Terakreditasi SINTA Peringkat 2
Surat Keputusan Dirjen Penguatan Riset dan Pengembangan Ristek Dikti No. 10/E/KPT/2019
masa berlaku mulai Vol. 1 No. 1 tahun 2017 s.d Vol. 5 No. 3 tahun 2021

Terbit online pada laman web jurnal: **http://jurnal.iaii.or.id**

JURNAL RESTI
(Rekayasa Sistem dan Teknologi Informasi)
Vol. 5 No. 3 (2021) 413 - 420 ISSN Media Elektronik: 2580-0760

Adaptive Streaming Server dengan FFMPEG dan Golang

David Kristiadi[1], Marwiyati[2]
[1,2]Manajemen Teknik Studio Produksi, Jurusan Penyiaran, Sekolah Tinggi Multi Media Yogyakarta
[1]davk@mmtc.ac.id, [2]marwiyati@mmtc.ac.id

Abstract

Quality of experience (QoE) when accessing video streaming becomes a challenge in varieties of network bandwidth/speed. Adaptive streaming becomes an answer to gain good QoE. An architecture system of the adaptive streaming server with Dynamic Adaptive Streaming over HTTP (DASH) was proposed. The system was consists of two services e.g transcoding and streaming. Transcoding service encodes an audio file, multi-bitrates video files, and manifest.mpd files. Streaming service serves client streaming requests that appropriate to client network profiles. The system is built using the Golang programming environment and FFMPEG. Transcoding service has some execution mode (serial and concurrent) and passing mode (1 pass and 2 passes). The transcoding service test results show that concurrent execution is faster 11,5% than the serial execution and transcoding using 1 pass is faster 46,95% than 2 passes but the bitrate of output video lower than the determinate bitrate parameter. The streaming service has a good QoE. In the 5 scenarios, buffer level=0 happens 5 times, and its total duration is 64 seconds. Buffer level=0 happens when extreme changes happen in network speed from fast to too slow.

Keywords: Adaptive Streaming, Video Streaming, DASH, FFMPEG, Golang

Abstrak

Kualitas pengalaman pengguna (QoE) dalam mengakses *video streaming* menjadi salah satu tantangan di tengah bervariasinya *bandwidth*/kecepatan jaringan. *Adaptive streaming* merupakan salah satu jawaban dalam mencapai QoE yang baik. Sebuah rancangan sistem *adaptive streaming server* dengan konsep *Dynamic Adaptive Streaming over* HTTP (DASH) disusun untuk menjawab masalah ini. Sistem terdiri dari dua layanan yaitu *transcoding* dan *streaming*. Layanan *transcoding* bertugas menyiapkan *file* audio, *file-file* video *multi bitrate* dan *file* manifest.mpd. Layanan *streaming* bertugas melayani permintaan *streaming* yang telah disesuaikan dengan kondisi jaringan klien. Sistem yang dibangun menggunakan FFMPEG dan lingkungan pemrograman Golang. Layanan *transcoding* dilengkapi beberapa kemampuan eksekusi (serial dan konkuren) dan *passing mode* (1 *pass* & 2 *pass*). Pada ujicoba layanan *transcoding* diperoleh bahwa eksekusi secara konkuren lebih cepat 11,5% dibandingkan secara serial. *Transcoding* dengan 1 *pass* dibandingkan 2 *pass* memiliki durasi lebih cepat 46,95% tetapi *bitrate* video hasil *transcoding* jauh di bawah parameter *bitrate* yang ditentukan. QoE layanan *streaming* cukup baik. Dari 5 skenario terjadi 5 kali *buffer level*=0 dengan total durasinya 64 detik. *Buffer level*=0 terjadi ketika ada perubahan yang ekstrem terhadap kecepatan jaringan dari yang semula cepat ke sangat lambat.

Kata kunci: Adaptive Streaming, Video Streaming, DASH, FFMPEG, Golang.

1. Pendahuluan

Publikasi hasil produksi video dapat melalui berbagai cara, salah satunya adalah melalui *video streaming*. *Video streaming* merupakan cara mengakses *file* video tanpa harus menunggu file tersebut selesai diunduh[1]. Pengguna internet semakin bertambah seiring dengan perkembangan teknologi pada perangkat mobile dan komputer. Konten video merupakan konten internet yang paling sering dikunjungi oleh pengguna internet[2]. Konten tersebut biasanya diakses dalam sebuah platform yang disebut sebagai *Video on Demand* (VOD). Beberapa layanan VOD yang umum diakses YouTube, Vimeo, dan Netflix[3–5].

Agar video streaming dapat dinikmati dengan baik oleh pengguna maka *bandwitdh*/kecepatan jaringan harus dapat mengakomodasi *bitrate* video. Kualitas kecepatan jaringan berbanding lurus dengan kualitas video. Hanya saja kecepatan jaringan tidak selalu konstan. Hal tersebut dipengaruhi oleh infrastuktur jaringan dan kepadatan jaringan internet dilokasi pengguna mengakses *video streaming*. Jika video diakses secara

Diterima Redaksi: 05-03-2021 | Selesai Revisi: 14-05-2021 | Diterbitkan Online: 20-06-2021

mobile maka kecepatan jaringan juga dipengaruhi oleh kualitas sinyal WIFI atau jaringan seluler.

Ketika proses *video streaming* berlangsung dan kondisi kecepatan jaringan tidak dapat mengakomodasi *bitrate video* maka akan terjadi *delay* dan *buffering*. *Delay* dan *buffering* tersebut mengurangi kualitas pengalaman pengguna atau *Quality of Experience* (QoE) dalam layanan *video streaming*[6]. Jika jaringan internet publik yang digunakan maka langkah memperlebar *bandwidth*/mempercepat kecepatan jaringan bukan solusi yang tepat karena akan berbenturan dengan banyak kepentingan.

