Morph: A Fast and Scalable Cloud Transcoding System

Guanyu Gao
School of Computer Science and Engineering
Nanyang Technological University
ggao001@ntu.edu.sg

Yonggang Wen
School of Computer Science and Engineering
Nanyang Technological University
ygwen@ntu.edu.sg

ABSTRACT

Morph is an open source cloud transcoding system. It can leverage the scalability of the cloud infrastructure to encode and transcode video contents in fast speed, and dynamically provision the resources in cloud to accommodate the workload. The system is composed of a master node that performs the video file segmentation, concentration, and task scheduling operations; and multiple worker nodes that perform the transcoding for video blocks. Morph can transcode the video blocks of a video file on multiple workers in parallel to achieve fast speed, and automatically manage the data transfers and communications between the master node and the worker nodes. The worker nodes can join into or leave the transcoding cluster at any time for dynamic resource provisioning. The system is very modular, and all of the algorithms can be easily modified or replaced. We release the source code of Morph under MIT License, hoping that it can be shared among various research communities.

Keywords

Video transcoding; cloud computing; distributed system; task scheduling; resource provisioning;

1. INTRODUCTION

Video transcoding is a popular solution for content adaptation [8]. It is commonly used in adaptive bitrate (ABR) streaming to encode video contents into multiple representations in different bitrates and resolutions. Particularly, with the rapid growth of mobile devices and global mobile data traffic, transcoding has become a must for content adaptation to ensure that the source content can be viewed on any device and in any network conditions. However, transcoding is a complicated signal conversion procedure [8], which consumes massive computing resource and incurs excessive processing delays. Moreover, the traditional in-house transcoding methods, which maintain a fixed number of servers, typically need to over-provision the computing resource by at least 30% to meet the peak workload [4]. This creates a resource wastage when the workload is low.

A new trend for transcoding is to adopt the cloud infrastructure to provide transcoding as a cloud service. Specifically, the cloud transcoding system can use many VM instances or Containers to transcode the video contents in parallel to achieve fast speed. Meanwhile, by leveraging the elasticity of the cloud infrastructure, the system can dynamically provision the resource to adapt to the workload, avoiding excessive processing delays or resource wastage.

Seeking to address the transcoding demand for the fast growing volume of video contents, many online cloud transcoding services have emerged in recent years. Bitcodin [3, 7] is capable of encoding up to 100x faster than real time. Amazon Elastic Transcoder is designed to be scalable and cost-effective for developers to transcode the video files [1]. Zencoder can encode high-definition (HD) quality video faster than real time. The content delivery network service provider Akamai also launched cloud-based transcoding service Sola Vision for adaptive media delivery [2]. Encoding.com can automatically scale the computing resource to achieve a level of computation efficiency. For managing the multiple versions of the video content in ABR cost-efficiently, we implemented the video content management system in our previous work [6] for reducing the operational cost.

Many commercial cloud transcoding services are available online, however, we observe there is still a lack of mature open source software projects for cloud transcoding. This paper presents the design and implementation of our cloud video transcoding system Morph, which is a well documented open source project. We believe it can be used in the research area of transcoding, distributed system, task scheduling, resource provisioning, etc. The system is reliable even under heavy workload, and we provide multiple types of service interfaces so that it can be easily deployed in a production environment. We release the source code of Morph on Github under MIT License. We are still developing new features for Morph, and we welcome suggestion and contribution from communities. The source code and documentation are available at the following links:

Source Code: https://github.com/cap-ntu/Morph
Project Page: http://morph.aidynamic.com/

2. FEATURES

Morph can transcode the video files in fast speed; schedule the tasks according to their Qos profiles; and dynamically

MM '16, October 15-19, 2016, Amsterdam, Netherlands
© 2016 ACM. ISBN 978-1-4503-3603-1/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/2964284.2973792

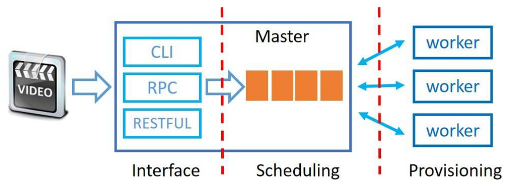

Figure 1: System architecture.

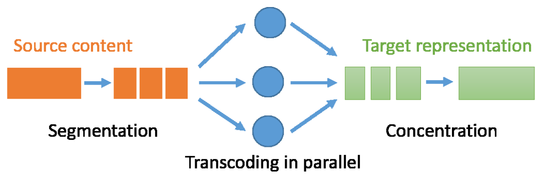

Figure 2: Workflow for a transcoding task.

provision the computing resource to accommodate the workload. Specifically, it has the following main features:

Block level parallelism: Each video file is segmented into many equal-size video blocks, and the video blocks can be transcoded on multiple workers in parallel. Compared with the file level parallelism, this can greatly improve the transcoding speed for the large video files if the number of available transcoding workers in the system is large.

