Dynamic Scheduling on Video Transcoding for MPEG DASH in the Cloud Environment

He Ma\textsuperscript{1,2}, Beomjoo Seo\textsuperscript{1}, Roger Zimmermann\textsuperscript{1}

\textsuperscript{1}School of Computing, National University of Singapore, Singapore 117417
\textsuperscript{2}National University of Singapore (Suzhou) Research Institute, 377 Lin Quan Street, Suzhou Industrial Park, Jiang Su, People’s Republic of China, 215123
\{mahe,seobj,rogerz\}@comp.nus.edu.sg

ABSTRACT

The Dynamic Adaptive Streaming over HTTP (referred as MPEG DASH) standard is designed to provide high quality of media content over the Internet delivered from conventional HTTP web servers. The visual content, divided into a sequence of segments, is made available at a number of different bitrates so that an MPEG DASH client can automatically select the next segment to download and play back based on current network conditions. The task of transcoding media content to different qualities and bitrates is computationally expensive, especially in the context of large-scale video hosting systems. Therefore, it is preferably executed in a powerful cloud environment, rather than on the source computer (which may be a mobile device with limited memory, CPU speed and battery life). In order to support the live distribution of media events and to provide a satisfactory user experience, the overall processing delay of videos should be kept to a minimum. In this paper, we propose a novel dynamic scheduling methodology on video transcoding for MPEG DASH in a cloud environment, which can be adapted to different applications. The designed scheduler monitors the workload on each processor in the cloud environment and selects the fastest processors to run high-priority jobs. It also adjusts the video transcoding mode (VTM) according to the system load. Experimental results show that the proposed scheduler performs well in terms of the video completion time, system load balance, and video playback smoothness.

Categories and Subject Descriptors

G.1.0 [General]: Parallel algorithms; G.3 [Probability and Statistics]: Statistical computing; I.2.8 [Problem Solving, Control Methods, and Search]: Scheduling

General Terms

Algorithms, Measurement, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

\textsc{MMSys ’14}, March 19 - 21, 2014, Singapore, Singapore
Copyright 2014 ACM 978-1-4503-2705-3/14/03 $15.00.

1. INTRODUCTION

Recently, Over-The-Top (OTT) streaming, \textit{i.e.}, delivering video and audio content through the public Internet infrastructure rather than proprietary infrastructures such as cable networks, has become an active topic in the broadcasting and content delivery communities. OTT in particular refers to content that arrives from a third party, such as Netflix and Hulu, and is delivered to an end-user device, leaving the Internet provider responsible only for transporting packets. The final link to end-users is usually handled with HTTP streaming, or other proprietary technologies. Consumers can access OTT content through various Internet-connected devices such as desktops, laptops, tablets, smartphones, TVs and gaming consoles (\textit{e.g.}, Xbox 360, PlayStation, Wii). The variety of devices requires the video hosting services to provide different bitrates of the original videos.

Moreover, the wide-spread availability of smartphones (and increasingly tablets) and the rapid improvement of wireless networks (3G/4G, or WiFi) have enabled the possibility of daily video streaming through mobile devices. A study from Cisco \cite{8} has indicated that the overall mobile data traffic reached 885 petabytes per month at the end of 2012, 51 percent of which is mobile video. Forecasts predict that mobile video will grow at a Compound Annual Growth Rate (CAGR) of 75 percent between 2012 and 2017 and reach one exabyte per month by 2017. An important consideration is that the bandwidth for mobile devices varies depending on location and time. The changes in bandwidth influence the quality of video streaming. For example, assume that all available bandwidth of a mobile client is used to watch a video. When the bandwidth increases, it has the capacity to watch a higher quality video. Conversely, when the bandwidth decreases, the playback would be interrupted due to insufficient bandwidth for the current bitrate. Therefore, to guarantee smooth playback and enable streaming of the highest possible quality, it is essential that media streaming can adapt to the current network bandwidth and conditions. Dynamic Adaptive Streaming over HTTP (DASH) is designed to provide high quality streaming of media content over the Internet delivered from conventional HTTP web servers. Figure 1 illustrates the architecture of DASH. Media content is encapsulated into a parallel sequence of segments with a number of different bitrates so that an MPEG DASH client can automatically and seamlessly select the next segment to download and play back based on current

Keywords

DASH, Video transcoding, Scheduling, Cloud computing
network conditions. YouTube [20] statistics show that over four billion hours of videos are watched each month and 72 hours of video are uploaded every minute. The popularity of video streaming has highlighted video transcoding as a challenging problem. In recent years, cloud computing has become an effective paradigm for many applications. It is a technology aimed at sharing resources and providing various computing and storage services flexibly over the Internet. For multimedia applications and services, there are strong demands for cloud hosting because of the significant amount of computation required for serving millions of Internet and mobile users simultaneously [23]. The complex nature of video transcoding (e.g., CPU-intensity) and the high demand requirements of streaming have enabled cloud computing to be uniquely suitable for video transcoding, especially in the context of large-scale video hosting systems. Therefore, transcoding is preferably executed in a powerful cloud environment, rather than on the source computer (which may be a mobile device with limited memory, CPU speed and battery life).