Beberapa teknologi dasar *streaming* dikembangkan untuk mencapai QoE layanan *video streaming* yang baik. Terdapat tiga macam teknologi dasar *streaming* yaitu *progresive download*, *streaming* dan *adaptive streaming*[4]. Pada *progresive download*, *server* mengirim keseluruhan *file* melalui HTTP setelah ada permintaan klien menggunakan HTTP GET[4, 7]. Klien segera memutar *file* audio/video yang telah di download. Pada *streaming*, *server* mengirimkan potongan video berdasarkan permintaan dari klien[4]. Pada *adaptive streaming*, *server* mengirimkan potongan video berdasarkan *request* klien yang telah disesuaikan dengan kondisi jaringan klien[4, 7].

Agar prinsip layanan *adaptive streaming* dipenuhi, maka *file* video di dalam *server* di-*encode* ke berbagai format *bitrate* dan sebuah *file manifest* digeneralisasi[4, 8]. *File manifest* berisi informasi dari representasi media yang berupa *bandwitdh*, ukuran *frame*, lokasi *file* video, *segment base* dan *initial range*. Ketika proses *streaming* dimulai, klien menerima *file manifest* dan menganalisis *file* tersebut untuk mengetahui *segment bitrate* yang tersedia di *server*. Kemudian klien memberikan *request segment* ke *server* yang video sesuai dengan kondisi jaringan[9, 10].

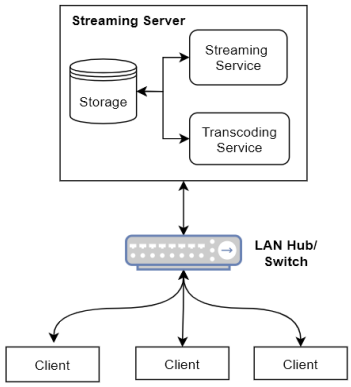
Terdapat dua bentuk teknologi *Adaptive Streaming* paling populer yaitu HTTP *Live Streaming* (HLS), dan *Dynamic Adaptive Streaming over* HTTP (DASH)[4]. DASH dikembangkan oleh *Moving Picture Expert Group* (MPEG) sehingga sering dikenal sebagai MPEG-DASH. Perkembangan selanjutnya MPEG-DASH menjadi standar dalam layanan *adaptive video streaming*. Beberapa implementasi MPEG-DASH sudah banyak dilakukan beberapa contoh diantaranya menggunakan FFMPEG[5], libdash[11], dan MP4Box dari GPAC[12]. FFMPEG merupakan salah satu *software* yang dapat digunakan secara *cross platform* dan memiliki kemampuan yang sangat lengkap dalam mengolah audio video seperti merekam, *convert* dan *streaming*[13, 14].

Supaya tercapai QoE yang baik dalam layanan video *streaming*, dibutuhkan *server* yang mampu menyediakan *file* video *multi bitrate* dan melayani permintaan *file* video sesuai dengan kebutuhan klien. Penyediaan *file* video multi *bitrate* dapat dilakukan dengan menggunakan FFMPEG. Tetapi FFMPEG membutuhkan lingkungan pemrograman untuk mengotomatisasi pengolahan audio video. Beberapa lingkungan pemrograman yang telah digunakan yaitu Javascript[5], PHP[15] dan Shell Script[14].

Salah satu lingkungan pemrograman yang digunakan untuk mewujudkan konsep *adaptive streaming server* adalah Golang. Golang menyediakan *library* untuk eksekusi secara *concurent* (konkuren) dan pembuatan web server[16–19]. Tetapi penggunaan FFMPEG dan Golang perlu dirancang dan diimplementasikan secara tepat supaya *adaptive streaming server* yang bangun dapat bekerja dengan efektif dan efisien.

2. Metode Penelitian

Rancangan sistem *adaptive streaming server* seperti pada Gambar 1. Implementasi rancangan menggunakan lingkungan pemrograman Golang dan menggunakan FFMPEG *tools*. Prototipe yang dibuat kompatibel dengan Golang 1.13. Sistem yang dibuat terdiri dari sebuah *streaming server* yang dapat diakses oleh beberapa client dalam sebuah LAN. *Streaming server* sendiri memiliki dua komponen utama yaitu *streaming service* dan *transcoding service*.



Gambar 1 Rancangan Sistem *Streaming Server*

2.1. *Streaming Service*

Streaming service melayani permintaan dari klien. Permintaan klien berupa *file manifest* (*.mpd) dan *file* video (*.webm) dari *hardisk server*. Layanan *manifest file* yang dikerjakan seperti pada Program serveMpd. Program tersebut memastikan *file* yang diakses ada dan jika *file*-nya ada dilanjutkan dengan mengirimkan *file manifest* kepada klien.

```
Program serveMpd
Input: w http.responsewriter, url string
Output: -
   Get manifest param from url
   mfFile,err := getManifestFile(manifest)
   If err!=nil {
      http.Error(w, err)
      return
```

DOI: https://doi.org/10.29207/resti.v5i3.2998

```
    }
    defer mfFile.Close()
    setHTTPHeader(w,mfFile)
    io.copy(w,mfFile)
```

Sedangkan program yang melayani *request file* video seperti pada Program serveVideo. Setelah *file manifest* dibaca oleh klien. Klien akan mengirimkan permintaan *file* video (*.webm) kepada *server* mendekati kecepatan jaringan. Setelah permintaan *file* video oleh klien dikenali oleh server, server mengirimkan *file* video yang dimaksud kepada klien.

```
Program serveVideo
Input: w http.responsewriter, r *http.Request
Output: -
    Get params from r.URL.Path
    Get mediaFile from params
    http.ServeFile (w, r,file)
    io.copy(w,mfFile)
```

Program serveMpd & serveVideo dikoordinasi oleh fungsi pada seperti pada Program mpdHandler. Penggunaan satu fungsi ini akan memudahkan dalam proses layanan *streaming*, dikarenakan *request file* mpd dan video dikirimkan pada alamat URL yang sama.