Scheduling to meet QoS requirements: The video contents may have different QoS requirements for transcoding. We implement many scheduling algorithms in our system for meeting the QoS profiles of the transcoding tasks. It is easy to modify or extend the scheduling algorithms in our system.

Elasticity for resource provisioning: The master node of Morph can work with an arbitrary number of transcoding workers. The workers can dynamically joint in or leave the cluster to accommodate the time-varying workload. The system can activate more workers when the transcoding workload is high, and turn off some workers when the workload decreases to achieve dynamic resource provisioning.

3. DESIGN

We first present the architecture of Morph, and then we introduce the workflow for fulfilling transcoding requests.

3.1 Architecture

We illustrate the system architecture of Morph in Fig. 1. The system is composed of the following layers:

Interface layer: It interacts with the users for processing the transcoding requests and preprocessing the video contents. The system provides three types of service interfaces, namely, command line interface (CLI), remote procedure call (RPC), and Restful API. The users can submit the transcoding tasks and query the transcoding progress via the service interfaces. For each of the user submitted transcoding tasks, it will estimate the required computing time for the task and segment the video files into video blocks for transcoding in parallel on the workers.

Scheduling layer: The user submitted transcoding tasks will be put into the scheduling queue. The task scheduler sequences the pending tasks in the queue according to the scheduling policy and the QoS profiles of the tasks. Whenever the master node receives a transcoding request from the worker, the task scheduler will select a video block from the pending tasks for dispatching by applying the scheduling policy. The transcoded video blocks on the worker will be sent back to the master for video concentration.

Provisioning layer: It manages the transcoding workers for dynamic resource provisioning. Our system can adopt the virtual machines (e.g., KVM) or containers (e.g., Docker) for resource virtualization. Each VM instance or container runs a worker. The worker will request a video block from the master node whenever it is idle. The worker will transcode the video block into the target representations, which will be sent back to the master node.

3.2 Workflow

We illustrate the workflow for transcoding a video content in Fig. 2. The user submit a transcoding task by uploading a video file and specifying the transcoding parameters. The video content will be segmented into independent video blocks according to the group of pictures (GOP) structure. The video block information of the task will be then put into the scheduling queue. When an idle worker requests a video block from the master node, the scheduler determines which video block in the scheduling queue will be selected for dispatching by applying the scheduling policy. The worker will transcode the video block into the target representation, and then send back the target representation to the master node. Because the system may have many available workers, the video blocks in the scheduling queue can be requested and transcoded by the workers in parallel. The master node continuously checks the transcoding status of the video blocks of each task. If all the video blocks of a task have been successfully completed, the master node will concentrate the video blocks into one video file. Then, the transcoding task is finished and the target representations of the video content are ready to be downloaded by the users.

4. IMPLEMENTATION

In this section, we present the system implementation. The main components of the system are illustrated in Fig. 3. The components of the interface, segmenter, scheduler, data transfer, and concentrater, are implemented as independent threads in the master node. Each worker is an independent progress running on a VM instance or container. The resource controller is an independent progress for managing the status of the workers. We adopt the FFmpeg for the transcoding operations. We discuss the details of the implementation for each part in the following subsections.

4.1 Interface

We provide three types of interfaces for the users to access the transcoding service, namely, CLI, RPC, and Restful API. When the user submits a task, the system will return a key. The user can use the key for querying the transcoding progress of the task. We adopt the SimpleXMLRPCServer for implementing the XML-RPC servers in Python.

4.2 Segmenter

The video segmenter determines the duration of each video block when segmenting the video files. The video block duration impacts the transcoding performance of the system. Specifically, if the duration of each video block is short, a video file is to be segmented into more blocks, and it would incur more communication overhead among the master and

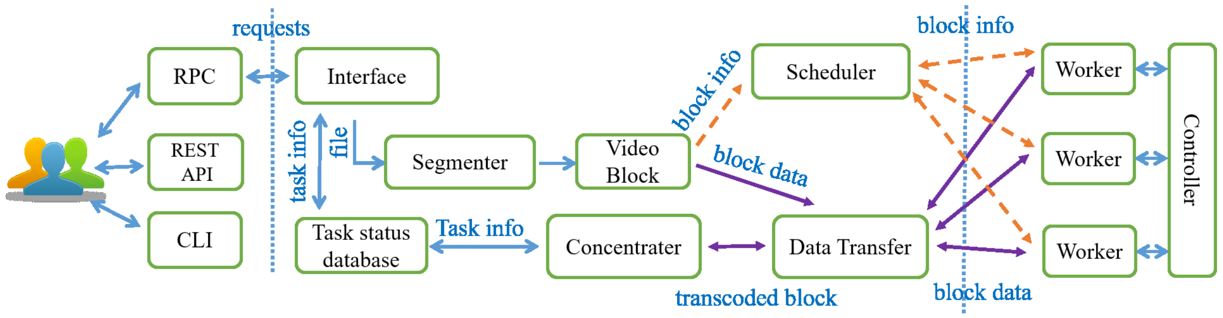

Figure 3: The system implementation of Morph.