In order to support live streaming of media events and to provide a satisfactory user experience, the overall video transcoding completion time should be minimized. In the rest of the paper we refer to this simply as the video completion time or the completion time. Minimizing the video completion time is desirable for several reasons. Obviously it is crucial not to exceed the waiting tolerance of clients. Because of the high computational complexity, scheduling video transcoding jobs in a cloud environment to minimize the processing latency and satisfy users’ expectations is a challenging problem. Furthermore, it is important to balance the workloads among all the processing nodes in the cloud. In this paper we propose a dynamic scheduling algorithm for DASH video transcoding designed for cloud environments. The scheduler makes use of the estimation of the video transcoding time (VTT). Jobs are distributed to free processors when they are not urgent, but to the fastest processors if video watching requests are pending. Experimental results show that the scheduler performs very well in executing video transcoding jobs and balancing the workload among all the processors. The main contributions of our method are summarized as follows:

- We introduce a video transcoding time (VTT) estimation, with respect to video segment duration and targeted bitrates, that is modeled based on measured statistics and probabilistic theory.
- We incorporate a job distribution mechanism among processors that achieves load balancing among the processing resources.
- The scheduler includes workload and status monitoring for each processor, and dynamically selects the fastest processors to run high-priority jobs when videos have pending viewing requests.
- The scheduler dynamically optimizes the video transcoding mode (VTM) when the number of processors is insufficient to support all video watching requests, so that the processors prioritize transcoding of the requested bitrates and leave other bitrates to be processed later.

The rest of this manuscript is organized as follows. Section 2 introduces the fundamentals of HTTP and DASH streaming, as well as cloud video transcoding services and scheduling strategies on parallel video transcoding. Section 3 models the VTT estimation methodology considering the video duration. The scheduling algorithm and evaluation metrics are detailed in Section 4. Section 5 presents the experimental results and analysis. Finally, Section 6 concludes the paper and investigates the future work.

2. BACKGROUND AND RELATED WORK

2.1 HTTP Streaming Fundamentals

With the development of content delivery networks (CDN), Hypertext Transfer Protocol (HTTP) streaming has emerged as the de-facto streaming standard, replacing the conventional streaming with the Real-Time Transport Protocol (RTP) and Real-Time Streaming Protocol (RTSP). Under typical HTTP streaming, once an client sends a request and establishes a connection between the server and itself, a progressive media download is activated until the streaming is terminated [19]. Disadvantages of a progressive download include: (a) unstable conditions of the network, especially a wireless connection for mobile clients, may cause bandwidth waste due to reconnection or rebuffering events, (b) it does not support live streaming (e.g., concert or football match), and (c) it does not support adaptive bitrate streaming. In recent years, streaming platforms such as Microsoft’s Smooth Streaming (MSS) [14], Apple’s HTTP Live Streaming (HLS) [16], and Adobe’s HTTP Dynamic Streaming (HDS) [4], all use HTTP streaming as their underlying delivery method and support adaptive streaming as well. The commonalities [18] of these techniques are: (1) splitting an original encoded video file into small pieces of self-contained media segments, (2) separating the media description into a single playlist file, and (3) delivering segments over HTTP. Among each other, these techniques differ in the following way: (i) MSS is a compact and efficient method for...
the real-time delivery of MP4 files from Microsoft’s IIS web server, using a fragmented, MP4-inspired ISO Base Media File Format (ISO/MBMF). (ii) HLS uses an MPEG-2 Transport Stream (TS) as its delivery container format and utilizes a higher segment duration than MSS, and (iii) DASH is based on Adobe’s MP4 fragment format (F4F) and its corresponding XML-based proprietary manifest file (F4M).

Published in April 2012, DASH [2] addresses the above weaknesses of simple, progressive HTTP streaming. A video file is broken into a sequence of small playable HTTP-based segments and these segments are uploaded to the standard HTTP server sequentially. The visual content is then encoded at a variety of different bitrates and the HTTP-client can automatically select the next segment from the alternatives to download and play back based on current network conditions. The client selects the segment with the highest possible bitrate that can be downloaded in time for smooth and seamless playback, without causing rebuffering events. DASH standardizes the two most important components: the Media Presentation Description (MPD) and the segment formats.

2.2 Video Transcoding Services in the Cloud

There exist several cloud services that offer video transcoding. Amazon released their Elastic Transcoder [7] in January of 2013. It executes transcoding jobs using Amazon’s Elastic Compute Cloud (Amazon EC2 [5]) and stores the video content in Amazon’s Simple Storage Service (Amazon S3 [6]). Amazon’s Elastic Transcoder manages all aspects of the transcoding process transparently and automatically. It also enables customers to process multiple files in parallel and organize their transcoding workflow using a feature called transcoding pipelines. EncoderCloud [9] also provides video transcoding services as well. In addition to providing typical video transcoding services in the cloud, it also supports live video transcoding. EncoderCloud [9] also provides the same web-based “pay-as-you-go” service and helps to build applications on top of other service providers (e.g., Amazon EC2 and RackSpaceCloud [17]), but offers a different pricing policy – charging by the volume of the total amount of source video transferred in and encoded video transferred out. These services provide the capability of video transcoding in the cloud, but the transcoding scheduling mechanism is transparent to end-users.

2.3 Scheduling Strategies

Cloud computing provides tremendous computing resources for applications but there is no universal “best” scheduling algorithm. Instead, they are usually optimized for certain applications. One study [12] introduces a parameter-tuning framework by combining the bitrate and encoding speed as encoding cost and provides a cost optimization method for video cloud transcoding. Li et al. [11] proposed a parallel video encoding strategy considering a load balance factor. By considering the granularity of the load partitions and the associated overheads caused, they utilized the divisible load theory paradigm to distribute the video frames among processors. Zaharia et al. [21] presented a fair job scheduling for Hadoop to minimize the job response time. Li et al. [13] developed a cloud transcoder that takes the utilization of CPU into account to help to predict video transcoding tasks. Kllapi et al. [10] presented an optimization of scheduling dataflows on these three aspects: minimizing completion time, minimizing the monetary cost given a deadline, and finding the trade-off between completion time and monetary cost. However, these approaches investigate neither the specific properties of DASH (e.g., correlation between segments, alternative bitrates of uploaded segments) nor the interaction between the service hosts and the end-users. To the best of our knowledge, although existing scheduling strategies can be applied, there exists no specific work on scheduling video DASH transcoding for cloud environments, especially while untranscoded segments already have pending video watching requests.