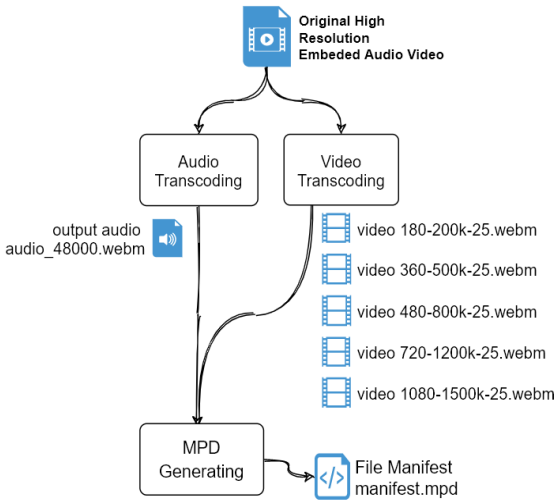
```
Program mpdHandler
Input: w http.responsewriter, r *http.Request
Output: -
    t:=validVideoFile(r.URL.Path)
    if t==nil {
       serveMpd(w, r.URL,Path)
       return
    }
    serveVideo(w,r)
```

2.2. *Transcoding Service*

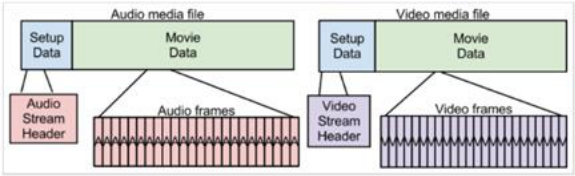
Transcoding service bertugas melayani proses *video transcoding* dan *audio transcoding* serta melayani pembuatan *manifest file*. *File* video hasil *transcoding* memiliki parameter *output* yang berbeda-beda. Parameter *output* yang dimaksud adalah *framesize*, *framerate* dan *bitrate*. Tahapan yang dikerjakan pada *transcoding service* digambarkan seperti pada Gambar 2. Terdapat tiga proses utama yaitu *audio transcoding*, *video transcoding* dan *mpd file generating*.

Transcoding service menggunakan FFMPEG untuk proses *transcoding* dan *mpd generating*. Pada proses *transcoding* dengan FFMPEG, digunakan *library* libvpx_v9 untuk *transcoding* video dan libvorbis untuk *transcoding* audio. Supaya proses *transcoding* berjalan sesuai tahapan *transcoding service*, pemanggilan FFMPEG dilakukan pada bahasa pemrograman Golang dengan *library* exec.

Proses *video transcoding* memiliki dua tipe *transcoding* yaitu 1 *pass* dengan baris perintah seperti pada Program VideoTranscoding1Pass dan 2 *pass* dengan baris perintah seperti pada Program VideoTranscoding2Pass. Baris perintah tersebut diulangi dengan mengubah parameter *framesize*, *bitrate* dan *framerate*. Kedua program menghasilkan *file* video (*no* audio) dengan ukuran *framesize* dan *bitrate* yang berbeda.



Gambar 1 *Transcoding* dan *Generating Manifest File*



Gambar 2. Ilustrasi File Hasil *Transcoding* Dengan *Muxed* dan *Chunked* [20]

Transcoding 1 pass memiliki tahap *transcoding* hanya sekali dan menghasilkan file video. Pada *transcoding* 2 *pass* dilakukan 2 kali tahap *transcoding*. Pada *pass* tahap 1 dilakukan analisa data input video menghasilkan log yang berisi data *bitrate* yang tepat untuk tahap berikutnya. Pass tahap 2 dilakukan *transcoding* video sesuai informasi yang diperoleh pada tahap 1 dan menghasilkan file video. Proses *transcoding* 1 *pass* memiliki durasi yang lebih cepat tetapi kualitas video lebih kurang.

```
Program videoTranscoding1Pass
Input: videoInputFile vi, videoOuputPath vo,
bitrate bt, screensize sz, framerate fr
Output: resultStatus

  command := []String {"-hide_banner", "-y"
    "-i", vo "-c:v", "libvpx-vp9",
    "-s", sz, "-b:v",
    bt, "-r", fr,
    "-row-mt", "1",
    "-keyint_min", "150",
    "-g", "150",
    "-tile-columns", "4",
    "-frame-parallel", "1 ",
    "-movflags", "faststart",
    "-frag_duration", "5",
    "-speed" , 2,"-threads",
    "4", "-an",
    "-f", "webm", "dash", "1",
    vo+"/videoname+"sz +"_"+
    bt +"_"+".webm"}
  return exec.Command(ffmpeg, command)
```

DOI: https://doi.org/10.29207/resti.v5i3.2998

```
Program videoTranscoding2Pass
Input: videoInputFile vi, videoOuputPath vo,
bitrate bt, screensize sz, framerate fr
Output: resultStatus

  command := []String {"-hide_banner", "-y"
    "-i", vo "-c:v", "libvpx-vp9", "-s", sz,
    "-b:v", bt, "-r", fr, "-row-mt", "1",
    "-keyint_min", "150", "-g", "150",
    "-tile-columns", "4", "-frame-parallel",
    "1 ", "-movflags", "faststart",
    "-frag_duration", "5", "-threads", "4",
    "-an", "-f", "webm", "dash", "1" }

  pass1Cmd:= []string{ "-pass", "1",
    "-speed", 4, "/dev/null"}
  pass2Cmd:= []string{ "-pass", "2",
    "-speed", 2, vo+"/videoname+"sz +"_"+
    bt +"_"+".webm"}

  resStatus1:= exec.Command(ffmpeg,
    merge(command,pass1Cmd))
  resStatus2:= exec.Command(ffmpeg,
    merge(command,pass2Cmd))
  return resStatus1 & resStatus2
```

