International Journal of Computing and Digital Systems
ISSN (2210-142X)
Int. J. Com. Dig. Sys. 9, No.6 (Nov-2020)

http://dx.doi.org/10.12785/ijcds/090603

Applying Dynamic HTTP Adaptive Streaming over Unmanned Aerial Vehicle Networks

Heejae Han[1], Seokgyu Lee[2] and Dongho Kim[3]

[1] Department of Computer and Information Technology, Purdue University, USA
[2] Department of Computer Science and Engineering, Dongguk University, Seoul, Korea
[3] Dongguk Institute of Convergence Education, Dongguk University, Seoul, Korea

Received 25 Nov. 2019, Revised 10 Mar. 2020, Accepted 1 Aug. 2020, Published 1 Nov. 2020

Abstract: With the growing demand for UAVs (Unmanned Aerial Vehicles) such as drones, technology for streaming video using drones is also improving. However, networks for drones are not as stable as the topology continuously changes. To provide optimal QoE (Quality of Experience) in these networks, DASH (Dynamic Adaptive Streaming over HTTP) can be applied to drones. In DASH, the source video is segmented into short duration chunks of 2–10 seconds, each of which is encoded at several different bitrate levels and resolutions. This paper provides prototypes for applying DASH to drones. The prototype consists of a Raspberry Pi, a video processing server, web servers, a dash.js, and Exoplayer-based client.

Keywords: DASH (Dynamic Adaptive Streaming over HTTP); UAV (Unmanned Aerial Vehicle)

1. Introduction

UAV, also commonly known as a drone, is an unmanned aircraft which had been first developed and utilized during World War Ⅰ for military purpose. Owing to technological advances, both hardware and on-board computing system for UAVs have made great strides. In recent years, UAVs are not only being used for military purpose, but they are vigorously being utilized for public and civil applications such as weather monitoring [1], surveillance [2, 3], search and rescue [4], and live-event broadcasts. Accordingly, there exists a necessity for stable video streaming and robust communication. However, UAVs keep moving and network topology of drones changes every second. Therefore, there exists a drawback that the drone's communication bitrate is unstable, and this may cause problem such as delay or disconnection when clients get live video streaming from drones.

With the growth of companies such as Youtube and Netflix, multimedia delivery has become a major source of Internet traffic. Cisco VNI predicts IP video traffic to be above 80% of all Internet traffic by 2020 [5]. In the early stage, a stateful protocol such as Real-Time Streaming Protocol (RSTP) over User Datagram Protocol (UDP) or Transmission Control Protocol (TCP) had been used for streaming [6]. It works in a way that once a connection between a streaming server and a client is set up, the server keeps track of the client's state until the client terminates the connection. Hence, communication between the server and the client occurs frequently during a single session. However, these days, HTTP has become a main protocol for multimedia delivery thanks to its advantageous properties. In contrast with RSTP, HTTP is stateless protocol. If a client requests some data, the server responds to that specific request and the connection between them is terminated. Above all, HTTP-based delivery can utilize the existing infrastructure of internet.

DASH (Dynamic Adaptive Streaming over HTTP) is an adaptive bitrate streaming technology based on HTTP which enhances clients' Quality of Experience (QoE) [6, 7]. In this paper, we implemented a prototype for applying DASH to a single drone [8]. With Raspberry Pi and its camera module Pi Camera, we recorded a video and sent it to a video processing server. On the server, using FFmpeg and MP4Box libraries, we converted videos into several bitrate versions and DASH'ed these files. These files were uploaded on a web server and could be accessed through its URL by clients. Clients can stream a live video on either a dash.js-based web video player or an ExoPlayer-based Android application.

2. Related Work

As UAV technology gets advanced, various researches utilizing UAVs have been actively carried out. Among many different areas of research, aerial monitoring using video streaming is one of the areas which would likely to be applied to a variety of real world matters, from traditional surveillance tasks, such

E-mail: han607@purdue.edu, dltjrrb0729@naver.com, dongho.kim@dgu.edu

as natural disaster prevention, accident and terrorism prevention, or security of strategic national facilities or military sites, to emerging use cases, such as live-event broadcasts [9, 10]. There are many different approaches to improve video based aerial monitoring systems, and in this section we introduce a couple of approaches which have been background of our research.