3. TRANSCODING TIME ESTIMATION

We first introduce our approach of estimating the video transcoding time (VTT) from the original stream to other qualities, i.e., different bitrates. For DASH, the server needs to prepare multiple bitrates and multiple formats of the originally uploaded videos. Since the time for the video transformatting jobs (i.e., transformatting .mp4 files to .ts files) is only a few milliseconds, we focus on scheduling the video transcoding jobs in this study. Furthermore, preparing the MPD, including the playable file name and its HTTP link, not affecting the performance of the scheduler, is also ignored in this paper. Without loss of generality, we consider two reduced quality streams, namely encoding the original video segments at a medium bitrate (708 kbps with resolution of 480×360) and low bitrate (256 kbps with resolution 360×240), respectively. The scheduling algorithm is also applicable to a larger number of targeted bitrates. The scheduler will hence distribute video transcoding jobs (VTJ) based on the estimation of VTT. The VTT includes the sum of the following two parts: the time for the file transfer between the storage repository and the processing nodes, and the time for the actual transcoding procedure. In order to model the estimation of VTT given the video duration, we measure the actual VTT (denoted as VTT_{act}) for statistics on a set of video segments in the shared cloud environment, which is also used to run experiments with the synthetic dataset.

3.1 Configuration of the Cloud Environment and Description of Dataset

The shared cloud environment includes one master node and 25 processing nodes. All the nodes are running CentOS 5.5 and can access a shared storage system. The master node is used to run the scheduler with two quad core Intel® Xeon® E5440 2.83 GHz CPUs and 16 GB of memory. The processor nodes used to run VTJs consist of two quad core Intel® Xeon® E5620 2.4 GHz CPUs and 24 GB memory. The dataset includes 11,194 video segments from 339 videos which we collected with Android phones and are coded in MPEG-4 (2 Mbps with resolution of 720×480). These videos are segmented at the mobile clients and uploaded to remote servers [18]. For smooth and seamless rendering of the segments, each contains an integral number of Group-Of-Pictures (GOP). In the configuration of the mobile application, the segment durations were chosen as 3, 4 or 5 seconds, and the duration of the last segment of each video varies accordingly from about 0.2 second to 6.5 seconds. On the remote servers, all the segments are transcoded and transformed with the open source platform FFmpeg [1].
Figure 2: VTT statistics and the fitting curves to different bitrates with respect to video duration.

### 3.2 VTT Estimation Methodology

In order to avoid the effect of caching on the VTTJs, we deployed the VTJ for low and medium bitrates on different processors, which could be fully utilized without being occupied by other jobs. To minimize the runtime bias (e.g., the variability of the connection speed between processors and the storage repository, and individual processing time) on VTJs, we ran these jobs 10 times across different processors and calculated the mean values of VTT. Figure 2 presents the measured VTT to low and medium bitrates, respectively, as well as the corresponding fitting curves (denoted as VTT_cal), with respect to the video segment duration. Compared among all the fitting curves, the power fitting curve achieves the minimum sum of square. Therefore, the fitting curves can be calculated as formulated in Eqn. 1.

\[
VTT_{cal}(\text{dur}(V_{TJ})) = a \cdot (\text{dur}(V_{TJ}))^b
\]

where \(\text{dur}(V_{TJ})\) indicates the video duration, and \(a\) and \(b\) are the fitting coefficients. In our current configuration of the cloud environment, the values of \(a\) and \(b\) are shown in Eqn. 2:

\[
[a \ b] = \begin{cases} 
0.796 & 0.621 \quad \text{(low)} \\
1.152 & 0.700 \quad \text{(medium)} 
\end{cases}
\]

As shown in Figure 2, the actual VTTs are always different from the value calculated by the fitting curve for most of the time. In order to match the VTT estimation with the measurements, we calculate the bias of the measured VTT with respect to the calculated VTT. The normalized error for each VTT is calculated as shown in Eqn. 3:

\[
T_{err} = \frac{VTT_{mea} - VTT_{cal}}{VTT_{cal}}
\]

The distributions of the value of \(T_{err}\) follow a Gamma distribution (shown in Eqn. 4):

\[
\text{Prob}(\tilde{T}_{err}) \sim \frac{1}{\Gamma(k)} \cdot \tilde{T}_{err}^{k-1} \cdot e^{-\frac{T_{err}}{\theta}}
\]

where \(\tilde{T}_{err} = \text{round}(T_{err} \cdot 100)\). Overall, we calculate \(\text{Prob}(\tilde{T}_{err})\) over the whole dataset for each individual VTT and compute the value of \(k\) and \(\theta\) (Table 1):

<table>
<thead>
<tr>
<th></th>
<th>(k)</th>
<th>(\theta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>low bitrate</td>
<td>8</td>
<td>2.4</td>
</tr>
<tr>
<td>medium bitrate</td>
<td>5.8</td>
<td>3.4</td>
</tr>
</tbody>
</table>

Table 1: The coefficients of Gamma distribution used in the shared cloud environment.

Based on this, the estimated VTT can be calculated as:

\[
VTT_{est} = VTT_{cal} + VTT_{err} = VTT_{cal}(1 + T_{err})
\]

Figures 3(a) and 3(b) illustrate the distribution of \(T_{err}\) to low and medium bitrates, respectively.