Baris perintah pada proses *audio transcoding* seperti pada Program *audioTranscoding*. Hasil dari eksekusi baris perintah tersebut menghasilkan *file* audio (no video).

```
Program audioTranscoding
Input: videoInputFile vi, videoOuputPath vo
Output: resultStatus

  command := []String {"-hide_banner", "-y"
    "-i", vo "-c:v", "libvorbis",
    "ar", "48000", "-vn", "-f",
    "webm", "dash", "1",
    vo+"/videoname"+"48k" +".webm"}
  return exec.Command(ffmpeg, command)
```

Sebelum proses *mpd file generating*, dilakukan proses pemanggilan program *video transcoding*, dilanjutkan *audio transcoding*. Pemanggilan program ini dapat dilakukan secara serial dan *concurent* (konkuren). Pada pemanggilan program secara serial, eksekusi program *video transcoding* dan *audio transcoding* dimulai dan diakhiri secara berurutan. Setelah diperoleh *file* hasil *transcoding* dengan parameter *output* video pertama dilanjutkan ke *transcoding* ke parameter *output* audio/video selanjutnya. Proses tersebut diulangi hingga dihasilkan *file* - *file* hasil *transcoding* sesuai dengan parameter - parameter *output* audio/video yang ditentukan.

Pada pemanggilan program *transcoding* secara konkuren, program *video transcoding* dan *audio transcoding* dipanggil beberapa kali dengan waktu hampir bersamaan. Ketika proses pemanggilan program *transcoding*, parameter input yang digunakan berbeda-beda sesuai dengan parameter – parameter *output* audio/video yang ditentukan. Pemanggilan program *transcoding* secara konkuren menggunakan fasilitas Goroutine dari Golang. Penggunaan Goroutine memungkinkan, program dapat menangani beberapa pekerjaan sekaligus [17, 18, 21].

Setelah program *video transcoding* dan *audio transcoding* selesai dieksekusi, dilanjutkan ke proses *generating* *.mpd *file*. Baris perintah tahap ini seperti pada Program generatingMPD. Di dalam baris perintah tersebut seluruh *file* video dan audio yang akan di-*streaming*-kan dipanggil.

```
Program generatingMPD
Input: transcodedVideos tv[], transcodedAudio
ta, manifestpath mp
Output: resultStatus

  command := []String { "-y",
    "-f","webm_dash_manifest", "-i", tv[0],
      ...
    "-f","webm_dash_manifest", "-i", tv[n],
    "-f","webm_dash_manifest", "-i", ta,
    "map", "0" ..., "map","n"
    "-f", "web_dash_manifest",
    "-adaption_sets",
    "id=0,streams=0,1 ,..len(tv)
     id=1,streams= len(tv)
    mp+"/manifest.mpd"}
   return exec.Command(ffmpeg, command)
```

2.3. Ujicoba dan Evaluasi

Untuk ujicoba *transcoding* digunakan video dengan properti *file* seperti pada Tabel 1. *File* video tersebut di-*transcoding* secara serial dan konkuren dengan 1 *pass* dan 2 *pass*. Durasi waktu dan ukuran *file* dan *bitrate* yang diperoleh kemudian dibandingkan untuk didapatkan metode *transcoding* dan tipe *pass* yang cepat.

Setelah *file* video di-*transcoding*, dilanjutkan ujicoba *adaptive streaming server*. Pada ujicoba ini digunakan sebuah halaman *website* yang dilengkapi dengan metrik yang menampilkan hasil pengukuran *buffer level* dan *video bitrate*. Halaman web untuk ujicoba merupakan sebuah halaman ujicoba dash *streaming* dari DASH Industri Forum [22]. Halaman tersebut di *copy* ke server lokal dengan tujuan agar supaya *file* *.mpd yang berada di *server* lokal dapat dimuat dan dijalankan.

Tabel 1. Properti Video Untuk Ujicoba

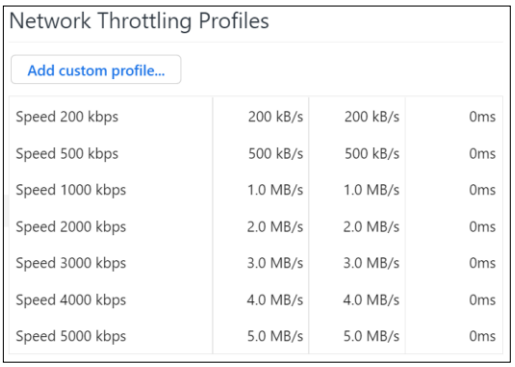
Video Properties	Nilai
Framesize	3840 x 2160
Bitrate	50087 kbps
Framerate	30 fps
Duration	10m 28s
Audio Sample rate	48000 Hz
File size	3.66 GB
Filename	video01.mp4

Web browser yang digunakan untuk proses ujicoba adalah Google Chrome. Untuk memvarisi kecepatan jaringan dilakukan konfigurasi *network throttling profile* (Gambar 3). Fasilitas tersebut dapat diakses dengan mengaktifkan *developer mode*.

