DASH Streaming on Android for Live and On-Demand Video

Jabir Shabbir Karachiwala, Sunny Prakash, Prateek Kumar

A0123480N, A0123516N, A0123398Y

{jabirsk, sunny, prateekk}@comp.nus.edu.sg

17 November 2015

1 Abstract

In recent years, multimedia data has formed a significant share of the total internet traffic but it has certain intrinsic dynamics like user intolerance for buffering delays and latencies. However, the architecture of its common carrier (the internet) was not specifically designed to cater to such data as IP networks run on (essentially) best-effort packet-switched protocols.

In this project/report we provide an implementation of one such technique of serving live and on-demand multimedia data over the shared internet, namely MPEG-DASH or Dynamic Adaptive Streaming over HTTP. We implemented a system as Android applications tested on tablets and mobile phones, with a server application in PHP that uses ffmpeg for transcoding. The system has been tested end-to-end for live and on-demand delivery of video with an adaptive scheduling algorithm necessitated by varying network conditions across the pipeline. We also created the m3u8 playlist using transcode streaming which can be played on Safari/iOS browser.

2 Introduction and motivation

The internet has become a massive enabler of delivering content, information and data, since it was commercialized nearly two decades ago. Since then it has seen a steady increase in the number of users and evolving types of content it has carried, leading to a virtuous cycle of investment an e-commerce which has driven its growth. The impact of video streaming has been such that it has prompted the net neutrality debate due to the preponderance of traffic from content providers such as Netflix and Youtube [5].

In recent years, there has also been a paradigm shift towards carrying multimedia data, such as audio/video which has been facilitated by the arrival of CDNs (Content Delivery Networks) which can cache/stream such data over the common carrier (the internet). Such data can either comprise live audio/video streamed over virtual sessions, similar to Skype or desktop sharing via VNC/Teamviewer or it can be the distribution of stored video streamed on-demand without it being broadcast at the time of delivery.

In this report we discuss the various existing techniques used to implement such features along with the existing methodologies that have already been commercialized and are available today.

3 Existing Streaming Techniques

There are already several techniques to stream audio/video multimedia data with varying complexity and performance characteristics that have already been adopted. The RTP/RTCP/RTSP streaming protocol suites are a common stack used in the application layer to stream audio and video[1].

RTP or (Real Time Transport Protocol) allows for packet transmission with application-level framing using UDP, alongside usual TCP traffic. It can choose to use a milder additive-increase-multicative-decrease backoff algorithm so as to dynamically adapt its sending rate while not adversely affecting the TCP traffic alongside it to prevent starving such TCP traffic. It works alongside supporting protocols like RTCP which sends aggregated data reports from Sender to Receiver and Receiver to Receiver. These allow RTP to calculate its packet loss, inter-arrival jitter, round-trip time etc. to calculate network characteristics and to generally adapt its sending rate. The suite also includes RTSP which is used to send VCR-like playback commands from receiver to the sender, like play, stop, repeat etc.

WebRTC[2] is also a technique that is used in browsers as it intends to provide real-time capabilities via a common platform (to avoid endpoint plugins) in the browser which uses JavaScript APIs for sending/receiving the multimedia pipeline. The networking protocol in between these sessions can still be RTP/RTCP/RTSP.

Existing solutions however, have certain inherent disadvantages like the fact that they don't run over HTTP, and as a result are likely to be blocked by many firewalls. Since they're not using HTTP, the server itself has to be implemented separately unlike stable/simpler HTTP servers already available like Apache or IIS. It's also unlikely to use the standard caching infra structure made available by the fairly ubiquitous CDNs (like Akamai).

DASH[3] presents a fairly simple solution that tends to work around such problems. It achieves streaming of multimedia with dynamic bitrate adaptation, but entirely on HTTP (thus uses a regular web server, isn't blocked by firewalls and standard caching works with it). It also doesn't tend to interfere negatively with regular TCP web traffic alongside it (like RTP can potentially do because TCP's algorithm backs off much faster).

As a result, DASH has already become fairly popular with Netflix and Youtube (on HTML5 platforms) both using the protocol for streaming[4]. In this project we have implemented the DASH protocol for live streaming. The idea is to record video using an Android application which also segments and uploads those streamlets (each of 3 seconds duration) on to the server, using a standard HTTP POST command. The player retrieves these streamlets from the MPD file generated by the server, parses and downloads the segments, and finally schedules them in the appropriate size/quality as per its assessment of the real time network conditions.

4 Implementation

Our project was implemented on the Android platform (using Java) consisting of the Recorder and Player applications (running on separate Android devices like tablets/smartphones), with PHP providing the backend server functionality hosted at pilatus.d1.comp.nus.edu.sg/~team06 as below

4.1 Recorder

This application allows the user to record video (with audio) with the device's camera (and microphone) which is periodically uploaded in real-time to the server via an HTTP post command. This uploaded segment should be a 3s long mp4 file which will be transcoded by the server application.