![Figure 3: The distribution of \(T_{err}\) in the shard cloud configuration.](image-url)

Ideally, when a processor is not running other jobs, the VTT of a given segment follows the calculation in Eqn. 5. However, due to various reasons (e.g., shared usage of CPU and/or memory, as well as the congestion of the network), the actual VTT might be significantly different from the estimated value. This would affect the performance of the scheduler, especially when processing videos with viewing requests. Practically, the scheduler compares the actual VTT with the estimated value and treats the difference as feedback from the processor that ran the VTJ. The scheduler will thus distribute VTJs, also considering the feedback from each of the processors. The detailed scheduling algorithm is presented in the next section.
4. SCHEDULING ALGORITHM

This section introduces the dynamic scheduling algorithm for video transcoding with DASH in a cloud environment. Figure 4 shows the overall architecture of the framework. When a video is uploaded to the front-end web server clusters (referred to as a job), the server load balancer accepts this job and assigns it to a HTTP server for processing with the consideration of balancing the workload. The HTTP server then forwards the job of the video transcoding request to the job scheduler and stores the uploaded video in the video storage repository. The job scheduler assigns jobs once any of the processors is available according to its scheduling strategy. When a client sends the video retrieval request (i.e., for viewing) to the web server, the request will be forwarded to the specific server for video retrieving. If the required video segment is available, an HTTP connection is set up between the mobile client and the storage repository for streaming. Otherwise, the request is converted to a high-priority job for the targeted video and the scheduler immediately searches for the fastest processor to answer the request.

4.1 Evaluation Metrics

In the experiment configuration, the HTTP client is also set up in the same LAN with the cloud, the transmission delay between the client and the server is thus assumed to be zero. The performance of the scheduler can be then evaluated by the following evaluation metrics, considering the playback on the client side.

Startup latency ($L_s$): This metric measures the time interval from when the video segment is started to be transcoded until the video is available for playback.

Number of quality switches ($N_{QS}$): The playback will switch to other bitrates if the subsequent segment of the current bitrate is not available, or the network conditions change.

Number of rebuffering events ($N_{RE}$): The video client has to pause the playback of both audio and video during rebuffering if there exists no lower bitrate to switch to.

Mean rebuffering time ($T_{mr}$): During rebuffering, the video is paused until the video can be restarted. The duration of the rebuffering period varies. The longer the rebuffering period is, the worse the video streaming performs.

Mean Opinion Score (MOS): We apply the calculation from Mok et al. [15] to express users’ quality of experience (QoE). The metric is based on a regression analysis to acquire the relationship between QoE and the application quality of service (QoS). The MOS can be calculated as shown in Eqn. 6:

$$MOS = 4.23 - 0.0672L_s - 0.742(N_{QS} + N_{RE}) - 0.106T_{mr}$$

(6)

Load Balance Factor (LBF): This measures the balance among different processors. We use the standard deviation of the overall VTT of all the processors normalized by the average total VTT to estimate the load balance factor.

$$LBF = \left( \frac{1}{N} \sum_{i=0}^{N-1} T_{pro(i)} - \frac{1}{N} \sum_{i=0}^{N-1} T_{pro(i)} \right)^2 / \left( \frac{1}{N} \sum_{i=0}^{N-1} T_{pro(i)} \right)$$

(7)

where $N$ denotes the number of processors in the cloud environment and $T_{pro(i)}$ indicates the overall running time of processor $i$.

4.2 Scheduler

In the backend cloud environment, the job scheduler maintains two queues: one queue keeps all the normally uploaded jobs (referred as $NQueue$), while the other one maintains the jobs with high-priority (referred as $PQueue$). All jobs are initially inserted into $NQueue$, and jobs are migrated to $PQueue$ only if the corresponding videos are requested for watching. Figure 5 illustrates the switching of jobs between $NQueue$ and $PQueue$. Figure 5(a) shows an example of the initial state of $NQueue$ and $PQueue$ at time $t$. Initially, Videos A to D are uploaded to the server and are waiting to be transcoded. Utilizing the methods introduced in Section 3, jobs in $NQueue$ are sorted according to their estimated VTT ascendingly (shown in Algorithm 1). Once new video segments are uploaded to the server, the scheduler calculates the estimated job completion time and inserts the related job into $NQueue$ at the correct location. In our current approach, we publish the video to be available for watching when the beginning $L_{va}$ seconds of a video are transcoded. As shown in the example of Figure 5, once Segment $V_{B_1}$ is under request, all of its subsequential segments from Video B ($V_B$) are assigned to be high-priority and transferred to $PQueue$ (shown in Figure 5(b) and lines 10-14 of Algorithm 2). Note that the jobs in $PQueue$ are sorted by their deadlines ascendingly. When one of the subsequent segments (e.g., $V_{B_{m+1}}$) is required by another client, itself and the subsequent jobs are reinserted into $PQueue$ according to the updated deadlines (shown in Figure 5(c) and lines 15-18 of Algorithm 2).

Figure 5: Illustration of inserting jobs into the $NQueue$ and $PQueue$.

**Definition 1** The deadline of a job.

When a video segment is requested to be watched by end-users, its corresponding and subsequent jobs are selected as a high-priority job. To guarantee smooth playback for video streaming, the VTJ needs to be accomplished before its previous segment is played to the end. The predicted time for its previous segment to be played to the end is defined as the deadline of a job (referred as $Job_{DL}$). As presented in the above paragraph, we publish a video as available for watching after the beginning $L_{va}$ seconds of
the video are transcoded. Therefore, \( Job_{DL} \) equals to the cumulative video duration of all its previous segments plus \( L_{va} \). Considering the example of Figure 5(b), Eqn. 8 shows the calculation of \( Job_{DL} \) given a required job on video segment \( V_{B1} \) and all its subsequent segments

\[
Job_{B_i}^{DL} = L_{va} + \sum_{k=1}^{i-1} (\text{dur}(V_{B_k})) \quad (i > 1)
\]

where \( \text{dur}(V_{B_k}) \) is the duration of \( V_{B_k} \).

