provided the exact functionality that we sought, allowing us to explicitly change the number of containers running a given service as well as cause nodes to join or leave the swarm.

We then had to determine when the system should scale and how to do so in an efficient and safe manner. We scale the number of containers based on the number of tasks on the queue. If this number is above a pre-configured threshold, we spin up a new container. If it’s below a threshold, we shut a container down. Whenever changing the number of containers, we also check the ratio of containers to machines. If it’s too high, we start up a new node, and if it’s too low, we deactivate a node. In other words, when demands change, we first scale vertically, and then horizontally if needed.

We can safely shut down containers without worrying about work being lost, because we have RabbitMQ only acknowledge having received a task when it’s done with that task’s processing. Thus, if a container is shut down while running a task, the “ack” will never be received by RabbitMQ, so a new container will receive that task and process

it.

The problem with this approach is that the computation performed by the original container before it is shut down goes to waste. In the next iteration of this system, we would handle this by having a grace period between when a container is signaled to shut down and when it actually shuts down. During this time, the container wouldn't accept more work and would have time to finish its current work before shutting off.

Testing Multiple Nodes

Another challenge we had to address was testing the system on multiple nodes. For multiple nodes to be in a single Docker Swarm, they all need to be on both the same network and subnetwork. This is not necessarily the case when multiple devices connect to a relatively large network such as YUWireless.

We initially approached this by having our laptops all connect to the same WiFi hotspot, but this was slow and tedious. The hotspot would often have an unreliable connection and would even shut down without warning. Then, Professor Sacknovitz brought in a router, which made it much easier for us to connect our laptops to the same subnetwork.

Failure Handling

Another key feature of distributed systems is the ability to seamlessly recover from failure. Given that our system has multiple components working in tandem, there are many things that can go wrong. We had to determine how to address various types of failure. There are three types of failures that our system handles:

- In the event that a container itself fails, Docker Swarm immediately detects this and spins up a new one. Tasks that were running on the failed container will be restarted on a new container, since, as mentioned above, RabbitMQ doesn't acknowledge having received a task until that task is complete.
- In the event that a container is still active but fails when trying to push or pull a file, this is a transient error and can potentially be recoverable. The container will retry to push or pull a set number of times with exponential backoff. If it still fails, it will escalate the issue by logging to the database that the task with the given ID has failed.
- If a container fails an attempt to transcode a video, this is a non-recoverable error. The container escalates the issue by logging to the database that the task with the given ID has failed.

Redundancy

Another key feature of distributed systems is redundancy. We currently have our RabbitMQ middleman deployed on a separate node, but we would ideally want to run it as a cluster to provide fault tolerance and redundancy. However, this would come along with some significant technical challenges we would need to address. Suppose a task publishing client connected to a particular instance of our RabbitMQ service and that instance went down before the task was completed. We use the default RabbitMQ Docker image, which “auto-acks,” which means that messages are acknowledged immediately upon their delivery. We would need to configure RabbitMQ to not “ack” until it receives an “ack” from the video transcoding service, because otherwise, the work would get lost in such a case. Even if we were to do that, similar to what was discussed in the Scaling section, computation resources would potentially be wasted. We would also implement some sort of algorithm such that if we’re waiting too long to get any acks (the threshold would depend on how many acks we’re expecting), we stop sending requests to that RabbitMQ instance. We would also need a way of keeping track of the number of instances of RabbitMQ currently running so we can dynamically scale them to accommodate varying numbers of incoming requests to ensure optimal availability. Finally, we would need a way for the RabbitMQ instances to coordinate which ports each of them is using.

Conclusion

In conclusion, the video transcoding system represents a sophisticated solution to the challenges posed by the burgeoning demand for video content processing. By leveraging distributed architecture, advanced technologies, and innovative solutions, the system ensures seamless operation, scalability, and high-performance transcoding capabilities. With its ability to handle large volumes of video transcoding tasks efficiently, the system stands poised to meet the evolving needs of today's digital landscape.