Selama proses *streaming* berlangsung dilakukan perubahan kecepatan jaringan dengan mengubah profil *network throttling*. Setiap perubahan kecepatan jaringan dilakukan analisis terhadap kualitas video yang diterima.

DOI: https://doi.org/10.29207/resti.v5i3.2998

Setelah data ujicoba terkumpul akan dilakukan evaluasi untuk menentukan bahwa prototipe *server* berhasil atau tidak dalam melakukan *adaptive streaming* dengan melakukan analisis kualitas dari pengalaman. Terdapat 5 skenario yang digunakan, skenario tersebut seperti pada Tabel 2. Sk=Skenario uji coba *streaming*; W= perkiraan interval waktu ujicoba dalam menit, misalnya 0-2 merupakan interval waktu dari ±0 hingga ± 2 menit; B=kecepatan jaringan yang merepresentasikan kecepatan *upload* dan *download* jaringan dalam MB/s.



Gambar 3 Pengaturan Kecepatan Internet Pada Google Chrome

Tabel 2. Skenario Waktu dan Kecepatan Jaringan Video untuk Ujicoba

Sk – I		Sk – II		Sk – III		Sk – VI		Sk – V	
W	B	W	B	W	B	W	B	W	B
0-2	0,2	0-2	5	0-2	1	0-1	2	0-1	0,2
2-4	0,5	2-3	4	2-4	0,2	1-3	0,5	1-2	1
4-5	1	3-4	3	4-6	4	3-5	4	2-3	4
5-7	2	4-5	2	6-8	0,5	5-7	0,2	3-4	2
7-8	3	5-6	1	8-10	3	7-10	1	4-5	0,5
8-9	4	6-8	0,5					5-6	3
9-10	5	8-10	0,2					6-7	0,2
								7-8	4
								8-9	1
								9-10	3

Skenario Sk-I merupakan ujicoba *streaming* dari kecepatan 0,2 sampai 5 MB/s. Skenario Sk-II merupakan ujicoba *streaming* dari kecepatan 5 – 0,2 MB/s. Skenario Sk - III, Sk – IV dan Sk V merupakan skenario ujicoba *streaming* dengan kecepatan jaringan yang *random*.

3. Hasil dan Pembahasan

3.1. *Transcoding*

Proses *transcoding* dan pembuatan *manifest file* berjalan sesuai dengan harapan. *File* video *original* dapat di *transcoding* ke beberapa *bitrate*. Terdapat 5 *file* video dengan *framesize* dan *bitrate* yang berbeda serta fps yang sama yaitu 25 fps. Terdapat satu *file* audio dengan sampling 48KHz. Kontainer kelima *file* video dan *file* audio adalah *.webm. Selanjutnya terdapat satu *file* manifest.mpd yang merupakan hasil mpd *file generating*.

Proses *transcoding* membutuhkan waktu yang bervariasi sesuai dengan durasi video, *framesize* dan *bitrate* yang digunakan. Semakin lama durasi, semakin besar *framesize* dan *bitrate* maka semakin lama waktu yang dibutuhkan. Video yang digunakan untuk ujicoba (Tabel 2) di-*transcoding* menggunakan komputer dengan prosesor I3-7100U CPU 2.40GHz dan RAM 11.6 GB. Proses tersebut menghasilkan *file-file* video dengan durasi *transcoding* seperti pada Tabel 3 dan Tabel 4.

Tabel 3. Durasi Eksekusi *Transcoding* Video01 Secara Serial dengan 1 *Pass* dan *Pass* 2

Output Transcoding (* = video01)	Durasi	
	1 *Pass*	2 *Pass*
*180-25-300k.webm	12m5,3s	22m56,5s
*360-25-500k.webm	33m39,7s	1h2m39,2s
*480-25-800k.webm	1h6m24,6s	2h2m56,7s
*720-25-1500k.webm	1h47m34,7s	3h19m30,7s
*1080-25-3000k.webm	2h59m31,5s	5h46m51,8s
*audio.webm	3h9m0,6s	5h57m21,0s
manifest.mpd	3h9m1,2s	5h57m21.1s

Tabel 4. Durasi Eksekusi *Transcoding* Video01 Secara Konkuren Dengan 1 Pass dan 2 *Pass*

Output Transcoding (* = video01)	Time Elapsed	
	1 *Pass*	2 *Pass*
*audio.webm	10m29,2s	10m29,5s
*180-25-300k.webm	49m6,8s	1h37m58,8s
*360-25-500k.webm	1h24m22,3s	2h37m18,3s
*480-25-800k.webm	2h0m1,6s	3h35m52,2s
*720-25-1500k.webm	2h14m16,5s	4h4m57,8s
*1080-25-3000k.webm	2h47m52,2s	5h15m24,3s
manifest.mpd	2h47m53,7s	5h15m24,4s

Dalam Tabel 3 dan Tabel 4 tersaji durasi waktu dari awal eksekusi (secara serial dan konkuren) hingga output *file* dihasilkan. Durasi *transcoding* dengan cara konkuren baik 1 *pass* (K_1) memiliki durasi yang lebih cepat dibandingan dengan *transcoding* dengan cara serial 1 *pass* (S_1). Hal yang sama untuk transcoding secara konkuren 2 *pass* (K_2) lebih cepat dari pada transcoding secara serial 2 *pass* (S_2). Dengan menggunakan rumus 1 dan data durasi di akhir eksekusi (*output file manifest.mpd* dihasilkan), diperoleh bahwa efisiensi durasi *transcoding* secara konkuren dibandingkan dengan eksekusi *transcoding* secara serial (e_{ks}) sebesar 11,5%.