We tried the two Camera classes available in Android to record the video:

1. Camera Intent – This takes a file location as the input, and will dump the entire recorded video/audio content after recording is stopped.

2. MediaRecorder – This class also dumps the recorded video content to a file (location taken as input), but it provides more options for setting the capture bitrate, resolution, encoder, output format. This was necessary for the project specification and so this class was used in our implementation.

The challenge with recording live video/audio with Android is that the recorded file is made available only ***after*** the recording is stopped by the user. This is because the MediaRecorder directly provides us the *encoded* MP4 file so we do not have the raw uncompressed frames available at the level of this class. One possible solution is to go a level deeper in the stack and using MediaCodec which allows us to grab the video and audio streams separately. The other solutions and the one we adopted are described below.

1. Restart MediaRecorder periodically – MediaRecorder is modeled as a state machine in Android and it has to be initialized before use. This process begins by taking a lock on the Camera and then setting the various video parameters, before connecting itself to the SurfaceView which renders the frames on screen, and then begins recording. It's possible to stop and restart the MediaRecorder object but this process takes around ~300ms because the initialization routine is heavy[6]. We have implemented this option and though it does lose a few odd frames at the streamlet boundary, we found it to be a tolerable penalty on a reasonably fast Android phone.

2. Tricking MediaRecorder using a LocalSocket – This workaround creates a LocalSocket and passes its descriptor to the MediaRecorder class. So instead of writing to a file, MediaRecorder keeps dumping the recording bitstream to a LocalSocket pipe. At the other end, its possible to have another socket accept() this connection and issue read() calls to drain this socket periodically and forward this datastream to the player. In our case we periodically drain this localsocket and dump this into a temporary file on the Recorder's file system, and then upload this to the server via an HTTP POST[7].

This method has been partly demonstrated by commercial applications on the Google Play Store (like Spydroid)[8], which uses LocalSocket instead of a file descriptor, but at the player's end it merely needs to read the bitstream and play it out. The challenge in using this for DASH is that the Recorder needs to drain the LocalSocket and have an MP4 file which in turn needs to be segmented and uploaded via HTTP. However, this trick does not give us a playable MP4 file when we periodically drain the LocalSocket because the socket is not seekable, and MediaRecorder computes/writes the header only at the end of the recording, but since it cannot be written for each streamlet (of 3 seconds), none of them are playable directly. We tried many ways to resolve this issue –

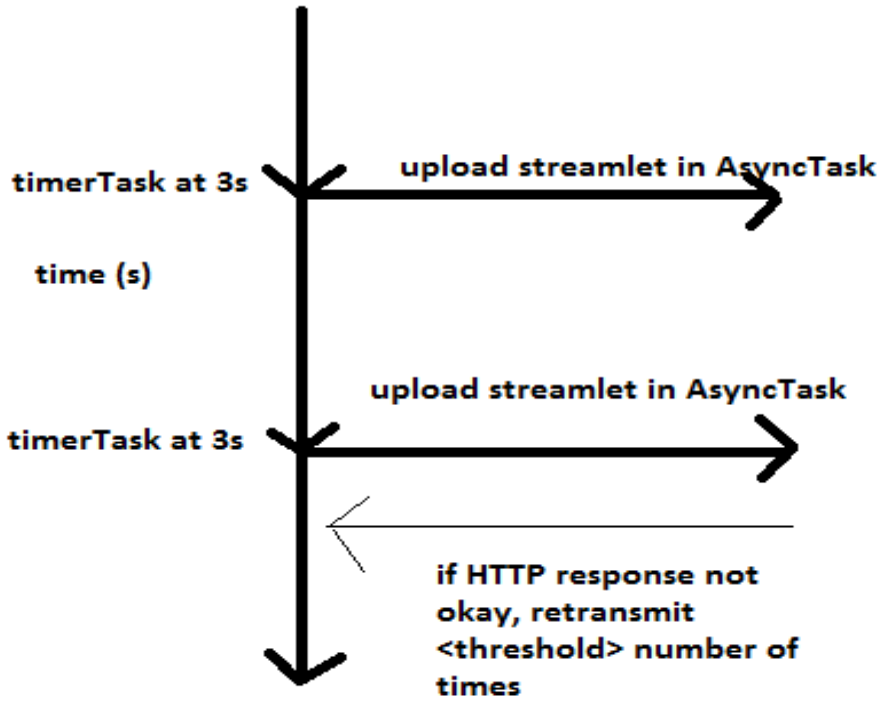
(a) Untrunc utlity[9] – This open source utility repairs a broken video, given a working mp4 file, but this was not able to restore our streamlets.