Algorithm 1: videoUpload()

Input: An uploaded video segment \( V_{Ij} \)

1. \( J_{ID} = \text{assignID}(V_{Ij}); \) // assign job ID to \( V_{Ij} \)
2. \( t_{est} = \text{timeEst} (\text{duration of } V_{Ij}); \) // transcoding time estimation
3. insert2NQ(\( J_{ID}, t_{est} \)); // insert into NQueue according to time by SJF

Figure 4: The architecture of scheduling on video transcoding for DASH in the cloud environment.
Algorithm 2: videoUnderRequest()

Input: Video segment $V_{ij}$ is under request, 
The request bitrate is $B_{req}$
1 for $j \leq j \leq m$ do
  2 $J_{ID} = \text{getID}(V_{ij})$; // get the job ID
  3 if $J_{ID}$ is accomplished then
  4 | send the link to the HTTP-client;
  5 end
  6 else if $J_{ID}$ is being processed then
  7 | wait until $J_{ID}$ is accomplished;
  8 | send the link to the HTTP-client;
  9 end
10 else if $J_{ID}$ is in $NQueue$ then
11 | $t_{dl} = \text{calDL}(J_{ID})$; // calculate the deadline
12 | insert2PQ($J_{ID}, B_{req}, t_{dl}$); // move $J_{ID}$ to $PQueue$
13 | removeNQ($J_{ID}$); // remove $J_{ID}$ from $NQueue$
14 end
15 else
16 | /* $J_{ID}$ is in $PQueue$ */
17 | $t_{dl} = \text{calDL}(J_{ID})$;
18 | updatePQ($J_{ID}, B_{req}, t_{dl}$); // reinsert $J_{ID}$
19 | into $PQueue$ and remove the old entry
20 end
21 end

5. EXPERIMENTAL EVALUATION

In order to test the performance of the scheduler under critical conditions, all the experiments are conducted under the cloud environment with limited (instead of elastic) computational resources. Two different datasets are applied in the experiments: a real-world dataset and a synthetic dataset.

5.1 Experiment with a Real-World Dataset

We compare the proposed scheduling algorithm with the FIFO policy utilizing a real-world dataset in a private cloud environment where all the computing resources can be fully utilized by the scheduler.

5.1.1 Experimental Configuration

We set up a cloud environment with ten commodity PCs connected with a high speed gigabit network, where one PC was the master node and the others are the processing nodes. Each PC contains an Intel® Quad Core® 2.66 GHZ CPU, 4 GB memory and is running under CentOS Linux 5.6. The testing real-world dataset was collected in collaboration with the Columbia College of Chicago, where journalism students recorded the events that happened in the streets of Chicago during the NATO Summit in May 2012. Overall, the dataset includes nine uploaders submitting a total of 259 videos, which consist of 4,899 video segments. The length of these video segments varies from 0 to 10 seconds and the overall duration of the segments is around 6.8 hours. Most of the segments are five seconds long, while about 2% of the segments have a longer duration (e.g., 10 seconds), and 4%

have a shorter duration.

5.1.2 Performance Evaluation

Following the method introduced in Section 3.2, using the same dataset (introduced in Section 3.1) but using a different cloud configuration, the coefficients of the fitting curves and the Gamma distributions of the private cloud configuration are summarized in Table 2.

<table>
<thead>
<tr>
<th></th>
<th>low bitrate</th>
<th>medium bitrate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>0.006173</td>
<td>0.01263</td>
</tr>
<tr>
<td>$b$</td>
<td>0.5655</td>
<td>0.518</td>
</tr>
<tr>
<td>$k$</td>
<td>7.8</td>
<td>2.5</td>
</tr>
<tr>
<td>$\theta$</td>
<td>5.5</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Table 2: The coefficients of the fitting curves and the Gamma distributions used in private cloud environment.

We study the efficiency of the proposed algorithm compared with standard FIFO policy with a real-world video uploading stream set. In this experiment, there exist only video uploading streams but no video viewing requests. Therefore, the video segments are transcoded to both low bitrate and medium bitrate simultaneously (VTM = 0). Figure 6(a) shows the number of segments received at the server side per
second, which during some periods of time exceeds the maximum capacity of the system. Figures 6(b) and 6(c) present the comparison of the cumulative segment queueing time (CSQT for short) and completion time (CSCT) between the proposed algorithm and the FIFO algorithm, respectively. Overall, the differences between CSQT and CSCT for both of the algorithms are not significant. The proposed algorithm presents a slight improvement on both the CSQT and CSCT. The reason is that we only have a small uploading dataset with a light workload and most of the video segments are of the same length. The transcoding sequences for both algorithms are almost the same. When the workload exceeds the system capacity, the scheduler starts to distribute the predicted faster jobs to other processors and hence the CSQT and CSCT decrease. Moreover, since a processing node can finish most VTJs with around 1.5 to 2 seconds, the difference on the CSQT is always small.

At the beginning of the uploading procedure (i.e., for the first 200 segments), the workload never reaches the maximum capacity of the system. The cumulative segment queueing time is almost zero and there exists no difference between the two algorithms. When the workload starts to exceed the system capacity (e.g., at the 240th segment) and the queue grows, the values of CSQT and CSCT increase and a small gap begins to exist between the two algorithms. While the system works under overload (i.e., uploading segments with IDs around 2,500 to 4,000), the CSQT has a larger increasing rate and both the gaps in CSQT and CSCT increase. From the above analysis, it can be inferred that the proposed scheduler can improve the video transcoding performance, especially when the system is overloaded with a heavy workload.

5.2 Experiment with a Synthetic Dataset

We report on the performance of the scheduler with respect to segment completion time, MOS, load balance and playback smoothness when the system keeps working under heavy workload with a synthetic dataset.