$$e_{ks} = \left(1 - \frac{K_1+K_2}{S_1+S_2}\right) x100\% \quad (1)$$

Selanjutnya, untuk mengetahui perbandingan kecepatan eksekusi 1 *pass* terhadap 2 ($e_{1,2}$) dihitung dengan rumus 2. Dari perhitungan tersebut diperoleh bahwa durasi *transcoding* dengan 1 *pass* lebih cepat 46,95% dibandingkan dengan *transcoding* dengan 2 *pass*.

$$e_{1,2} = \left(1 - \frac{K_1+S_1}{K_2+S_2}\right) x100\% \quad (2)$$

DOI: https://doi.org/10.29207/resti.v5i3.2998

Selain durasi *transcoding* 1 *pass* lebih cepat daripada 2 *pass*, diperoleh juga bahwa ukuran *file* hasil transcoding 1 *pass* lebih ringkas daripada 2 *pass* (Tabel 5). Rata-rata selisih *file* antara kedua metode *transcoding* adalah 11.32%. Perhitungan tersebut menggunakan rumus 3 dengan s_1 adalah ukuran file hasil 1 *pass*, s_2 adalah ukuran file hasil 2 *pass*, dan *n* adalah jumlah *file* video hasil transcoding, dan $\overline{s}$ adalah rata-rata selisih ukuran *file*. Umumnya *transcoding* 2 *pass* mendapatkan *file* yang lebih ringkas, tetapi pada kasus ini menunjukkan hasil yang berbeda.

$$\overline{s} = \frac{\sum_{n=1}^{5}\left(\frac{s2\_n}{s1\_n} - 1\right)}{n} x100\% \quad (3)$$

File video hasil kedua tipe *passing* menampilkan kualitas hasil pemutaran video yang mirip ketika diputar menggunakan *video player*. Tetapi ketika diperhatikan lebih detail, gambar pada video hasil *transcodi*ng 2 *pass* memiliki detail gambar yang lebih baik. Hal tersebut selaras dengan stastistik *file* video hasil *transcoding* 1 *pass* dan 2 *pass* pada Tabel 5. Pada Tabel 5 video hasil *transcoding* 2 *pass* memiliki nilai *bitrate* lebih tinggi. Dari data tersebut dapat dinyatakan bahwa *transcoding* 2 *pass* mendapatkan *bitrate* yang lebih dekat dengan parameter *bitrate* pada proses *transcoding.*

Tabel 5. Perbandingan Bitrate & File Size
hasil transcoding 1 pass dan 2 pass

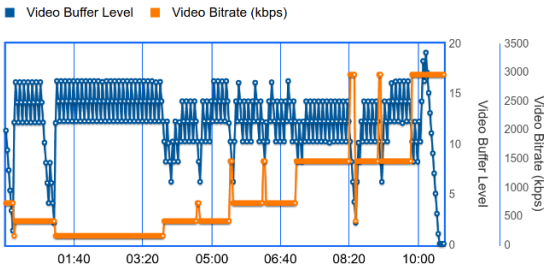
n	*Filename* (* = video01_)	*Bitrate*		File Size (MB)	
		1 *Pass*	2 *Pass*	1 *Pass*	2 *Pass*
1	*180-25-300k.webm	113 KB/s	123 KB/s	8,9	9,6
2	*360-25-500k.webm	314 KB /s	389 KB/s	24,9	30,6
3	*480-25-800k.webm	577 KB /s	639 KB/s	45,2	50,3
4	*720-25-1500k.webm	1207 KB /s	1262 KB/s	94,9	99,2
5	*1080-25-3000k.webm	2737 KB /s	2975 KB/s	215,2	233,9

3.2. *Streaming*

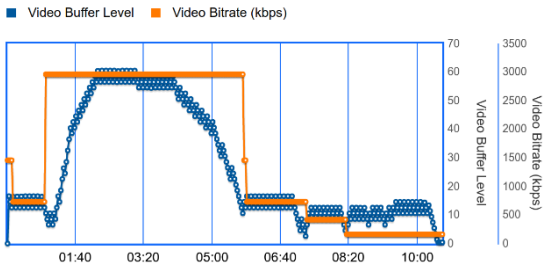
Ujicoba *adaptive streaming* dengan skenario SK-I hingga SK-V menunjukkan bahwa server dapat melayani permintaan *file* video sesuai dengan kecepatan jaringan jaringan klien. Permintaan *file* video tersebut diinisialisasi klien setelah halaman pemutar video mendapatkan informasi tentang kecepatan jaringan dan informasi dari *file* manifest.mpd. Metrik hasil ujicoba dapat dilihat pada Gambar 4 hingga Gambar 8. Uji coba skenario SK-I pada Gambar 4, SK-II pada Gambar 5, SK-III pada Gambar 6, SK-IV pada Gambar 7, dan SK-V pada Gambar 8.

Pada metrik Gambar 4 sampai Gambar 8 terlihat bahwa proses *adaptive streaming* berjalan dengan baik. Hal tersebut terlihat dari *video bitrate* yang diunduh oleh klien menyesuaikan kecepatan jaringan saat itu.

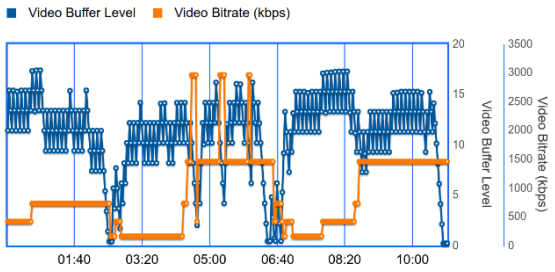
Pada metrik Gambar 4 sampai Gambar 8 juga teramati beberapa kali *buffer level* menyentuh level 0 seperti yang terangkum pada Tabel 6. *Buffer level* menyentuh level 0 disebabkan karena terjadi penurunan yang ekstrem dari kecepatan jaringan yang cepat ke kecepatan jaringan yang sangat lambat misal dari 1 MB/s ke 0,2 MB/s.



