Seamless Remote Rendering of DICOM Images

Krešimir Jozić
Industrial Applications
Siemens Energy
Zagreb, Croatia
kresimir.jozic@siemens-energy.com

Alan Jović
Faculty of Electrical Engineering and Computing
University of Zagreb
Zagreb, Croatia
alan.jovic@fer.hr

Željka Mihajlović
Faculty of Electrical Engineering and Computing
University of Zagreb
Zagreb, Croatia
zeljka.mihajlovic@fer.hr

***Abstract*—Many solutions display DICOM images locally or in a web browser but require that the full DICOM images be transported over the network or on a physical medium. As DICOM image size and the number of DICOM images increase, their transport over the Internet becomes more difficult. Contemporary computer resources used by physicians cannot keep up with the ever-increasing amount of new DICOM data, which complicates and prolongs diagnosis. This problem is exacerbated in the COVID crisis, making it difficult to examine patients, communicate between patients and physicians, and have physical access to DICOM images and volumetric data. In this paper, we present a distributed architecture for remote rendering of DICOM images. We propose an innovative web-based solution hierarchically divided into several levels that can be used to manage patient data, DICOM images, and radiology reports based on those images. The architecture is organized to leverage existing hardware, storage, and telecommunications infrastructure. We have approached the solution in such a way that the images are stored in healthcare facilities, preferably where they were taken, and only the necessary data is securely and seamlessly transported to the physicians. Our carefully engineered solution shows significant improvements in terms of rendering time and speed of data transfer, which is about 110 ms for an image with a resolution of 1996 by 2457 pixels, achieving seamless user experience for at least Full HD images.**

Keywords*—DICOM, remote rendering, hierarchical architecture, web application

I. Introduction

DICOM (Digital Imaging and Communications in Medicine) is the international standard for medical images and related information developed by the American College of Radiology (ACR) and the National Electrical Manufacturers Association (NEMA). It is divided into 22 parts and is republished several times a year [1]. It is used to store, exchange, and transmit medical images of different modalities (PET, CT, MR, etc.) and is a good basis for PACS (Picture Archiving and Communication System) systems. In addition to the data format, DICOM describes several layers of the ISO/OSI network model – WADO (Web Access to DICOM Persistent Objects) [2]. Due to its popularity, DICOM is used not only in large medical facilities, but also in dental clinics and veterinary practices. Current solutions usually show local DICOM images, or may employ a web browser for viewing, but require that the full DICOM images be transported over the network or on a physical medium, such as CD, DVD, or USB drive to a local computer. As network speeds increase and the cost of telecommunication services decreases, transporting DICOM images over the Internet is becoming easier. At the same time, however, DICOM image size and the number of DICOM images are increasing, so that transport over the Internet is once again becoming a problem. On the other hand, the computer resources used by physicians cannot keep up with the larger amount of new DICOM data, which complicates and prolongs diagnosis and can have serious consequences for patient health. This problem is exacerbated in the current COVID crisis, making it difficult to examine patients, communicate between patients and physicians, and have physical access to DICOM images.

Recently, there have been many approaches to remotely render DICOM images and display them on desktop computers [3], mobile devices [4], or both [5]. The mobile devices are usually underpowered and with limited hardware capabilities. An approach using desktop computer viewer often requires installation of many software libraries and applications, which leads to a slow and unsecure environment. As an alternative to desktop application, a web browser-based solution can be used [6], [7], [8] which is much more fitting for the task.

However, web browser-based solutions are currently much too slow and underoptimized, mostly due to some improper choices regarding system architecture. Therefore, our first task was to discover the most appropriate system architecture for seamless rendering of DICOM images. After the examination of a complex collaborative environment [9], we concluded that the solution should be distributed, hierarchical, and should export its functionality by using RESTful API. The next step was to find a proper way for rendering images. We have considered rendering libraries and game engines [10] but found that they are too complex inefficient for simple rendering algorithms that are required for the task. Therefore, we have decided to implement image rendering directly in OpenGL, storing data in textures and implementing algorithms that run on a GPU [4], [11]. Finally, we had to find a solution for storage of DICOM images. Lately, cloud-based or hybrid (edge + cloud) solutions are popular [12]. Our conclusion was that this approach is inappropriate for our goals, due to data confidentiality requirement, low response time, and high quantity of transferred data.

