Implementing a System Architecture for Data and Multimedia Transmission in a Multi-UAV System

Borey Uk, David Konam, Clément Passot, Milan Erdelj, Enrico Natalizio

▶ **To cite this version:**

Borey Uk, David Konam, Clément Passot, Milan Erdelj, Enrico Natalizio. Implementing a System Architecture for Data and Multimedia Transmission in a Multi-UAV System. 16th International Conference on Wired/Wireless Internet Communications (IFIP WWIC 2018), Jun 2018, Boston, MA, United States. pp.246-257, 10.1007/978-3-030-02931-9_20 . hal-02269736

HAL Id: hal-02269736

https://inria.hal.science/hal-02269736

Submitted on 23 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Implementing a system architecture for data and multimedia transmission in a multi-UAV system

Borey Uk, David Konam, Clément Passot, Milan Erdelj, and Enrico Natalizio

Sorbonne Universités
Université de Technologie de Compiègne
UMR CNRS 7253 Heudiasyc, France
{borey.uk, david.konam, clement.passot}@etu.utc.fr
{milan.erdelj, enrico.natalizio}@hds.utc.fr

Abstract. The development of Unmanned Aerial Vehicles (UAV) along with the ubiquity of Internet of Things (IoT) enables the creation of systems that can provide real-time multimedia and data streaming. However, the high mobility of the UAVs introduces new constraints, like unstable network communications and security pitfalls. In this work, the experience of implementing a system architecture for data and multimedia transmission using a multi-UAV system is presented. The system aims at creating a bridge between UAVs and other types of devices, such as smartphones and sensors, while coping with the multiple fallbacks in an unstable communication environment.

1 Introduction

The development of UAVs and their applications in different domains opens new possibilities in natural disaster management [1, 2]. In UAV-assisted disaster management applications, UAVs not only report the affected area but also establishes and maintains a communication network between multiple types of actors, like smartphones or web clients.

This work describes the communication architecture for a system of systems composed of UAV, smartphones, and sensors to transmit telemetry and data streaming, which we proposed in the framework of the project IMATISSE (Inundation Monitoring and Alarm Technology In a System of SystEms). The main contributions of this work are the following:

1. We review the state of the art of the technologies for multimedia streaming in dynamic networks;
2. We identify a set of technologies that can be used within a framework composed of different kinds of mobile communication devices;
3. We propose a whole novel communication architecture for disaster management that includes UAVs, smartphones and sensors.

In the rest of the paper, multimedia streaming approaches are reviewed in Section 2, we present the data transmission architecture in Section 3 and its implementation in Section 4. Conclusions are drawn in Section 5.

2 Background on multimedia streaming

This section, first, surveys the existing protocols for video streaming. It ranges from protocols designed for Video on Demand to real-time low latency protocols. Then, it presents the encoding/decoding algorithms that play an important role in providing low latency and high quality multimedia transmission.

2.1 Network protocols

Streaming protocols - RTSP/RTMP Back in the 90s, videos served over HTTP needed to be fully downloaded before they can be played. The creation of *progressive download* - video can be played as soon as a fragment of video is downloaded - helped a bit with giving a sense of streaming. However, functionality was still limited. As an example, there was no look-ahead seeking control.

The RTSP (Real Time Streaming Protocol) stack was designed in the 90s as an answer to these issues, and is composed of the following protocols:

- RTP (real time transport protocol): transport layer, built on top of UDP;
- RTCP (real time control protocol): session layer, quality control;
- RTSP (real time streaming protocol): presentation layer, "network remote control".

This suite of protocols was the basis for the RTMP (Real Time Messaging Protocol), the leading protocol for multimedia streaming at the time. The main concept of RTSP and RTMP is to create a stateful connection between the server and the client. Thus, the protocol offers multimedia functionality to the client, like fast-forwarding or rewinding. Moreover, as the protocol suite has control over the transport, session and presentation layer, it performed better than HTTP at the time. Transfer rates were faster and bandwidth was saved, in comparison to HTTP Progressive Downloading. Latency was also fairly low, averaging delay in seconds. However, RTMP and, by extension, RTSP, had heavy restrictions regarding the client and server. Indeed, as RTMP is based on another protocol, it required the use of a special player and server, and the stateful connection implied increased network usage. This need of an additional infrastructure and lack of compatibility with HTTP was a burden for the clients and the servers. As Adobe Flash (the main technology mandatory for RTMP) is being phased out and is now unsupported by a rising numbers of device and software, the need for a replacement started to grow.