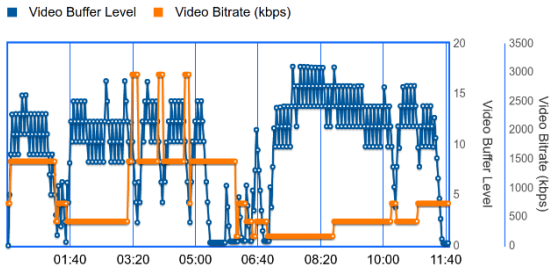
Gambar 4. Matrik Hasil *Streaming* Dengan Skenario SK – I



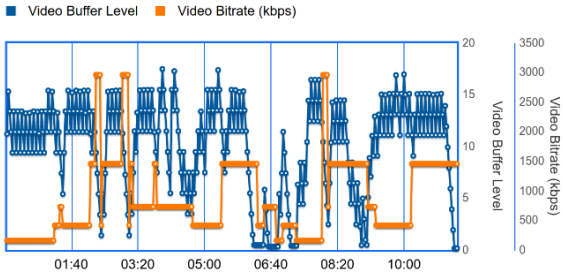
Gambar 5. Matrik Hasil *Streaming* Dengan Skenario SK – II



Gambar 6. Matrik Hasil *Streaming* Dengan Skenario SK – III



Gambar 7. Matrik Hasil *Streaming* Dengan Skenario SK – IV



Gambar 8. Matrik Hasil *Streaming* Dengan Skenario SK – V

Beberapa saat pada awal kecepatan jaringan yang lambat tersebut klien tetap berusaha mempertahankan

DOI: https://doi.org/10.29207/resti.v5i3.2998

permintaan *bitrate* yang sama hingga akhirnya *buffer level* = 0. Akibatnya pemutar video disisi klien akan menghentikan sementara pemutaran video dan menunggu *buffer level* > 0 untuk kembali memutar video. Ketika permintaan potongan video dengan *bitrate* yang sama tidak segera terpenuhi, permintaan klien selanjutnya adalah video dengan *bitrate* yang mendekati kecepatan jaringan dan menunggu sampai *buffer level* di atas 0 untuk kembali memutar video.

Tabel 6. Catatan Waktu dan Kasus Saat *Buffer Level* Menyentuh Level 0

Skenario	Perkiraan Waktu	Perkiraan Durasi (s)	Kasus
SK-III	2’	2	NP 1 ke 0,2 MB/s
SK-III	6’30” - 6’50”	20	NP 4 ke 0,5 MB/s
SK-IV	1’35”	2	NP 2 ke 0,5 MB/s
SK-IV	5’20” - 5’40”	20	NP 4 ke 0,2 MB/s
SK-V	6’30” - 6’50”	20	NP 3 Ke 0,2 MB/s

Ketika permintaan *file* video dengan *bitrate* yang mendekati kecepatan jaringan telah terpenuhi dan *buffer level* di atas 0, sesekali pada kecepatan jaringan sama, pemutar video melakukan ujicoba permintaan *file* video dengan *bitrate* yang lebih tinggi. Jika pemintaan potongan video tersebut dapat di *download* dan menyebabkan *buffer level* naik signifikan, permintaan video *bitrate* selanjutnya adalah *file* video dengan *bitrate* yang lebih tinggi. Hal tersebut juga berlaku kebalikannya. Apabila *buffer level* justru semakin turun, pemutar video akan meminta *file* video dengan *bitrate* yang lebih rendah sebelum *buffer level* benar-benar mencapai 0.

Dari beberapa skenario pengujian juga diperoleh bahwa kualitas pengalaman *video streaming* cukup baik. Dari kelima skenario pengujian, *buffer level* pemutar video jarang menyentuh level 0. Hal tersebut layanan *video streaming* tetap dapat dinikmati walaupun kecepatan jaringan saat itu tidak stabil.

4. Kesimpulan

Sistem *adaptive video streaming server* dibangun dengan konsep DASH menggunakan FFMPEG dan Golang. Sistem yang dibangun terdiri dari layanan utama yaitu layanan *transcoding* dan layanan *streaming*. Layanan *transcoding* dengan eksekusi secara konkuren memiliki durasi lebih cepat 11,5% dibandingkan dengan eksekusi secara serial. Metode *transcoding* 1 *pass* memiliki durasi 46,95% lebih cepat dari pada *transcoding* 2 *pass* dengan rata-rata ukuran *file* 11.32% lebih ringkas dan *video bitrate* sedikit jauh di bawah parameter *transcoding* dari pada *video bitrate* pada hasil *transcoding* dengan 2 *pass*.

Layanan *streaming* dapat melayani permintaan *file* video sesuai dengan kecepatan jaringan yang dimiliki oleh klien, sehingga layanan *streaming* terus berlangsung walaupun terjadi perubahan kecepatan jaringan di sisi pengguna. Pada aspek kualitas pengalaman *video streaming* diperoleh bahwa terjadi lima kali dengan level *buffer* = 0 dengan total 64 detik dari lima skenario ujicoba *video streaming*. Kondisi tersebut dicapai ketika ada penurunan kecepatan jaringan yang sangat ekstrem yaitu dari kecepatan jaringan yang cepat ke kecepatan jaringan yang sangat lambat.

Daftar Rujukan

[1] E. Kurniawan, A. Sani, and T. Pustaka, “Analisis Kualitas Real Time Video Streaming Terhadap Bandwidth Jaringan Yang Tersedia,” *Singuda ENSIKOM*, vol. 9, no. 2, pp. 92–96, 2014.