the workers. On the other hand, if the duration of each video block is very long, a video file is to be segmented into fewer blocks, it cannot take the advantage of the large number of workers for parallel transcoding the video file. Our system provides the two following methods for determining the duration of the video block for a task:

1) *Predefined*: The system can read a predefined value from the configuration file when starting. With this method, the video blocks of all tasks have the same duration.

2) *Dynamically determined*: The duration of a video block for a specified task can be determined on the fly when segmenting the video file by applying a segmentation algorithm.

4.3 Scheduler

The system maintains a queue for the pending transcoding tasks, and the scheduler sequences the tasks in the queue periodically according to the scheduling algorithms. The scheduling algorithms are implemented as an independent module in the source file of 'algorithms/scheduling.py'. Each scheduling algorithm is implemented as a function. It is easy to modify or extend the scheduling algorithms without affecting the other modules of the system. We have implemented the following scheduling algorithms in the system: 1) *FIFO*: First In First Out. 2) *LIFO*: Last In First Out. 3) *EDF*: Earliest Deadline First. 4) *HPF*: Highest Priority First. 5) *HVS*: Highest Value First. The scheduling algorithms affect the QoS of the transcoding service. The system operator can set the scheduling algorithm in the configuration file based on the QoS considerations of the service. For instance, one can set sch_alg = 'hpf' in configuration for scheduling the tasks with the highest priority first.

4.4 Transfer

The data transfer interacts with the workers to send the video blocks and receive the transcoded video blocks from the workers. The video block data will be packed with the transcoding parameters and MD5 checksum of the data, and then sent to the worker for transcoding. The format for packing and unpacking the video block data is defined in the file 'common.py'. The data transfer uses multithreading mechanism for sending and receiving the data, and it can communicate with multiple workers at the same time.

4.5 Worker

The worker requests the video block for transcoding from the master whenever it is idle. After receiving the video block data from the master, it will check the validity of the data and transcode it into the target representations. If the transcoding for the video block is finished successfully, it will pack the video data with the transcoding information and send it to the master. Otherwise, it will notify the master of the failure of the task.

4.6 Concentrater

The concentrater is implemented as a thread for checking and updating the status of each transcoding task. If all the video blocks of a task have been transcoded successfully, it will concentrate the video blocks of the task into one video file and update the status of the task as success. If any video block of a task is transcoded unsuccessful, it will update the status of the task as fail.

4.7 Controller

The controller determines which worker should be activated. It reads the pending tasks information from the system database for calculating the workload, and makes the control decisions by applying the provisioning policy. In our implementation, the control of the status of the workers is independent of starting or stopping the VMs or Containers. We create a field for each worker in the database to represent the status of the workers, where 1 represents active and 0 represents sleep. The controller updates the values according to the control decisions. The worker determines its own working status by accessing the value of this field. The API for starting or stopping a VM or container depends on the specific resource virtualization software. Each software provides its own version of interfaces (e.g., the remote API for Docker). We did not cover the control for the VM and Container in our implementation due to the dependence on the API interface of the specific virtualization software.

5. DEPLOYMENT

In this section, we present the system deployment in the real environment. We also discuss the potential system bottleneck in large scale deployments and the possible solutions.

5.1 Software Stack

We illustrate the software stack of Morph in Fig. 4. The master and worker depend on ffmpeg for video transcoding. Therefore, each VM or Container that runs the master or worker needs to install ffmpeg. We adopt MySQL database for storing the task information. The master, workers, and controller should all can connect to the MySQL server for accessing the information. If one needs to use the Restful API to access, the Apache HTTP webserver should be installed and configured for processing the HTTP requests.

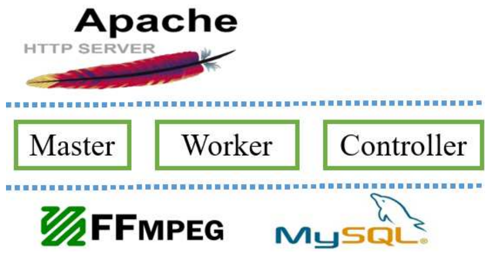

Figure 4: The software stack of Morph.