The main innovation of this work is a web based, hierarchical system for rendering DICOM images that is easy to install and use. Data confidentiality and high response time were also considered. Because of semiconductor shortage on the market, we have strived to find an engineering solution which leverages existing computer equipment by directly implementing algorithms in OpenGL, thus ensuring a high level of efficiency and seamless user experience.

II. System Arhitecture Description

The system architecture consists of a database, a backend, and a frontend. The database is CockroachDB – Core edition. The backend was written in the programming language Go. For the frontend, the frameworks: CSS, HTML5, TypeScript, and Angular were used.

The system is designed to be easily portable and consists of only two files. One is the database, the other is the backend. No installation is necessary, the files only need to be copied and a configuration file in YAML format for the backend needs to be created. After that, the database and backend can be run.

The backend has four functions. The function is selected via a configuration file, as are other parameters, such as IP address, port, secret JWT key, etc., Fig. 1.

The first function is a web server. It is used to serve HTML, CSS, and Javascript files that make up the frontend. The Angular CLI tool is used to pack the frontend into several production-ready files. When the backend is compiled, these files are stored in a virtual file system that resides inside the backend executive file.

The second function is a top-level server that serves RESTful API and is intended for serving data that is stored in a central location and shared among all facilities and physicians. For example, a top-level server is used at the country level. The data stored is a list of facilities, physicians and patients, as well as various metadata required for the system to function (IP addresses of servers, types of DICOM images, etc.).

The third function is an institutional server that serves the RESTful API and is intended for serving data stored at the location of a medical facility. These are data from radiology reports and a list of DICOM images stored on that facility's servers. Metadata such as IP address, MD5 sums of DICOM images and storage locations are also stored.

The fourth function is a node that is used to store and render DICOM images. The DICOM images do not necessarily have to be stored at that node but should be accessible from there on storage systems like NAS or SAN [13].

An overview of the entire architecture is depicted in Fig. 2. Since there is no direct communication between backend servers, neither on the same hierarchy level, nor between hierarchy levels, multiple instances of the same backend server can run simultaneously. For example, the top-level server will surely be heavily loaded with hundreds of thousands or millions of queries per day, and multiple instances will be needed to serve all the requests. All instances of the top-level server may use the same database, but there may also be multiple databases – not necessarily the same number as the servers.

```
name: "Toplevel server"
serverType: "toplevel"
address: "127.0.0.1"
port: 8081
databasePort: 9000
JWTPassword: "secret"
debug: true
```

Fig. 1. Example of the backend configuration file.

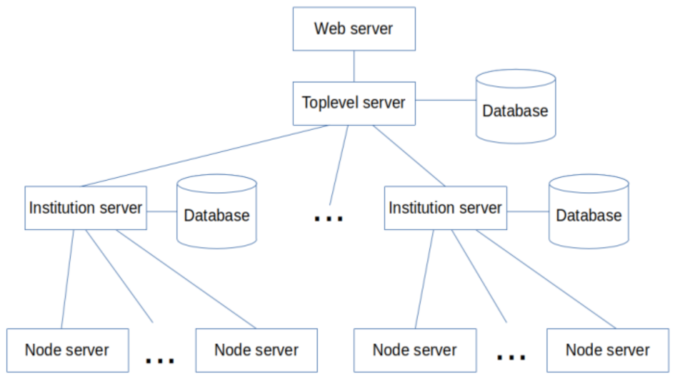


Fig. 2. System architecture overview.

This is possible because of the ease of replication offered by the CockroachDB database. It is a good example of a hybrid database that uses the best of SQL and NoSQL databases. The Postgres-compatible SQL language is used to write queries and manipulate data, while NoSQL is used at the storage level because it is easy to build a scalable cluster at the local or global level.

Each backend function corresponds to one of the four hierarchical levels of a system. User interaction with the system begins with loading the frontend (level 1). Due to the confidentiality of medical data, the minimum security used is the HTTPS protocol, and the HTTP/2 protocol is preferred due to its advantages in security, traffic multiplexing, congestion control, and compression. The HTTP/3 protocol is also supported, although it is still under development [14].