5.2.1 Experimental Configuration

To test the functionality whether the scheduler can dynamically identify the fastest processor to run urgent jobs, all the experiments were conducted in another shared cloud environment. The used processors could also be accessed by other users and used to run other CPU or memory intensive jobs. The configuration of the master node and processors were earlier presented in Section 3.1, and the coefficients of the fitting curves and the Gamma distributions are stated in Eqn. 2 and Table 1 respectively. All the video watching requests are generated from a VLC Media Player [3]. To test the performance of the proposed scheduling algorithm, we used a testing dataset with 400 videos, which consisted of overall 20,000 video segments. The number of segments in each individual video varies from 10 to 197, which means that the video lengths varies from around 30 seconds to 16 minutes.

5.2.2 Performance Evaluation

In the following experiments, we use two scenarios which commonly happen among video hosting services: (1) the videos (e.g., historical videos) are only uploaded to the server and none of them is being watched before all alternative bitrates are transcoded; and (2) the videos (usually news clips and live sports events) are watched shortly after being uploaded and not all required bitrates are prepared yet. Table 3 summarizes the parameters used in the experiments with the synthetic dataset.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream 1 (uploading)</td>
<td>MPEG-4 video, AAC audio</td>
</tr>
<tr>
<td>Number of uploaders (M_1)</td>
<td>6</td>
</tr>
<tr>
<td>Uploading frequency (\Delta t)</td>
<td>1 segment per second</td>
</tr>
<tr>
<td>Stream 2 (uploading)</td>
<td>MPEG-4 video, AAC audio</td>
</tr>
<tr>
<td>Number of uploaders (M_2)</td>
<td>7</td>
</tr>
<tr>
<td>Uploading frequency (\Delta t)</td>
<td>1 segment per second</td>
</tr>
<tr>
<td>Stream 3 (uploading)</td>
<td>MPEG-4 video, AAC audio</td>
</tr>
<tr>
<td>Number of uploaders (M_3)</td>
<td>8</td>
</tr>
<tr>
<td>Uploading frequency (\Delta t)</td>
<td>1 segment per second</td>
</tr>
<tr>
<td>Mean inter-arrival time (\lambda)</td>
<td>50 seconds</td>
</tr>
<tr>
<td>Video Trans. Manner (VTM)</td>
<td>0: low and medium together 1: first low then medium 2: first medium then low 3: only low 4: only medium</td>
</tr>
<tr>
<td>Video available latency (L_{avg})</td>
<td>5, 10, 15 seconds</td>
</tr>
<tr>
<td>Targeted video bitrates (B_{vt})</td>
<td>Medium bitrate 480×360, 768 kbps Low bitrate 360×240, 256 kbps</td>
</tr>
<tr>
<td>Request arrival rate (N_{arr})</td>
<td>9 per second</td>
</tr>
</tbody>
</table>

Table 3: Parameters used in the experiments to analyze the system capacity.

Scenario 1: No video watching requests. In this scenario, we tested the performance of the scheduler on the process of the videos being uploaded to the cloud and none of them being requested for watching by end-users. Since no video watching requests arrived, all the segments are transcoded to medium and low bitrates simultaneously, which means that each segment needs to be transferred between the storage repository and the processors once. The video uploading streams are generated by \(M_1\) video uploaders and each of them submits one segment every \(\Delta t\) seconds until all the segments are uploaded to the server. The arrival time of the first uploaded segment from each uploader follows a Poisson distribution with mean inter-arrival time of \(\lambda = 50\) seconds.

We first analyze the system capacity by comparing the latencies of uploading Streams 1 to 3. Figures 7(a), 7(c), and 7(e) show the number of segments received at the server side per second, and for each stream, the server works at the maximum workload for a period of time. Figures 7(b), 7(d), and 7(f) show the latency (both the queueing time and the completion time) of each uploaded segment for Streams 1 to 3, respectively. When the maximum arrival rate of segments is 6 (Figure 7(b)), the queueing time of each segment is fairly small and the completion time also remains less than 10 seconds during the first 50 minutes. We can conclude that receiving six segments per second is within the capacity of the current system, and the scheduler can support near-live streaming because of the small segment completion latency. Figure 7(d) shows that when the maximum arrival rate increases to 7, both the queueing time and the completion time for the segments grow. This indicates that the workload in Stream 2 exceeds the system capacity. The uploaded segments start to queue up and the queue size grows as time elapses. Compared with the latency when the arrival rate is 7, Stream 3 (Figure 7(f)) overloads the system more as the growth rate of the latency is larger than that of Stream...
Figure 6: Comparison on the segment queueing time and completion between the proposed algorithm and FIFO algorithm.

2. The latency decreases as soon as the workload is smaller than 7. One important observation is that there exist two situations of completion time increase: (a) the completion time increases with the same pattern as the queueing time (shown in Figures 7(d) and 7(f)), and (b) the completion time increases while the queueing time remains a small and relatively constant value (illustrated in the last minutes of Figures 7(b), 7(d), and 7(f)). Situation (a) is due to the growing queue size as the segment uploading rate is larger than the system capacity and hence it represents an accumulation of the previous transcoded segments. On the other hand, situation (b) is due to the long duration of VTT for specific segments. For all these three streams, VTT of the last segments are almost the same. Consequently, the system capacity is sensitive to the VTT of each segment. The scheduler needs to be smart enough to find the fastest processors to run urgent jobs.

<table>
<thead>
<tr>
<th>Stream</th>
<th>LBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream 1</td>
<td>0.0199</td>
</tr>
<tr>
<td>Stream 2</td>
<td>0.0036</td>
</tr>
<tr>
<td>Stream 3</td>
<td>0.0132</td>
</tr>
</tbody>
</table>

Table 4: Statistics on LBF for each stream.

We next present the load balancing results of the system. Table 5 illustrates the statistics of all processors to transcode videos in Stream 2 as an example. From the table, we observe that although the number of jobs completed by each processor varies from 550 to 946, the overall VTT for each processor only differs little. Some processors execute more jobs and transcode a higher duration of videos than others. The same situation occurs when uploading Stream 1 and Stream 3. We can conclude that the scheduler performs well with respect to load balancing. Table 4 summarizes the value of LBF for processing Streams 1 to 3. When comparing the LBF of these three streams, the scheduler achieves the best load balance when transcoding Stream 2. It is inferred that the scheduler works well when the system reaches its full capacity, and the performance decreases by a small percentage when the system is less or more loaded.