HTTP Streaming - HLS and DASH As described in [12], there was a need for a user-convenient video streaming protocol, which can be used without using any other software than a web browser. One of the first "new generation" HTTP-based streaming technology alternative to RTMP was HLS (HTTP Live Streaming), a protocol developed by Apple. It used the "Progressive Download" design, by breaking the stream into small files, letting the user play each file at a time. It also adapted the bit-rate according to the internet connection:

more than *Progressive Download*, HLS was *Adaptative Streaming*. As HLS was proprietary and designed by Apple, it was not widely supported by other devices or browsers.

DASH (Dynamic Adaptive Streaming over HTTP) is now the open-source standard protocol for HTTP Video Streaming. The main concept is the same than HLS, but differs in the sense that it is codec agnostic, open-source, and clearly defined by a international standard [4]. DASH is now a standard technology and is used by Netflix or YouTube, as described in [4].

In [5], the usage of DASH for low-latency communications is described. DASH was not designed as a low-latency solution, in fact it rather targets multimedia usages like video serving on YouTube. DASH has, on average, more latency than RTMP solutions, as described in [13], mainly due to the segmentation and downloading process. However, by tweaking the segment size and other parameters, the authors of [5] achieve a best-case 240ms lag on a local network. It is important to note here that DASH relies on HTTP/1.1, which has a lot of overhead for real-time communications.

HTTP/2, Websockets and WebRTC HTTP/2 is the successor of HTTP/1.1, defined by the IETF (Internet Engineering Task Force) in 2015. Since the 90s, Internet and its content has changed: from text and images, Internet is now mainly composed of multimedia content, like video and audio. HTTP/2 aims at removing the protocol overhead of HTTP/1.1 while reducing latency, lowering the number of connections and enabling data streaming [14]. Furthermore, HTTP/2 introduces *server push*, which means that a server can push data to the client. HTTP/2 seems to be the ideal transport protocol for DASH, which is currently implemented over HTTP/1.1. Indeed, the implementation of DASH over HTTP/2 is still a work in progress[1]. While we are waiting for HTTP/2 to become mainstream, there are other ways to have real-time communication in a web browser, like WebRTC or WebSockets.

WebRTC is a browser-based real-time protocol API for web browsers. It is still in a draft state but the main web browsers support it[2]. WebRTC provides peer-to-peer communication between two browsers and at a transport layer, it can transfer any type of data (sound, video, binary data, etc). However, WebRTC does not include signaling, therefore a user would still need a signaling server to coordinate data exchange between two browsers.[3] Furthermore, WebRTC also requires the use of "STUN" and "TURN" servers: the "STUN" server exposes a public IP for each of the peers whereas the "TURN" server is a cloud fallback server which is used if a peer-to-peer communication cannot be used. As a consequence, WebRTC can be quite complex to deploy and use. Still, an im-

[1] A draft is available here https://www.iso.org/obp/ui/#iso:std:iso-iec:23009:-6:dis:ed-1:v1:en

[2] Support for iOS browsers was added with iOS 11, while Google Chrome, Mozilla Firefox and (partially) Microsoft Edge supports WebRTC.

[3] http://io13webrtc.appspot.com/

plementation of DASH over WebRTC is described in [11]. In this paper, authors achieve a latency of 170ms, which is lower than described in [5].

The WebSocket protocol was standardized in 2011 by the IETF. Like WebRTC, it enables full-duplex communication between a web browser and a server. Even if the WebSocket protocol differs from the HTTP protocol, they are compatible. WebSocket is not inherently designed for multimedia communication, and may be less performing than WebRTC for video transmission. Nevertheless, WebSocket is supported by all the browsers, and is simpler to use than WebRTC. A solution for low-latency video streaming would be to use WebSockets to transfer raw video data and use the browser to decode it, which is the solution proposed in this paper.

2.2 Encoding and decoding

Each multimedia container format supports different video, audio formats and compression types. There are numerous video file formats, each with different features and benefits.

Encoding with FFmpeg. FFmpeg is a multimedia framework, able to decode, encode, transcode, multiplex and stream multimedia flows. It supports the most obscure ancient formats up to the cutting edge. It is also highly portable – FFmpeg compiles, runs under a wide variety of build environments, machine architectures, and configurations.