[2] APJII, “Laporan Survei Penetrasi & Profil Perilaku Pengguna Internet Indonesia.” 2018.

[3] S. Goring, A. Raake, and B. Feiten, “A framework for QoE analysis of encrypted video streams,” *2017 9th Int. Conf. Qual. Multimed. Exp. QoMEX 2017*, pp. 1–3, May 2017, doi: 10.1109/QoMEX.2017.7965640.

[4] S. S. Kallungal, “A Survey on Adaptive Video Streaming Techniques with Cloud,” *Int. J. Adv. Res. Comput. Eng. Technol.*, vol. 6, no. 3, pp. 350–352, 2017.

[5] Y. Bandung, Sean, L. B. Subekti, I. G. B. B. Nugraha, and K. Mutijarsa, “Design and Implementation of Video on Demand System Based on MPEG DASH,” in *2020 International Conference on Information Technology Systems and Innovation (ICITSI)*, Oct. 2020, pp. 318–322, doi: 10.1109/ICITSI50517.2020.9264973.

[6] Z. Duanmu, K. Zeng, K. Ma, A. Rehman, and Z. Wang, “A Quality-of-Experience Index for Streaming Video,” *IEEE J. Sel. Top. Signal Process.*, vol. 11, no. 1, pp. 154–166, Feb. 2017, doi: 10.1109/JSTSP.2016.2608329.

[7] L. De Cicco and S. Mascolo, “An adaptive video streaming control system: Modeling, validation, and performance evaluation,” *IEEE/ACM Trans. Netw.*, vol. 22, no. 2, pp. 526–539, 2014, doi: 10.1109/TNET.2013.2253797.

[8] N. Fimic, S. Tanackovic, J. Kovacevic, and M. Temerinac, “Implementation of multi-format adaptive streaming server,” *5th IEEE Int. Conf. Consum. Electron. - Berlin, ICCE-Berlin 2015*, pp. 218–220, 2016, doi: 10.1109/ICCE-Berlin.2015.7391239.

[9] S. Han, Y. Go, H. Noh, and H. Song, “Cooperative server-client http adaptive streaming system for live video streaming,” *Int. Conf. Inf. Netw.*, vol. 2019-Janua, pp. 176–180, 2019, doi: 10.1109/ICOIN.2019.8718151.

[10] R. Dubin, R. Shalala, A. Dvir, O. Pele, and O. Hadar, “A fair server adaptation algorithm for HTTP adaptive streaming using video complexity,” *Multimed. Tools Appl.*, vol. 78, no. 9, pp. 11203–11222, 2019, doi: 10.1007/s11042-018-6615-z.

[11] C. Mueller, S. Lederer, J. Poecher, and C. Timmerer, “Demo paper: Libdash - An open source software library for the MPEG-DASH standard,” *Electron. Proc. 2013 IEEE Int. Conf. Multimed. Expo Work. ICMEW 2013*, pp. 1–2, 2013, doi: 10.1109/ICMEW.2013.6618220.

[12] H. Azwar, “Jurnal Politeknik Caltex Riau Pengaruh Panjang Segmen Video pada Dynamic Adaptive Streaming over HTTP (DASH) terhadap Kualitas Pengiriman Video H.265,” 2015. Accessed: Nov. 25, 2020. [Online]. Available: http://jurnal.pcr.ac.id.

[13] ffmpeg.org, “FFmpeg.” https://ffmpeg.org/ (accessed Sep. 16, 2019).

[14] N. Jain, H. Shrivastava, and A. A. Moghe, “Production-ready environment for HLS Player using FFmpeg with automation on S3 Bucket using Ansible,” *2nd Int. Conf. Data, Eng. Appl. IDEA 2020*, pp. 26–29, 2020, doi: 10.1109/IDEA49133.2020.9170694.

[15] Y. Xu and S. Cao, “Design and implementation of a multi video transcoding queue based on MySQL and FFMPEG,” *Proc.*

DOI: https://doi.org/10.29207/resti.v5i3.2998

IEEE Int. Conf. Softw. Eng. Serv. Sci. ICSESS, vol. 2015-Novem, pp. 629–632, 2015, doi: 10.1109/ICSESS.2015.7339136.

[16] Golang.org, "Documentation - The Go Programming Language." https://golang.org/doc/.

[17] S. S. CHANG, *Go Web Programming*. Manning Publications, 2016.

[18] Noval Agung, "Golang Web Server - Dasar Pemrograman Golang," 2017. https://dasarpemrogramangolang.novalagung.com/A-web-server.html (accessed Apr. 08, 2021).

[19] A. A. Kristanto, Y. Harjoseputro, and J. E. Samodra, "Implementasi Golang dan New Simple Queue pada Sistem Sandbox Pihak Ketiga Berbasis REST API," *J. RESTI*, vol. 4, no. 4, pp. 745–750, 2020, Accessed: Apr. 09, 2021. [Online]. Available: http://www.jurnal.iaii.or.id/index.php/RESTI/article/view/2218/287.

[20] webmproject.org, "WebM VOD Baseline format - wiki," 2012. https://sites.google.com/a/webmproject.org/wiki/adaptive-streaming/webm-vod-baseline-format (accessed Dec. 07, 2019).

[21] R. Pike, "Go Concurrency Patterns," *Google*, 2012. https://talks.golang.org/2012/concurrency.slide#17 (accessed Apr. 14, 2021).

[22] dashif.org, "Dash JavaScript Player." http://reference.dashif.org/dash.js/latest/samples/dash-if-reference-player/index.html (accessed Dec. 07, 2019).

DOI: https://doi.org/10.29207/resti.v5i3.2998