UAV video streaming systems could be improved with a focus on different perspectives - improvement could be done on entire system perspective, or it could be done by enhancing a single UAV video streaming performance. Most of the recent studies regarding UAV video streaming have been done on the entire system side view, whereas not mush research has been conducted on the single UAV performance side view. Thus, in this paper, we propose a practical approach to improve a single UAV video streaming performance by demonstrating detailed implementation of the system so that it could be applied to larger UAV video streaming systems consist of multiple UAVs and control stations or vantage points.

Qazi *et al.* [11] has proposed a UAV based video surveillance system consists of multiple UAVs and vantage points, such as outdoor macro cells and indoor femto cells, by utilizing existing 4G LTE wireless network cellular infrastructure. They have proposed closed-circuit monitoring framework for streaming real time video using the legacy 4G LTE wireless network and investigated the performance of the framework by metrics of throughputs, loss rates and delay. As for another view, Scherer *et al.* [12] has focused on implementing an autonomous UAV system for search and rescue tasks. The proposed system is implemented in the Robot Operating System (ROS) and it provides real-time video streaming from a single UAV to multiple base stations using a wireless network infrastructure. Overall, many researchers have focused on implementing and developing entire video based UAV systems.

However, although many researchers have conducted investigations on whole UAV networks consists of multiple UAVs and Ground Control Stations (GCS) [13], not much work has been done on improving quality of a single UAV streaming itself. Therefore, we have mainly focused on enhancing quality of video streaming operated by a single UAV. Wang *et al.* [14] have made a first step toward adaptive video streaming algorithm that could be applied to UAV video streaming. They have used two factors - content based compression and video rate adaptation based on location sensors and client buffer status - to leverage their system performance.

Different standard bodies have been making efforts to organize and standardize video streaming technology as well. Among several different video streaming deployment methods, such as Microsoft Smooth Streaming [15], Adobe HTTP Dynamic Streaming [16], and Apple's HTTP Live Streaming [17], we have decided to apply DASH, specifically MPEG-DASH, to provide enhanced QoE to clients. DASH is an adaptive streaming technology based on HTTP, which is specified by Moving Picture Experts Group (MPEG) and ratified as a standard by ISO (International Organization of Standardization) [13]. Various standards bodies have participated in developing DASH and it has become a primary component of media delivery. Although there has been a variety of different deployments for media delivery, HTTP based media delivery due to advantageous properties of HTTP.

Conventional HTTP-based media delivery architecture, including DASH architecture, consists of a media preparation part, HTTP servers, and clients. In the media preparation part, a video source is segmented into several chunks and they are encoded applying several different options. Then, the chunks are hosted on a web server along with a Media Preparation Description (MPD) file. The MPD file is a metadata file which contains descriptive information about the chunk such as a content location, segment length, encodings, resolution, and bandwidths range of the chunk. Referring to this MPD information, clients request for data using HTTP GET methods. DASH works by cutting each video into 2-10 second chunks and converting them into several different bitrate versions. Thus, HTTP servers involved in DASH contains several different bitrate versions of chunks and clients can get proper bitrate video files depending on their network conditions and resources. The proper bitrate is chosen by adaptive bitrate algorithm (ABR) such as buffer-based ABR, throughput-based ABR or power-based ABR [6, 13].

3. System Design

3.1 System Overview

In the media preparation stage, Raspberry Pi records a video of which length is less than 10 seconds with its camera module, Pi Camera. The video recorded by Pi Camera is then sent to the video processing server over File Transfer Protocol (FTP). At this point, a format of the video is H.264 instead of mp4 because Raspberry Pi records video in a raw format. On the video processing server, a H.264 video file received from Pi-end is converted into mp4 file using MP4Box library. Then, the mp4 video file is encoded into several different quality videos using FFmpeg library. In this implementation, we used 360p, 480p and 720p. These three different quality videos containing the same contents will be DASH'ed using MP4Box library. As an output of DASH, three kinds of files – m4s, mp4, and MPD are obtained. A segment is represented as one m4s file. An mp4 file contains first 1000 bytes of the original video so that clients can get information about the video before m4s files. An MPD file contains descriptive information such as the content location, segment length, encodings, resolution, and bandwidths range of each m4s files.