Decoding with JSMpeg. JSMpeg is a video player written in JavaScript, that consists of MPEG-TS demuxer, MPEG1 video and MP2 audio decoders, WebGL and Canvas2D renderers and WebAudio sound output. JSMpeg can load static videos via Ajax and allows low latency streaming via web sockets. It can work in any modern browser (Chrome, Firefox, Safari, Edge). JSMpeg can connect to a web socket server that sends out binary MPEG-TS data. When streaming, JSMpeg tries keeping latency as low as possible - it immediately decodes everything it has, ignoring video and audio timestamps altogether.

We need to keep in mind that MPEG1 is not as efficient as modern codecs. MPEG1 needs quite a bit of bandwidth for HD video (for example, 720p video quality begins to look acceptable at 2 Mbits/s throughput). Also, the higher the bitrate, the more work JavaScript has to do to decode it.

2.3 Image processing

In the context of natural disaster management, in addition to receive the video stream in real time, it could be useful to notify the users of a web application with information related to the detection of human beings hit by the disaster or the detection of the source of the disaster (fire for instance). For this reason, in the following subsections we will review some solution for image processing.

WebAssembly. WebAssembly (or wasm) is a portable and load-time-efficient format suitable for compilation to the web. It is currently being designed as an open standard by a W3C Community. It is efficient and fast because the wasm stack machine is designed to be encoded in a size-binary format. WebAssembly executes at native speed by taking advantage of common hardware capabilities available on a wide range of platforms.

OpenCV. OpenCV (Open Source Computer Vision) is a library of programming functions mainly aimed at real-time computer vision for all the operation related to image processing. The library is composed of around 3000 algorithms, which include a set of both classic and cutting edge computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects. The architecture proposed in this paper relies on this technology due to its richness in the algorithms it offers.

3 Data architecture

We will now focus on the data side of the solution we want to build. Other than video streaming, our system has to manage telemetry data and commands message, so we have to decompose our system into functional blocks.

3.1 Functional architecture

In the target system, 4 segments can be identified: UAVs, UAV server, web server, and clients (smartphones and web browsers), as in Figure 1.

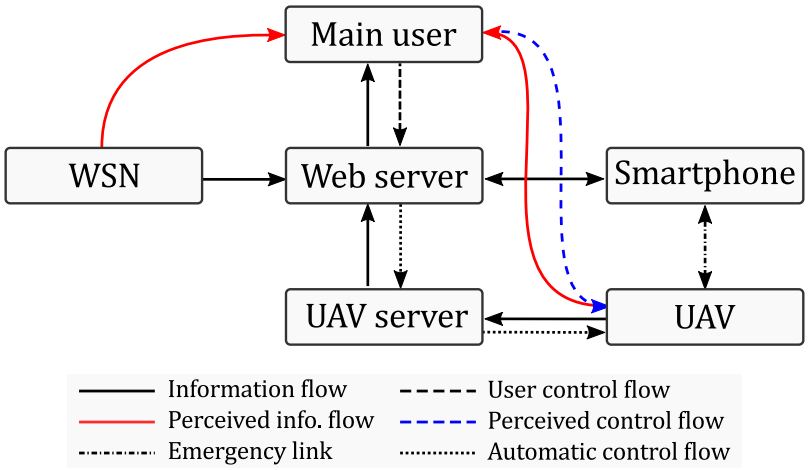

Fig. 1. Proposed architecture

The *UAVs* are gathered by fleets, where each fleet of UAV sends telemetry data, while each UAV sends its video. All the data sent by the UAVs is received by the *UAV server*, which is connected to the UAVs through a local wireless network. In return, the UAVs receive Mavlink commands from the UAV server. The UAV server centralizes all the data sent by the UAVs. Additionally, it also exposes each videostream for the *web clients* and also receives the command messages sent by the web server. The **Web server** is the main element of the architecture: it stores telemetry and stream processing data into a database, and also provides an API to the web clients. The core of the web server is a program written in Golang which is responsible for launching the different modules in several threads. The web server comprises the following modules:

- API: Responsible for exposing a RESTful API to the clients. Enables two-way communication between the clients and the UAVs. On one hand, the web application can query the database through the API module to retrieve data such as the last telemetry of a UAV or retrieve a snapshot of the video stream. On the other hand, clients can also send commands to the UAV fleet.
- Processing modules: This set of modules are responsible for processing data coming from the UAV fleet. Each type of data is assigned to a specific submodule:
  - `Streaming`: Manages the websocket stream video servers. Synchronizes the different video inputs (UAV streams) with the outputs (video players which are requesting a given stream).
  - `Screenshot`: Manages the reception of the snapshot resulting from the video stream and its analysis by OpenCV (use of face-detection algorithm). It also allows the recording of streaming information into the database (addresses where UAVs publish their streams, and the addresses where the web client can retrieve them).
  - `Telemetry`: Process telemetry data received and stores them in a database.