The user logs into the system and the top-level server generates a JWT token that is later used to access all system resources. The JWT token contains the username, the user’s permissions, the information about whether the user is a physician, which facility the user works for, and the time the token was generated. Simplified JWT tokens in decoded and encoded forms are shown in Fig. 3. and Fig. 4. The token contains all the required data, because otherwise the user’s permission would have to be checked every time the RESTful API is accessed on a particular server, which would increase the load on the top-level server and slow down user interaction with the system. Although the secure HTTPS protocol is used, there is a theoretical risk that the token could be intercepted due to exploits in the protocol implementation [15]. The JWT token provides a second layer of protection by using a secret key to create a checksum of the payload. Even if someone manages to intercept the JWT token and change the data (e.g. user permissions), verification of such a key will not be successful because the secret key is not transported but stored on the backend servers. If someone manages to break into one of the backend servers and obtain the secret key, recovery is very fast and can be done without shutting down the system.

```
Header: Algorithm and token type
{
 "alg": "HS256",
 "typ": "JWT"
}
Payload: useful data
{
 "name": "John Doe",
 "issued_at": 1516239022
}
Signature:
HMACSHA256(
 base64UrlEncode(header) + "." +
 base64UrlEncode(payload),
 )
```

Fig. 3. A decoded JWT token.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
uYW1lIjoiSm9obiBEb2UiLCJpc3N1ZWRfYX
QiOjE1MTYyMzkwMjJ9.9k4Q31i3_WCZpe7z
BlGWgKC0ZYS1hPyY2boYQrJhS1U
```

Fig. 4. An encoded JWT token.

It is necessary to enter a new secret key into the configuration files of all the backend servers and send a signal SIGUSR1 to the backend servers requesting the backend servers to reload the configuration file on the fly. The final level of protection is the renewal of the JWT token. The session data is not stored on the server or in cookies on the client, but all the required data resides in the JWT token, which the frontend automatically renews every few minutes. If the backend receives a token that is older than a few minutes in case of a connection loss or for any other reason, it rejects such a request.

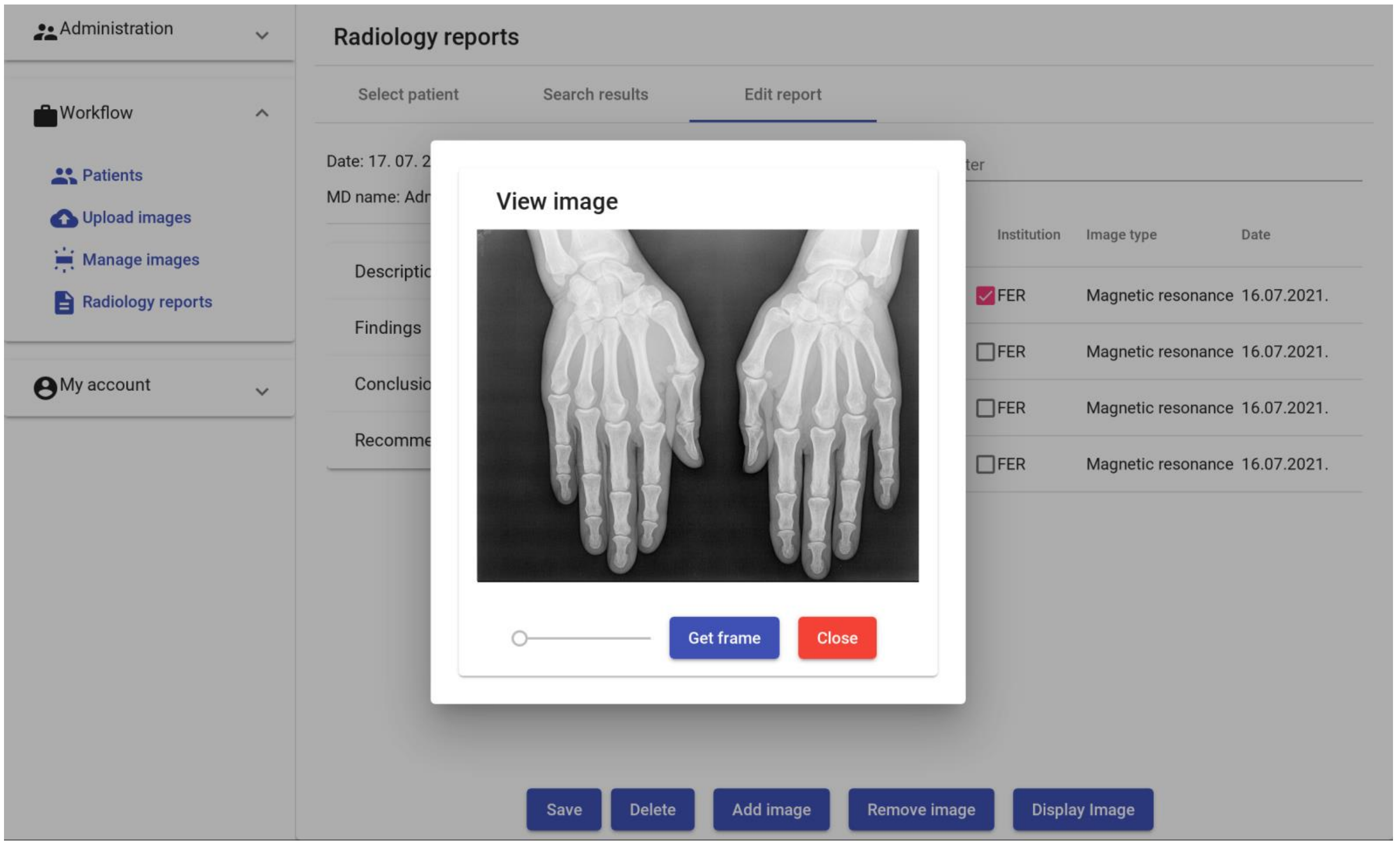
After logging into the system, the user interacts with the system at any of the levels 2 through 4. If the user wants to view or change patient data, the interaction between the frontend and backend takes place on the second level (top-level server). If the user wishes to search for radiological findings or DICOM images, the interaction takes place on the third level (facility). If the user wishes to add or view new DICOM images, the interaction takes place on the fourth level (rendering node within the facility). Thus, the interaction always takes place directly between the frontend loaded in the user's web browser and the backend server where the requested data resides. In this way, the servers are relieved of hierarchical data transfer. Only the network resources are burdened, but this is unavoidable anyway, since all required data must be transferred. However, even in this case, care is taken to keep the total amount of data transferred over the chargeable connection (Internet) as low as possible. For example, uploading DICOM images is best done within the facility where the images were created. If the image is on an optical or USB medium in another facility, it will be uploaded to the servers of that facility. In other words, DICOM images are stored on the server that is physically closest to them. This avoids the transmission of DICOM images over the Internet. Transmission over the Internet is not only chargeable, but also usually much slower than within a facility.