When a client gets access to the streaming service, it will request an MPD file from a server first. Then, it requests a sequence of m4s files from the server based on the MPD file. These m4s files are uploaded on Apache web server so that clients can access files through the server's URL. We have developed two different kinds of video players on the client-side – one is a dash.js-based web video player and the other is an ExoPlayer-based Android application. Both two heterogeneous players can access to MPD files through the server's URL. The entire system flow is shown in Figure 1.

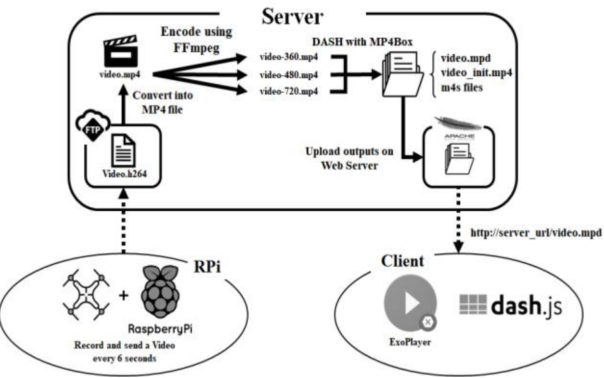

Figure. 1. Flow Chart of Prototype.

3.2 System Implementation

3.2.1 Video Recording on UAV

From this section, we cover the specific implementation of Hardware which is Raspberry Pi. The reason why we utilized Raspberry Pi for the prototype is that we can install a Linux-based operating system on it. Besides, since Raspberry Pi has a comprehensive camera module, which is Pi Camera, we don't need to implement networking parts such as Wi-Fi and camera recording part. Moreover, we can use Python programming language which has a variety of libraries including Pi Camera, FTP, and multiprogramming. Lastly, its size and weight are also suitable to be loaded on drones and we can supply power easily by using portable power batteries.

In this prototype, Raspberry Pi repeats recording a fixed length of video and sending it to a video processing server. Since recording a video and sending it should occur at the same time, we have implemented multiprocessing. As Raspberry Pi completes recording a video, it forks another process which sends a video file to an FTP server and the existing process begins recording the next video. With this multiprocessing process, we could record the videos without any missing points.

3.2.2 Servers

The implemented server is an FTP (File Transfer Protocol) server. First of all, it plays the role of receiving the H.264 file from Raspberry Pi. A reason why the server is implemented as an FTP server is that it is the fastest protocol for transferring files between Raspberry Pi and the server over TCP.

World Wide Web (WWW) accessed through the HTTP protocol has an advantage of being able to easily use general characters, pictures, music or video contents, but it has a fatal weakness in sending a large number of files and the file control is troublesome. Therefore, it is more advantageous to use FTP, which is a file transfer service, when a large number of files are constantly led through a network. Since FTP is a protocol designed solely for sending and receiving files over the Internet, the operation method is very simple and intuitive. More than anything, FTP's biggest advantage is that it can send and receive at a faster speed than HTTP.

The operation principle of FTP is relatively simple. Two connections are created between the server and the client, one for sending and receiving signals to control the data transmission (network 21 port) and the another for real data (a file) transmission (network 20 port). Network port refers to the path through which data travels over a network. Figure 2 shows a data transfer process between a Raspberry Pi and the FTP server implemented in this project.

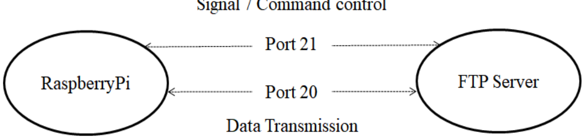

Figure. 2. Connective Operations of FTP.

FFmpeg, an open-source project under the license of GNU General Public License, which is aimed at decoding and encoding all the video, music and photo formats and being developed under the leadership of Michael Niedermayer, is a computer program that records and converts various types of digital audio streams and video streams. FFmpeg works by directly entering commands and consists of a variety of free software and open-source libraries. The H.264 file received by the FTP server is first encoded into MP4 (720p) using FFmpeg. After that, it is encoded into three resolutions of MP4 (360p, 480p, 720p). This is to provide each resolution according to the network environment. As the network condition gets improved, the original resolution of 720p could be seen.