3.2 Dataflow and encoding

As described in [7], there are three ways of communicating between software blocks, using a database, services, or messages. In our architecture, we make use of these three ways :

- Communication through a service is used between the `API` module and the clients;
- Communication through a database is used between the `API` module and the Processing submodules;
- Every other communication uses messages with *ZeroMQ*.

Encoding protocols is also described in depth in [7]. In this part, we will explain which data encoding technologies we chose for each block, by describing their general usage and their use in our implementation.

JSON. *JSON* is the short for *JavaScript Object Notation*. It is a human-readable, text-based file format, which is independent of any programming language. A *JSON* object is composed of a set of key/value pairs: a key is a string while a value can be a string, a number, a boolean expression, an array containing a value or even another *JSON* object. As *JSON* is simple to comprehend, it boasts a wide range of compatibilities. Every modern programming language has a library implementing JSON encoding/decoding, which makes this format effortless to use. However, even if *JSON* is lighter than *XML*, it is still heavier than other binary formats, like *FlatBuffers*.

Protocol Buffers. *Protocol Buffers.* - or *protobuf* - is a binary encoding technology. Protocol Buffers functions with *protocol buffer message types*, which are language-agnostic files defining the messages that the user want to serialize. These *protocol buffer message types* are then used with a specific compiler, *protoc*, which generates an encoding/decoding library for the majority of modern languages - C++, Java, Go, Javascript, etc.
Protocol Buffers was designed by Google, and is tightly coupled with *gRPC*, a RPC-based framework also developped by Google and based on HTTP/2. However, Protocol Buffers is also usable without gRPC as a serialization framework. Indeed, a message encoded with Protocol Buffer is generally lighter than the equivalent in JSON, thus faster to transfer over the network. The use of generated encoder/decoder functions also ensures speed, compared to JSON. Nevertheless, using a binary protocol like Protocol Buffers also have some drawbacks: each party wanting to communicate with this type of encoding technology has to be compatible with it. We also need to generate files for each language that we want to use, and so we have to ensure that Protocol Buffers is compatible with the target programming language. These issues are common to every binary serializing technology. However, Protocol Buffers also have room for improvement: the decoding step can be avoided to lead to a greater speed. That is the goal of the successor of Protocol Buffers: FlatBuffers.

FlatBuffers. It is an efficient cross platform serialization library and it was originally created at Google for performance critical applications. What makes FlatBuffers special is that it represents hierarchical data in a flat binary buffer, in such a way that it can still be accessed directly without parsing and unpacking, while also still supporting data structure evolution. FlatBuffers require only small amounts of generated code, and just a single small header as the minimum dependency, which is very easy to integrate. According to benchmarks, it is lighter than JSON.

3.3 Storing data

MongoDB MongoDB is an open-source document-oriented database program. It supports sharding, which permits horizontal scaling by dividing a collection of documents across a cluster of nodes, thus making reads faster. In addition,

Mongo offers replication in two modes: master-slave and replica sets. Mongo is schema-less, that means it will store any document you decide to put into it. There is no upfront document definition requirement. Ultimately, documents are grouped into collections, which are equal as tables in a relational database. Collections can be defined on the fly as well. Documents are stored in a binary JSON format, called BSON, and encapsulate data represented as name-value pairs. JSON documents in Mongo do not force particular data types on attribute values. That is, there is no need to define the format of a particular attribute. Working with MongoDB is not without challenges. For starting, Mongo requires a lot of memory, preferring to put as much data as possible into working memory in order to have fast access. Besides, data is not immediately written to disk after an insert and a background process eventually writes unsaved data to disk. This makes writing extremely fast, but corresponding reads can occasionally be inconsistent. As a result, running Mongo in a non-replicated environment courts the possibility of data loss. Furthermore, Mongo does not support the notion of transactions, which is a touchstone of the database world. As with traditional databases, indexing in Mongo must be thought through carefully. Improperly indexed collections will result in degraded read performance. Moreover, while the freedom to define documents at will provides a high degree of agility, it has repercussions when it comes to data maintenance over the long term. Random documents in a collection present search challenges.