(b) ffmpeg on Server - /usr/bin/local/ffmpeg binary is able to recover this broken streamlet. In our project we upload this 3s-worth streamlet onto the server and execute the following command to recover it:

```
/usr/local/bin/ffmpeg -i input.mp4 -codec:v libx264 -profile:v high -preset slow -b:v 500k -maxrate 500k -bufsize 1000k -vf scale=-1:720 -threads 0 -codec:a libfdk_aac -b:a 128k –y output.mp4;
```

(c) ffmpeg on Client - There are already ffmpeg libraries[10] compiled for ARM available on the internet. We took one such binary and passed the following command to it with the broken streamlet on the client itself. Unfortunately this recovery on the ARM processor takes far too long (several minutes) for this to be feasible so we were forced to abandon it.

As a result of our tests of these options, we have implemented two Recorders, one which periodically restarts MediaRecorder and uploads the output streamlet, and the other which periodically drains the LocalSocket into a broken mp4 file which is uploaded onto the server (as is) and recovered by the ffmped binary on the powerful server machine's processor.



The high-level description of the Recorder is as follows – in both implementations (above) we use a TimerTask() which interrupts our main thread every 3 (or any constant) duration of seconds. We then have a streamlet available for upload (either as is or by draining the LocalSocket for the same duration of seconds), which is then uploaded via an asynchronous task, which automatically schedules itself. We also re-attempt re-transmission for each streamlet if the HTTP POST did not respond with an "OK" response status, once when the network is again available (but do not repeat this upon).

4.2 Server/Transcoder

This part of the application comprises of a series of PHP scripts which are called by the client when it uploads the streamlets onto the server. The server scripts are self contained and carries out the transcoding and updation of the MPD file. Upon the very first upload, the server script clears the MPD file of segments from the last recording, and then calls convert.sh (provided on the server already) or a custom ffmpeg command to transcode the streamlet into 3 different quality levels (by downrating the resolution and audiobitrate). This script gives us files of 3 different sizes (low ~200KB, medium ~450KB and high ~800KB) which are stored in the appropriate folder and listed in the MPD. We also created the m3u8 playlist using transcode streaming which can be played on Safari/iOS browser.

4.3 Player

This application allows to play live and recorded video on client. The three second segments are periodically fetched from the server and played on the device. The segment requests are sent by HTTP request to an Apache server. The availability and the size details of a particular segment is found out by parsing the mpd file. Here the mpd file is regularly updated as server receives the segments. On receiving the segments they are played by Android's Media Player class. The player is implemented as following.

1. Initial Buffering Stage-There we start the video only after we buffer 3 segments. The first ever segment requested would be the second last segment in the mpd file of the player. After that the consequent segments are downloaded. Before downloading the segment the availability of segment is always checked in MPD file. So the initial three segments are always requested for high quality to give good user experience. In ideal case we except the buffer length to be maintained, that is after playing every one segment, a new segment has to be downloaded.

Requesting the Segment before start-Here we request a segment after which we start playing the live video. Depending upon the segment size and network bandwidth we decide the time it requires to download, since we are not sure if it gets downloaded by the time we play, we finish playing the first segment. So three seconds before completing the download of the Segment before start we start playing the video. The segment before start is also requested in high quality as the player has not started playing the video and there is not risk of buffer under flow.

Playing Stage-The player is consecutively playing the segments, requesting it after every three seconds as it takes at least three seconds for the next segment to get appear on the server since the pipeline is maintained. The oldest segment in the buffer is removed and played. The player only stalls if the buffer gets exhausted, it then waits for 3 segments to get buffered only then it resumes the play.

Few constraints we took care of include the following cases:

1. Segment Out Of Order-If the segment arrives out of order it was to be rearranged in the player list .In this scenario we check if the segment which arrived has appeared before the last segment played,if so it is discarded else it is rearranged in its exact order in the queue.

2. Buffer Exhausted-If the buffer is exhausted ,the player stalls until 3 segments are buffered.The requests are continued as usual ,and the ones whose download deadlines and play deadlines are missed are requested for low quality and the rest are requested as usual.If a segment arrives late missing its deadline and buffer is not full it is still accepted provided its number is after the latest segment played. During this scenario if the segments misses deadline are counted once the buffer is full and then then the deadlines if future segments are increased since these segments are considered even after their deadline and play back of other segments are delayed.

3. Checking the Segment quality to be Requested-We determine the segment quality to be played considering certain constraints.We basically check the download deadline of the segment to be requested by checking the time it requires to download using various qualities and the highest one which is able to meet the deadline is requested.Another constraint checked here is that if it has already missed the download deadline then we check the playing deadline considering the average network bandwidth and size of the segment.