However, client sides (Web, Android) do not receive MP4 files directly. They receive m4s files instead of the MP4 files. For the preparation of streaming contents, we used MP4Box. MP4Box is an opensource software that enables video file conversion and file hinting for video streaming. In this project, we utilized MP4Box to divide each MP4 files of 3 different resolutions into 2 seconds m4s files. In addition, init.mp4 file and MPD file are generated by MP4Box.

The init.mp4 file is an initialization segment necessary to start streaming a video and the MPD file is a metadata file containing information about m4s files which the clients can refer to.

All of the previous files processed by FTP server – init.mp4, MPD, m4s files – are uploaded to the Apache Web server so that we could implement DASH. The web server loads Web (dash.js) and Android (ExoPlayer). Consequently, clients can view the proper quality of contents seamlessly depending on their network conditions by referring to the MPD file.

3.2.3 Clients

Here we cover how we have implemented a client-side of the system. We have developed two different types of clients to make heterogeneous access to single video content available. Sample codes for implementation are included hereafter.

A) Web Client: Dash.js

At this point, only Microsoft Edge, which operates on Windows 10, supports DASH streaming natively. Thus, implementing DASH on other browsers and operating systems is available through Media Source Extensions (MSE). For instance, although MPEG-DASH is not directly supported in HTML5, dash.js provides JavaScript implementations of MPEG-DASH. It facilitates developing MPEG-DASH in web browsers by using the HTML5 MSE.

A-1) ***How to implement a web client****: To create a simple web browser that displays a video player with expected functions such as play, pause, rewind, etc., you need to:*

1. Create an HTML page
2. Add the video tag
3. Add the dash.js player
4. Initialize the player
5. Add some CSS style
6. View the results in an MSE browser

Initiating a video player can be completed in just a handful of lines of JavaScript code. Using dash.js, it is that simple to embed MPEG-DASH video in your browser-based applications.

The first step of implementing a browser-based video streamer is to create a standard HTML page which contains a video element and save this file as basicPlayer.html. Following is an example code:

```
<!DOCTYPE html>
 <html>
   <head>
    <title>Adaptive Streaming in
HTML5</title></head>
   <body>
    <h1>Adaptive Streaming with HTML5</h1>
    <video id="videoplayer"
controls></video>
   </body>
 </html>
```

To add dash.js reference implementation to your application, you need to grab the dash.all.js file from the 1.0 release of dash.js project. This file should be saved in the JavaScript folder of your application. It is a convenience file that pulls all the necessary dash.js code together into a single file. If you take a look into the dash.js repository, you can find each files, test codes and much more. However, if all you want to do is using dash.js only, the dash.all.js file is all you need. To add the dash.js player to your applications, add a script tag to the head section of basicPlayer.html as follow:

```
<!-- DASH-AVC/265 reference implementation -
->
    < script src="js/dash.all.js"></script>
```

Next, create a function to initialize the player when the page loads. Add the following script after the line in which you load dash.all.js:

```
<!-- DASH-AVC/265 reference implementation -
->
< script src="js/dash.all.js"></script>
<script>
    // setup the video element and attach it
to the Dash player
    function setupVideo() {
      var url =
"http://wams.edgesuite.net/media/
MPTExpressionData02/BigBuckBunny_1080p24_IYU
V_2ch.ism/
manifest(format=mpd-time-csf)";
      var context = new
Dash.di.DashContext();
      var player = new MediaPlayer(context);
      player.startup();
      player.attachView(document.querySelect
or("#videoplayer"));
      player.attachSource(url);
    }
</script>
```

The setup video() function above first creates a DashContext by using Dash.di.DashContext(). This is used to configure the application for a specific runtime environment. Next, instantiate a primary class of the dash.js framework, MediaPlayer. This class contains

core methods of a video player such as play and pause and manages video elements and interpretations of MPD files. The startup() function of the MediaPlayer class is called to ensure that the player is ready to play a video. It ensures that all the necessary classes have been loaded. Once the player is ready, you can attach a video element to the player using the attachView() function. The startup() function enables the MediaPlayer to inject video chunks into the element and also control playback as necessary. Passing URL of the MPD file to the MediaPlayer allows MediaPlayer to recognize the video expected to play next. Lastly, the setupVideo() function mentioned above needs to be executed once the page has fully loaded. It can be achieved by using the onload event of the body element. Change your element to:

```
<body onload="setupVideo()">
```

Finally, set the size of video elements using CSS. In an adaptive streaming environment, this is especially important because the size of the video being played is changeable as playback adapts to changing network conditions. In this simple demo, we simply forced the video element to be 80% of the available browser window by adding the following CSS to the head section of the page:

```
<style>
    video {
      width: 80%;
      height: 80%;
    }
</style>
```

To play a video, point your browser at the basicPlayback.html file and click play on the video player displayed.