InfluxDB InfluxDB is a time series, metrics, and analytics database. Time series databases are designed to tackle the problem of storing data resulting from successive measurements made on a period of time. This data consists of items such as system metrics. The longer a system operates, the greater the amount of data accumulated. InfluxDB provides a solution for efficiently storing this data. Indeed, the InfluxDB data model has key-value pairs as labels, which are called tags. In addition, InfluxDB has a second level of labels called fields, which are more limited in use. InfluxDB supports timestamps with up to nanosecond resolution. InfluxDB uses a variant of a log-structured merge tree for storage with a write ahead log, sharded by time. This is much more suitable to event logging. Influx accepts queries via an SQL-like query language. It already supports filtering using where clauses, in addition to aggregates using group by, merge and join. InfluxDB also includes a feature called continuous queries, which allows users to "precompute expensive queries into another time series in real-time". Language bindings already exist for Javascript, Ruby, Python and Node.js. However, according the purpose, we may find that interacting directly with the HTTP API was already simple enough. Coupled with Grafana which is a visualization tool, InfluxDB allows data visualization by producing graphs and charts.

ElasticSearch Elasticsearch is a distributed search engine based on Apache Lucene. It has become one of the most popular search engines, and is commonly used for log analytics, full-text search, and operational intelligence cases. When coupled with Kibana, a visualization tool, Elasticsearch can be used to provide

real time analytics using large volumes of log data. Elasticsearch offers REST API, a simple HTTP interface, and uses schema-free JSON documents making it easy to index, search, and query data. *Elasticsearch* uses an index to achieve fast search responses.

4 System implementation

The technologies chosen for the proof of concept system are the following:

- Database: ElasticSearch
- Data visualisation: Kibana
- Data format: Flatbuffers
- Communication library: ZeroMQ
- Programming languages: NodeJS and Golang
- Image processing library: OpenCV
- Video reading library: JSMPEG

4.1 Multimedia transmission

The system implements a multimedia server in Node.js that offers an access point available to UAVs allowing them to send their video streams, and an access point to allow web clients to retrieve the stream. With this method using the publisher/subscriber pattern, the server automatically manages the different UAVs in a completely independent and transparent way (Figure 2).

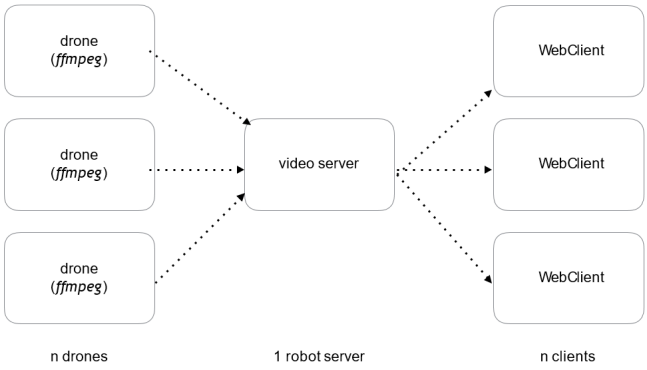

Fig. 2. Multimedia transmission architecture.

4.2 Architecture adaptation and fault tolerance

The autonomy of UAVs in the system facilitates two aspects of faut tolerance:

– Error confinement with the isolation of the suspected faulty agent so as to preserve the system reliability;
– System readjustment – agents have adaptability capacities that will ensure in case of loss of some agents, the continuation of services.

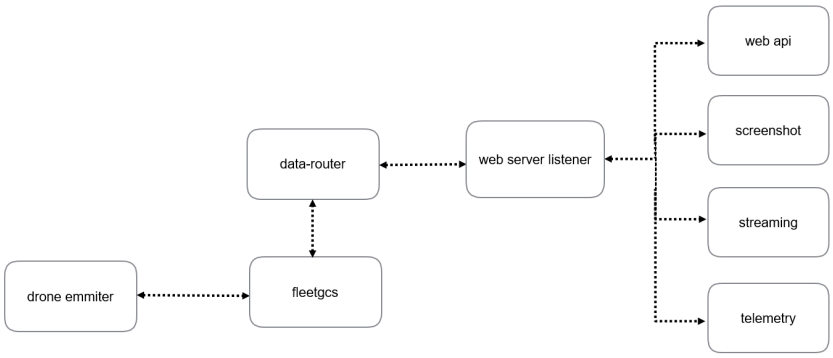

Fig. 3. Multi-agent system approach.

Our architecture includes different units that can be called agents with the multi-agent system approach. Indeed, these units are independent and operate autonomously, the whole communicating via ZeroMQ. We can notice that the architecture has been sufficiently decomposed so that the units have well identified services. For example, the UAV emitter only takes care of sending information and video streams while the screenshot unit deals only with snapshot and image processing (Figure 3).