4. Requesting Fail Segments -The segment is will be considered as fail if if HTTP request to its url is fail or the if the segment entry is not found in xml or XML download is failed.These segments are sent for retransmission and based upon their deadlines their qualities are decided.The failed requests are then re requested with delay of every 200 milliseconds,if it is successful then we add them to video playlist,

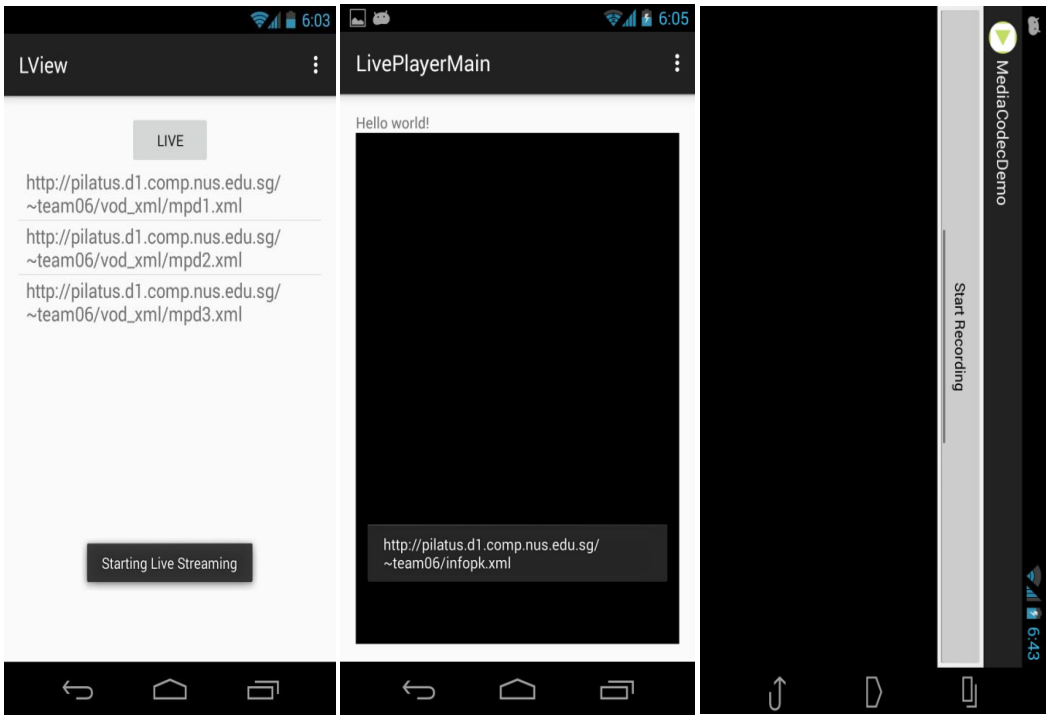
else they are again pushed in failed queue.If the deadline of the segment is missed and it is failed,then it is simply discarded.

We maintain separate queues for failed requests and the pending requests.

5 Results and Comparisons

We have tested our project end-to-end and can share the results as follows. It is feasible to implement a live streaming DASH player which can also fetch video on-demand. There are some unavoidable glitches which occur while converting a 3s-worth streamlet of live stream, into a file which is uploaded as-is onto the server, and recovering this file using ffmpeg results in certain distortions in some of the frames.

Screenshots of the Player and Recorder demonstrator applications:



6 Conclusion(s)

This method has streaming of live/on-demand video is feasible though it is more challenging for live multimedia data, as the inter-conversion between live encoded streams to persistent self-contained data on file, and back to decoded play out stream, has some intrinsic drawbacks and frame losses. Our implementation showcases a DASH

implementation with workarounds to resolve some of these challenges and works as per expectation.

7 References

[1] https://en.wikipedia.org/wiki/Real-time_Transport_Protocol
[2] https://en.wikipedia.org/wiki/WebRTC
[3] https://en.wikipedia.org/wiki/
Dynamic_Adaptive_Streaming_over_HTTP
[4] http://www.dash-player.com/blog/2015/02/the-status-of-mpeg-dash-today-and-why-youtube-and-netflix-use-it-in-html5/
[5] http://www.cnet.com/news/netflix-youtube-gobble-up-half-of-internet-traffic/
[6] http://developer.android.com/reference/
android/media/MediaRecorder.html
[7] https://foxdogstudios.com/peepers
[8] https://github.com/fyhertz/spydroid-ipcamera
[9] https://github.com/ponchio/untrunc
[10] http://hiteshsondhi88.github.io/ffmpeg-android-java/

Others – from CS5248 lecture slides.

[END]