2) Problems and solutions: The first problem we had faced in a web client is the Cache problem. Since a video is encoded further on the server-side, MPD file should be updated periodically on the client. Otherwise, the same image repeats itself. To resolve this problem, the HTTP specification allows the server to return Cache-Control directives that control how, and for how long, the browser and other intermediate caches can cache the individual response. And here we use no-cache. "no-cache" indicates that the returned response can't be used to satisfy a subsequent request to the same URL without first checking with the server if the response has changed. As a result, if a proper validation token (ETag) is present, no-cache incurs a roundtrip to validate the cached response but can eliminate the download if the resource has not changed.

The second problem we had encountered is Cross-Origin Resource Sharing (CORS) problem. CORS is a mechanism that uses additional HTTP headers to tell a browser to let a web application running at one origin (domain) have permission to access selected resources from a server at a different origin. A web application makes a cross-origin HTTP request when it requests a resource that has a different origin (domain, protocol, and port) than its origin. Initially, the domain of the server providing the client page was different from the domain of the server providing the MPD file, so there was a problem importing the MPD file from the server. This is because browsers restrict cross-origin HTTP requests initiated from within scripts for security reasons. Allowing CORS has solved the problem. We have used Chrome's extension to make the CORS request.

B) Android Client: ExoPlayer

ExoPlayer is an application-level media player for Android and an open-source project which is provided by Google and distributed separately from the Android SDK. ExoPlayer supports features not currently supported by Android’s MediaPlayer API, including DASH and SmoothStreaming adaptive playbacks. ExoPlayer’s standard audio and video components are built on Android’s MediaCodec API, which was released in Android 4.1(API level 16). ExoPlayer is easy to customize and extend and is a library and people can easily take advantage of new features as they become available by updating their app.

B-1) ***How to implement a web client****: To create an Android application that displays a video player with expected functions such as play, pause, rewind, etc., you need to:*

1. Create an Android application
2. Add ExoPlayer
3. Initialize the player
4. Play the video which comes with DASH protocol

As ExoPlayer can be easily customized, we changed a lot of features after including ExoPlayer library to make a smooth DASH player.

We include Exoplayer to depend on the library modules that need. Adding a dependency to the full ExoPlayer library is equivalent to adding dependencies on all of the library modules individually. ‘exoplayer-dash:2.8.0’ supports DASH contents.

```
dependencies {
   implementation 'com.android.support:appcompat-v7:27.1.1'
   implementation 'com.android.support:support-v4:27.1.1'
   implementation 'com.google.android.exoplayer:exoplayer-core:2.8.0'
   implementation 'com.google.android.exoplayer:exoplayer-dash:2.8.0'
   implementation 'com.google.android.exoplayer:exoplayer-ui:2.8.0'
}
```

These lines should be included in 'build-gradle'.

Next, we put URL we got from the server side in 'MediaSource source' to play the video server-side sends.

```
MediaSource source =
buildMediaSource(Uri.parse("http://210.94.185.47/test/test.mpd
"));
player.prepare(source, true, false);
player.setPlayWhenReady(playWhenReady);
```

We put 'initializePlayer' in 'onStart()' to make the player play videos when the application is turned on.

```
public void onStart() {
 super.onStart();
 if (Util.SDK_INT > 23) {
  initializePlayer();
 }
}
```

Also, we made it keep receiving the video segments from the server by put 'initializePlayer' in 'onStop()'. It made the player keep getting video segments and we were able to make it live-streaming.

```
public void onStop() {
 super.onStop();
 if (Util.SDK_INT > 23) {
   initializePlayer();
 }
}
```

B-2) ***Problems and Solutions****: We came across some problems in the early stage and the biggest problem was that the player keeps playing the same video segments even though we tried to make a live-streaming DASH player. We figured out what the problem was and it turned out that the problem was that the player didn't get fresh video chunk when the previous one ends. So we made the player get the new chunk by making the player get the same URL when one video segment ends.*

```
case Plyaer.STATE_ENDED
      stateString = "ExoPlayer.STATE_ENDED -";
       MediaSource mediaSource =
buildMediaSource(Uri.parse("MPD URI"));
       Player.prepare(mediaSource, restPosition: true, resetState:
false);
    break;
```