4.3 Security

The purpose of this part is to enumerate quickly the security vulnerabilities that have been considered in our architecture and that motivated some of our technologies choices. As described before, the data-streaming architecture relies on two encoding technologies: *Mavlink* and *FlatBuffers*.

Mavlink Security-wise, the *Mavlink* protocol offers a 12-round RC5-based message encryption. Such encryption is considered efficient for text up to 2^{44}-bit length. Mavlink security issues are tackled extensively in [9] and [10].

FlatBuffers Messages are binary-encoded but not encrypted by default. Thus, we need to encrypt messages with a symmetric key, which we can encrypt itself with an asymmetric key. RSA encryption is recommended for its reliability.

5 Conclusion

This paper presents an overview of technologies useful for building a system architecture for data and video streaming with UAVs. It also details the design and the implementation of a system of this kind by properly selecting the right technology.

Acknowledgments

This work has been carried out in the framework of the FUI project AIRMES (Heterogeneous UAVs cooperating within a fleet) and IMATISSE (Inundation Monitoring and Alarm Technology in a System of SystEms) project, which is funded by the Region Picardie, France, through the European Regional Development Fund (ERDF).
The authors would like to thank *Syntony GNSS*, which employs Borey Uk, for adapting their schedule so that this work could be carried out.

References

1. M. Erdelj, E. Natalizio, K. R. Chowdhury and I. F. Akyildiz. Help from the Sky: Leveraging UAVs for Disaster Management. In *IEEE Pervasive Computing*, 16(1):24–32, 2017.
2. M. Erdelj, M. Król, E. Natalizio. Wireless Sensor Networks and Multi-UAV systems for natural disaster management. In *Computer Networks*, 124:72–86, 2017.
3. M. Bajer, *Building an IoT Data Hub with Elasticsearch, Logstash and Kibana.* 2017 5th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW).
4. D. K. Krishnappa, D. Bhat and M. Zink, *DASHing YouTube: An analysis of using DASH in YouTube video service.* 38th Annual IEEE Conference on Local Computer Networks, Sydney, NSW, 2013, pp. 407-415.
5. N. Bouzakaria, C. Concolato and J. Le Feuvre, *Overhead and performance of low latency live streaming using MPEG-DASH.* IISA 2014, The 5th International Conference on Information, Intelligence, Systems and Applications, Chania, 2014, pp. 92-97.
6. D. Antón, G. Kurillo, A. Yang and R. Bajcsy, *Augmented Telemedicine Platform for Real-Time Remote Medical Consultation.*
7. M. Kleppmann, *Designing Data-Intensive Applications*, O'Reilly Books, 2017.
8. R. Ravindran, A. Chakraborti, S. O. Amin, A. Azgin and G. Wang, *5G-ICN: Delivering ICN Services over 5G Using Network Slicing.* IEEE Communications Magazine, vol. 55, no. 5, pp. 101-107, May 2017.
9. K. Domin, E. Marin and I. Symeonidis, *Security Analysis of the UAV Communication Protocol: Fuzzing the MAVLink protocol.* Proceedings of the 37th Symposium on Information Theory in the Benelux. Werkgemeenschap voor Informatie-en Communicatietheorie, January 2016.
10. Butcher, A. Neil et al, *Securing the MAVLink Communication Protocol for Unmanned Aircraft Systems*, 2014.

11. S. Zhao, Z. Li, and D. Medhi, *Low delay MPEG DASH streaming over the WebRTC data channel*, 1-6. 10.1109/ICMEW.2016.7574765, 2016.
12. B. Li, Z. Wang, J. Liu, and W. Zhu, *Two decades of internet video streaming: A retrospective view*, ACM Trans. Multimedia Comput. Commun. Appl. 9, 1s, Article 33, 2013.
13. T. Lohmar, T. Einarsson, P. Frojdh, F. Gabin, M. Kampmann, *Dynamic adaptive HTTP streaming of live content*, World of Wireless, Mobile and Multimedia Networks (WoWMoM), IEEE International Symposium, 2011.
14. A. Borysov, *Enabling Googley microservices with HTTP/2 and gRPC*, JavaDay Kyiv, 2016.
15. D. Pohl, S. Nickels, R. Nalla and O. Grau, *High quality, low latency in-home streaming of multimedia applications for mobile devices*, 2014 Federated Conference on Computer Science and Information Systems, Warsaw, 2014, pp. 687-694.