The user interacts with the system at the first three levels through many small packets containing textual data. The servers on the first three levels need only have sufficient memory and CPU power. Interaction at the fourth level consists of a smaller number, but much larger packets, containing binary data or binary data base64 encoded into text data. This means that the fourth level servers must also have a graphics card. Since the fourth level deals with large amounts of data (a few dozen MB up to several GB), the latency of data transfer between the frontend and the backend is large. Therefore, instead of waiting for a particular operation to complete, the backend returns the response to the frontend immediately after the operation is successfully started, the response to the frontend is returned, and the operation continues asynchronously on the backend.

An example of such an operation is loading a DICOM image. When the user wants to view the image, the user selects it and issues the command to load the image. The frontend forwards this command to the backend. The backend opens the DICOM image and allocates memory for the OpenGL 2D texture array. These operations take a very short time, and the user receives feedback within a few tens or hundreds of milliseconds. In the meantime, the backend loads the DICOM image into the OpenGL texture. The image is not fully loaded into the computer's RAM, and is then transferred to the graphics card's memory, but the memory is reserved for one layer (a 2D texture). The data from the drive is waited for and then the image is transferred to the graphics card's memory.

The slowness of the hard drive is used here to reduce the consumption of working memory, because while the next image is being loaded from the hard drive, the first one is already stored in the graphics card's memory. Hence, a certain small memory buffer is used to load an image that can be very large. It may take a few seconds for the entire DICOM image to load, but it is assumed that the user will not need to view the image immediately. The user first uses the slider to select the image layer he wants to preview, and then clicks the button to retrieve the image. This process takes a few seconds, and that is enough to hide the loading of the image on the backend server. This gives the user the impression that the system is responsive. The user can set the size of the window in which the image is displayed. The window size information is transmitted to the backend, which creates a rendering buffer of the same size (rendering is done off-screen in RAM). After selecting the desired image layer and possibly performing some operations on the image, the user sends a request to the backend, which activates the OpenGL context. Using OpenGL from a specific layer of the 2D texture array, the image is sent to a frontend that displays it to the user. An example of DICOM radiology image displaying on frontend is depicted in Fig. 5.

When OpenGL is used to display an image on the same computer, there is no frequent switching of the OpenGL context, while this is not the case in the client-server architecture. For each request that comes from the frontend, the



Date: 17. 07. 2

MD name: Adr

Institution	Image type	Date
✓ FER	Magnetic resonance	16.07.2021.
FER	Magnetic resonance	16.07.2021.
FER	Magnetic resonance	16.07.2021.
FER	Magnetic resonance	16.07.2021.

Fig. 5. Screenshot of the frontend rendering of a DICOM radiology image.