Another problem we encountered is a buffer problem. When we tried to play a video file which we got from the server-side, it had a problem which skipped some video segments or played the previous segments. We tried to figure out what the problem was and we found that this problem occurred because we didn't make a proper buffer for the player. So, we modified the code to make a buffer and when we played it later on, it worked well.

```
DefaultAllocator allocator
       = new DefaultAllocator(true,
C.DEFAULT_BUFFER_SEGMENT_SIZE);
LoadControl loadControl =
       new DefaultLoadControl(
       allocator,
       10000,
       30000,
       8000,
       5000,
             C.DEFAULT_BUFFER_SEGMENT_SIZE,
             true
       );
```

EXPERIMENT

To evaluate the performance of the prototype, we conducted an experiment with different video lengths. We could adjust two kinds of video lengths in the prototype. One is the length of the original video which is being taken on the Raspberry Pi and the other is the length of video chunks which are cut from the original video. In some conditions, it occurred that client-side player skips some video segments. Focusing on this problem, we tried to find the optimal length of video and chunk video size for smooth playback.

We conducted the experiment with three different conditions. In [α, β], we set α as the length of the original video, and β for the length of chunks. The three different conditions are shown in Table 1.

TABLE 1. CONDITIONS OF EACH EXPERIMENT

Experimen	α	β
A	20	4
B	6	2
C	4	2

We have tried to find optimal length of video and chunk video size for smooth playback by trial and error. At first, we tried recording a video every 2 seconds. However, when sending this video to the server, sometimes it took more than 2 seconds. This can result in stacking videos on Raspberry Pi. Therefore, we needed to lengthen the video recording time. Also, we had to consider a video encoding time on video processing server because the sum of video transmitting time and video encoding time should be shorter than video recording time. As you can see in Figure 3, there

was little miss of timing when we recorded videos every 6 seconds. The video transmission time took about 2.3 seconds and converting time took 3.5 seconds.

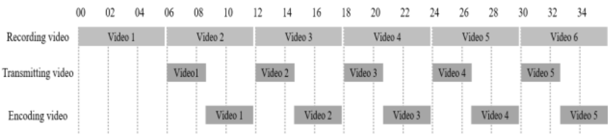

Figure. 3. Video Processing Timeline.

CONCLUSIONS

We have made an attempt to apply DASH to implement UAV video streaming system. One of the key points of DASH is selecting appropriate segment length to support smooth playback on client-side. Accordingly, we experimented 3 different pairs of video length and chunk length to find out an optimal chunk size for smooth playback. We have determined those lengths based on observations and client-end heuristics. Although network conditions between web server and clients were erratic, communication between FTP server and UAV and video processing time on FTP server had consistency. Thus, based on the consistency, we could figure out the optimal original video length and chunk length which support the smooth playback on client-side.

As future work, some alternate adaptive bitrate algorithms (ABRs) can be applied to improve client's QoE. In addition, although we have applied DASH to a single drone, we can develop more scalable deployment of network which involves a number of UAVs. Addressing more complex UAV Ad hoc Networks (UAANETs) remains as an issue.

ACKNOWLEDGMENT

This work was supported by the MSIT (Ministry of Science and ICT), Korea, under the National Program for Excellence in SW supervised by the IITP (Institute for Information & communications Technology Planning & Evaluation)"(2016-0-00017)

REFERENCES

[1] Wesley, M. D. (2010). Unmanned aerial vehicle systems for disaster relief: Tornado alley. In AIAA Infotech@ Aerospace Conference. AIAA, 2010-3506.

[2] Dieter, H., Werner, Z., Gunter, S., Peter, S. (2005). Monitoring of gas pipelines-a civil UAV application. Aircraft Engineering and Aerospace Technology, 77(5), 352–360.

[3] Zainab Z., Atiya U., Ekram K., Mohammed A. Q. (2016). Aerial surveillance system using UAV. In Thirteenth International Conference on Wireless and Optical Communications Networks (WOCN). IEEE.