Scenario 2: Videos are requested for watching while they are being uploaded to the server. In this scenario, we study the performance of the scheduler while some of the videos are requested for watching before all the segments are transcoded. One assumption is that the clients do not interact with the videos during playback, such as pausing and forward/backward seeking. This means that each video is watched from the beginning to the end. Note that the processors in Scenario 2 are working under different workloads during the experiments, indicating that the VTT for the same video segment might vary on different processors. Therefore, the scheduler needs to search for the fastest processors to transcode videos that are being watched.

<table>
<thead>
<tr>
<th>Client ID</th>
<th>$L_{vo}$</th>
<th>$L_s$</th>
<th>$N_{QS}$</th>
<th>$N_{RE}$</th>
<th>$T_{mf}$</th>
<th>MOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>4.26</td>
<td>1</td>
<td>1</td>
<td>3.54</td>
<td>2.08</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>8.51</td>
<td>0</td>
<td>0</td>
<td>3.66</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>13.11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3.35</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>4.78</td>
<td>1</td>
<td>8</td>
<td>3.14</td>
<td>3.10</td>
</tr>
<tr>
<td>10</td>
<td>9.05</td>
<td>1</td>
<td>3</td>
<td>2.05</td>
<td>0.44</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>16.08</td>
<td>1</td>
<td>2</td>
<td>3.41</td>
<td>0.56</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6.96</td>
<td>1</td>
<td>3</td>
<td>1.24</td>
<td>0.66</td>
</tr>
<tr>
<td>10</td>
<td>11.44</td>
<td>1</td>
<td>2</td>
<td>2.6</td>
<td>1.70</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>14.99</td>
<td>1</td>
<td>1</td>
<td>2.17</td>
<td>1.51</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>7.15</td>
<td>1</td>
<td>3</td>
<td>3.16</td>
<td>1.93</td>
</tr>
<tr>
<td>10</td>
<td>10.69</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2.77</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>14.32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3.27</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6.82</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3.77</td>
</tr>
<tr>
<td>10</td>
<td>8.47</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3.66</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>13.15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3.35</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Measured values for the evaluation metrics on varying $L_{vo}$.

We first tested the extreme condition that videos are requested to be watched as soon as they are available. In this case, we used Stream 2 to upload videos and generated five clients to send video watching requests when the system reached its maximum capacity. The requests are sent to the server as soon as the initial $L_{vo}$ seconds of video are transcoded, and we keep $Bitr$ as the medium bitrate. The value of $Bitr$ changes to the low bitrate only when the required medium rate is not available and it will never change back again. Table 6 summarizes the values of the evaluation metrics of each client when $L_{vo}$ changes. Figure 8 shows the VTT normalized by segment duration for all the clients and the average among processors running jobs in NQueue at the time interval since Stream 2 starts to upload videos. When comparing Clients 1 to 5, a small $L_{vo}$ always causes quality switches during the playback for the subsequent segments. The other outlier is for Client 5 whose requests are
received last. The reason is the scheduling algorithm carried out by the server. Since the scheduler always searches for the fastest CPU and reserves enough processors to run jobs in PQ, the processing speed of processors that transcode for Clients 1 to 4 are faster than or at least equal to that for Client 5. Therefore, transcoding segments for Client 5 on any processors causes no quality switch. On the other hand, when segments requested by other clients are transcoded by the processors joined due to Client 5 sending a video watching request, the playback might be interrupted due to the slower speed of the these processors. For example, there exist jitters with the normalized VTT for Clients 2 to 4, which indicates that the video transcoding speed varies significantly during the playback. This situation hence causes quality switches and rebuffering events on Clients 2 to 4. When comparing the values of MOS, there are no best parameters. Small values of $L_{va}$ enable end-users to access the video shortly after it is uploaded, while a large $L_{va}$ supports smooth playback better. Consequently, we need to consider the trade-off among these parameters and adjust them based on the target application. In the following experiments, we set $L_{va}$ to 10 seconds as it neither delays the playback too long nor causes too many interrupts.

We next show that the proposed scheduler distributes urgent jobs to the fastest processors. Figure 8 compares the normalized VTT among processors transcoding segments for
Table 5: Statistics on each processor for Stream 2. The similarity of the overall VTT shows that the workloads among processors are well balanced, while the normalized VTT and the overall duration differentiate the processing capacity of each individual processor.