5.2 Potential Bottleneck

We design and implement the system in a simple and reliable way, however, it still have some potential performance bottleneck. The main performance bottleneck is from the master node, which is a single point of failure. One concern is that the master node will have more overhead for transferring and processing the video data with more active workers in a cluster. However, the video data transferring and processing (segmentation and concentration) time in the master node is relatively small compared with the transcoding time in the worker node. In our experiment, the workload of the master node is very small when the system has 30 active workers that work at the full capacity. If the number of provisioned workers in the system is very large, the performance bottleneck of the master node will become significant. Another concern is that the single point failure of the master node will make the transcoding service unavailable, while the system reliability is critical for the online service.

One effective solution for the above discussed problems is to introduce a load balancer into the system, as illustrated in Fig. 5. A load balancer has daemon processes and can recover itself from failure very fast if it collapses on its own. We can divide the system into smaller groups, and each group consists of a master node and many workers. Normally, the balancer balances the workload among the groups. If the balancer detects the failure of one master node, it will direct the transcoding requests to the other active groups until the broken-down master node recovers.

6. PERFORMANCE

Testbed: The duration of the test video file is 138 minutes. The resolution is 1920x1080, and the bitrate is 2399 kb/s. The video data is encoded in H.264, and the audio data is encoded in AAC. The CPU frequency is 2.10GHz. The master node is allocated with 8 CPU cores, and the memory size is 8GB. The worker node is allocated with 4 CPU cores, and the memory size is 2GB. We use the Docker to build the cloud platform. The video block duration is 2 minutes. The target resolution of the transcoding is 480x360. *Transcoding time:* We measure the transcoding time using Morph with different number of active workers, and compare it with the standalone FFmpeg method. The transcoding time and the speedup ratio compared with the standalone FFmpeg method are illustrate in Table 1. The transcoding time for using a standalone FFmpeg on a single server is 1775 seconds. With one active worker, the transcoding time for Morph is 1843 seconds, larger than the standalone ffmpeg method. The overhead comes from the video segmentation, transmission, and concentration operations. Specifically, the video segmentation time is 46 seconds, and the video block concentration time is 13 seconds. With more active workers, the transcoding time for Morph decreases, because it can transcode the video blocks in parallel with more workers.

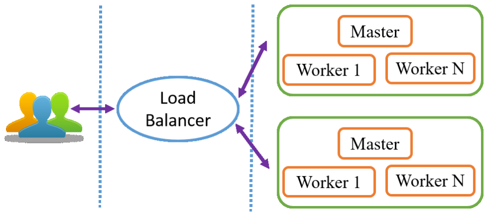

Figure 5: Load balancing for large scale deployment.

Table 1: Transcoding time comparison

Worker Num	FFmpeg	1	5	10	20	30
Time (s)	1775	1843	605	369	213	181
Speed-up (x)	1	0.96	2.9	4.8	8.3	9.8

Task scheduling and resource provisioning: We have also designed some task scheduling and resource provisioning algorithms based on Morph, and evaluated the performances in our previously work [5].

7. CONCLUSION

This paper presents the design and implementation of our open source cloud transcoding system Morph. We aim to design a cloud transcoding system that can achieve fast transcoding speed and elastic system scaling. The system is useful for the research and system development in the field of video transcoding, distributed system, task scheduling, and resource provisioning. The project is open sourced and well documented. In the future, we will continue to maintain and develop new features and functions to extend the system. We hope it can be easy for the researcher to verify the algorithms on it, and we also expect that it can be used as an efficient software for transcoding in a cloud environment.

8. REFERENCES

[1] http://aws.amazon.com/elastictranscoder/.
[2] http://www.beet.tv/2012/09/akamaicloud.html.
[3] Bitcodin. https://www.bitmovin.com/encoding/.
[4] J. Careless. Cloud video encoding: When to go online and when to stay in-house. *Streaming Media*, 2012.
[5] G. Gao, Y. Wen, and C. Westphal. Resource provisioning and profit maximization for transcoding in information centric networking. Infocom Workshop on Muisc, 2016, http://arxiv.org/abs/1605.05758.
[6] G. Gao, Y. Wen, W. Zhang, and H. Hu. Cost-efficient and qos-aware content management in media cloud: Implementation and evaluation. In *IEEE International Conference on Communications (ICC)*. IEEE, 2015.
[7] C. Timmerer, D. Weinberger, M. Smole, R. Grandl, C. Müller, and S. Lederer. Cloud-based transcoding and adaptive video streaming-as-a-service. *E-letter*.
[8] A. Vetro, C. Christopoulos, and H. Sun. Video transcoding architectures and techniques: an overview. *Signal Processing Magazine, IEEE*, 20(2):18–29, 2003.