[4] Jürgen, S., Saeed, Y., Samira, H., Evsen, Y., Torsten, A., Asif, K., Vladimir, V., Christian, B., Hermann, H., Bernhard, R. (2015). An Autonomous Multi-UAV System for Search and Rescue. In Proceedings of the First Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use. (pp. 33-38). ACM.

[5] Cisco. (2019). Cisco Visual Networking Index: Forecast and Trends, 2017–2022 White Paper. Retrieved from https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html

[6] Thomas S. (2011). Dynamic Adaptive Streaming over HTTP --: Standards and Design Principles. In Proceedings of the second annual conference on Multimedia systems. (pp. 133-144). ACM.

[7] Jonathan, K., Grenville, A., Philip, B. (2017). A Survey of Rate Adaptation Techniques for Dynamic Adaptive Streaming over HTTP. IEEE Communications Surveys & Tutorials, 19(3), 1842-1866.

[8] Abdelhak B., Ali C. B., Saad H., & Roger, Z. (2018). A distributed approach for bitrate selection in HTTP adaptive streaming. In Proceedings of the 26th International Conference on Multimedia. (pp. 573-581). ACM.

[9] A. Merwaday, A. and I. Guvenc, "UAV assisted heterogeneous networks for public safety communications", in Proceedings of IEEE Wireless Communications and Netwrking Conference Workshops, pp. 329-334, New Orleans, LA, March 2015

[10] Mike, Ohleger; Geoffrey G Xie and John H. Gibson, "Extending UAV Video Dissemination via Seamless Handover: A Proof of Concept Evaluation of the IEEE 802.21 Standard", in Proceedings of Hawaii International Conference on System Sciences, pp. 5106-5114, USA, January 2013

[11] Sameer Q., Ali S. S., Asim I. W. (2015). UAV based Real Time Video Surveillance Over 4G LTE. In Proceedings of the International Conference on Open Source Systems and Technologies. (pp. 141-145). IEEE.

[12] Jurgen S., Saeed Y., Samira H., Evsen Y., Torsten A., Asif K., Vladimir V., Christian B., Hermann H., Bernhard R. (2015). An Autonomous Multi-UAV System for Search and Rescue. In Proceedings of the First Workshop on Micro Aerial Networks, Systems, and Applications for Civilian Use. (pp. 33-38). ACM.

[13] Jean-Aimé M., Mohamed-Slim B. M., Nicolas L. (2017). Survey on UAANET Routing Protocols and Network Security Challenges. Ad Hoc & Sensor Wireless Networks, PKP Publishing Services Network. (hal-01465993).

[14] Xiaoli W., Aakanksha C., Mung C. (2016). SkyEyes: Adaptive Video Streaming from UAVs. In Proceedings of the 3rd Workshop on Hot Topics in Wireless. (pp. 2-6). ACM.

[15] Microsoft. (2019). Retrieved from

[16] https://www.microsoft.com/silverlight/smoothstreaming

[17] Adobe. (2016). HTTP Dynamic Streaming (HDS). Retrieved from https://www.adobe.com/devnet/hds.html

[18] Apple. (2016). HTTP Live Streaming (HLS). Retrieved from https://developer.apple.com/streaming

Heejae Han Department of Computer and Information Technology, Purdue University, USA … Received her B.S. degree in Industrial Systems Engineering from Dongguk University, Seoul, Korea in 2019 and pursuing her M.S. degree in Computer and Information Technology at Purdue University, West Lafayette, Indiana, USA. Her research interests include artificial intelligence, sensor systems, robotics and multi agent systems.

Seokgyu Lee Department of Computer Science and Engineering, Dongguk University, Seoul, Korea. Received his B.S. degree in Computer Science Engineering from Dongguk University, Seoul, Korea in 2019. He is now an employee of e-government team, D&T innovation department, LG CNS. His research interests include IoT systems and networks.

Dongho Kim Dongguk Institute of Convergence Education, Dongguk University, Seoul, Korea … Received his B.S. degree in Computer Engineering from Seoul National University, Korea in 1990 and his M.S. and Ph.D. degrees in Computer Science from University of Southern California, Los Angeles, California, U.S.A., in 1992 and 2002, respectively. He is now a professor at Dongguk Institute of Convergence Education, Dongguk University, Korea. His research interests include network security and IoT.