<table>
<thead>
<tr>
<th>Node ID</th>
<th># of jobs</th>
<th>Overall VTT (s)</th>
<th>Mean normalized VTT (s)</th>
<th>Median normalized VTT (s)</th>
<th>Overall duration (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>810</td>
<td>2963.9</td>
<td>0.889</td>
<td>0.811</td>
<td>3628.2</td>
</tr>
<tr>
<td>2</td>
<td>832</td>
<td>2965.2</td>
<td>0.857</td>
<td>0.788</td>
<td>3737.4</td>
</tr>
<tr>
<td>3</td>
<td>718</td>
<td>2973.4</td>
<td>1.016</td>
<td>0.913</td>
<td>3183.2</td>
</tr>
<tr>
<td>4</td>
<td>860</td>
<td>2957.9</td>
<td>0.832</td>
<td>0.745</td>
<td>3874.6</td>
</tr>
<tr>
<td>5</td>
<td>946</td>
<td>2954.4</td>
<td>0.745</td>
<td>0.682</td>
<td>4278.3</td>
</tr>
<tr>
<td>6</td>
<td>797</td>
<td>2968.1</td>
<td>0.927</td>
<td>0.823</td>
<td>3543.2</td>
</tr>
<tr>
<td>7</td>
<td>777</td>
<td>2967.5</td>
<td>0.917</td>
<td>0.830</td>
<td>3498.9</td>
</tr>
<tr>
<td>8</td>
<td>921</td>
<td>2951.6</td>
<td>0.797</td>
<td>0.701</td>
<td>4110.8</td>
</tr>
<tr>
<td>9</td>
<td>725</td>
<td>2958.0</td>
<td>0.993</td>
<td>0.898</td>
<td>3234.5</td>
</tr>
<tr>
<td>10</td>
<td>714</td>
<td>2964.9</td>
<td>1.003</td>
<td>0.910</td>
<td>3163.8</td>
</tr>
<tr>
<td>11</td>
<td>727</td>
<td>2973.3</td>
<td>1.181</td>
<td>1.101</td>
<td>2708.9</td>
</tr>
<tr>
<td>12</td>
<td>604</td>
<td>2948.2</td>
<td>0.819</td>
<td>0.739</td>
<td>3909.2</td>
</tr>
<tr>
<td>13</td>
<td>854</td>
<td>2944.4</td>
<td>0.815</td>
<td>0.753</td>
<td>3842.5</td>
</tr>
<tr>
<td>14</td>
<td>657</td>
<td>2965.7</td>
<td>1.078</td>
<td>0.980</td>
<td>2938.4</td>
</tr>
<tr>
<td>15</td>
<td>809</td>
<td>2950.8</td>
<td>0.887</td>
<td>0.798</td>
<td>3629.4</td>
</tr>
<tr>
<td>16</td>
<td>846</td>
<td>2945.6</td>
<td>0.835</td>
<td>0.773</td>
<td>3801.3</td>
</tr>
<tr>
<td>17</td>
<td>898</td>
<td>2948.9</td>
<td>0.797</td>
<td>0.720</td>
<td>4020.1</td>
</tr>
<tr>
<td>18</td>
<td>926</td>
<td>2948.7</td>
<td>0.794</td>
<td>0.703</td>
<td>4121.2</td>
</tr>
<tr>
<td>19</td>
<td>886</td>
<td>2958.2</td>
<td>0.810</td>
<td>0.731</td>
<td>3973.2</td>
</tr>
<tr>
<td>20</td>
<td>887</td>
<td>2952.1</td>
<td>0.812</td>
<td>0.728</td>
<td>3936.9</td>
</tr>
<tr>
<td>21</td>
<td>648</td>
<td>2971.3</td>
<td>1.106</td>
<td>1.028</td>
<td>2907.8</td>
</tr>
<tr>
<td>22</td>
<td>550</td>
<td>2988.2</td>
<td>1.283</td>
<td>1.193</td>
<td>2483.7</td>
</tr>
<tr>
<td>23</td>
<td>833</td>
<td>2900.3</td>
<td>0.845</td>
<td>0.777</td>
<td>3782.1</td>
</tr>
<tr>
<td>24</td>
<td>912</td>
<td>2943.9</td>
<td>0.775</td>
<td>0.701</td>
<td>4085.6</td>
</tr>
</tbody>
</table>

Figure 8: Comparison of the normalized video transcoding time VTT ($L_{va} = 10$ seconds).

clients and those for normally uploaded segments. For most of the time, the normalized VTT is smaller for Clients 1 to 5 than that for NQueue, indicating that the fastest processors are used to transcode video segments for the watching clients. On the other hand, the average VTT for NQueue increases when the server receives video watching requests. The reason is that only the slowest processors are assigned to run jobs in NQueue.

Finally, we illustrate the benefits of the automatic VTM switching functionality. Figure 9 shows the comparison between the deadline and completion time of transcoding. At the beginning, after the server receives the video watching request from Client 2, the time difference between the two curves increases since faster processors transcoding for Client 1 can serve Client 2. After Clients 3 to 5 join, some slower processors also serve Client 2. The sharp VTT changes on Client 2 and the increasing VTT result in a quality switch when Segment #38 is finished transcoding. After the quality switch, Client 2 changes its request from medium to low bitrate and the scheduler only changes VTM from 2 to 1. This strategy shortens the completion time for the required bitrate for a single transcoding job but does not help subsequentially in transcoding since the overall VTT on the processor remains the same. The next three rebuffering events also happen for the same reason. Conversely, when the scheduler switches VTM after encountering the first rebuffering event, the transcoding completion time is shortened significantly. This can improve the performance of the scheduler and the system throughput, as well as improve end-users’ experiences. However, the drawback of automatic VTM is that the system might suffer from frequent changing of the required bitrates, which seldom happens.

6. CONCLUSIONS AND FUTURE WORK

In this study we proposed a dynamic scheduling algorithm for video transcoding jobs designed to support Dynamic Adaptive Streaming over HTTP in a cloud environment. We first modeled the video transcoding time VTT estimation based on statistics from video segment durations and the targeted bitrate. The scheduler then dynamically distributes video transcoding jobs VTJ according to VTT estimation and the system workload. Overall, the proposed
scheduler can support near live streaming while balancing the workload and provide smooth and seamless playback. Most importantly it can dynamically serve requests and adjust the video transcoding mode $VTM$ accordingly. Experimental results show that the scheduler can distribute urgent jobs to the fastest processors and shorten the transcoding completion time by using $VTM$ switching. Multiple configurations of the scheduler allow it to be adjusted according to the needs of different applications. In the future, we plan to investigate a cost model for the scheduler, which would be helpful for administrators to select appropriate cloud services while considering multiple parameters.

Acknowledgements

This research has been supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office. The authors would also like to thank Professor Junchang Xin, from Northeastern University, China for his kindly support on providing the cloud computing resources.

7. REFERENCES