OpenGL context has to be reactivated. This context switching is compensated using hardware acceleration for rendering, so that this whole process takes a relatively short time, and the user still has the impression that the system is responsive, thus achieving seamless rendering.

Since the operations on an image are not algorithmically complex, an integrated graphics card can also be used. The most important thing is that the computer has enough working memory. Therefore, a timer is used that clears the texture from memory when the user is inactive for a certain period of time. If the user wants to continue working, he must reload the image. This promotes fair use of computer resources, reduces the need to purchase new hardware, and extends the life of existing hardware.

III. Results

Two types of benchmarks were performed on the implemented system. The first one is the loading time benchmark, where we measured how long it takes to load a DICOM image into an OpenGL texture array, as shown in Table I. The second benchmark concerns the rendering and transport times. The second benchmark measured the total time between clicking a button, rendering on a backend, and transporting over the network, Table II.

PC hardware configuration used in both benchmarks consisted of:

TABLE I. Benchmark Results of DICOM Images Loading Times

Image	Rows	Cols	Layers	File size	Load time
A	512	512	10	5 MB	0.14 s
B	512	512	54	27 MB	0.61s
C	512	512	555	277 MB	5.06 s
D	1996	2457	84	786 MB	13.49 s
E	1996	2457	96	898 MB	14.58 s

TABLE II. Benchmark Results of Rendering and Transport Times

Image	Image size	Total time
A	40.88 KB	31 ms
B	56.79 KB	31 ms
C	76.07 KB	34 ms
D	1.17 MB	145 ms
E	1.10 MB	110 ms

- Intel i7-11700, base clock 2.5 GHz, max clock 4.9 GHz, 8C16T

- Integrated Intel UHD Graphics 750
- 64 GB RAM, DDR4-3200, dual channel
- Samsung SSD 970 EVO 1TB, NVMe
- OpenZFS 2.0.2 file system

The second benchmark used Firefox web browser with network speed throttling enabled, using the predefined "Regular 4G/LTE" profile (4/3 Mbps down/up, 20 ms latency). We compared the results with two similar papers [5], [6]. The paper [5] uses a portable radiology workstation, transmits images over a network, and displays them within a web browser. The results do not state the transmission times, but qualitatively state that they have achieved near real-time response on the WiFi network and that the solution is usable on the 3G network. According to [16], the response time of 100 ms is perceived as instantaneous, so we can take this value as the limit of real-time. Near real-time response in this case can be characterized as a value in the range of 100 ms to 1 s. In our examples, with images A, B and C (resolutions 512x512), the response times were about 34 ms, so they are well within the real-time limit. With images D and E, we are a little above that limit, so we can say with certainty that we achieved a real time response for at least Full HD resolution (1920x1080).

In the second paper [6], several different experiments with several different image sizes, using solutions made in Java, Flash, and HTML5, were made. We compare a specific case when they used a 512x512x500 resolution image of 250 MB and an HTML5 solution. Their solution first saves the selected image on a local computer, while with us only the final image is downloaded. Therefore, we need to set a frame within which we can compare results. The time until the first image is displayed includes image transfer, image loading and processing, and display. They took 23.19 s to transfer the image, did not specify how long it took to load, and processed it in 0.0151 s. Thus, from the beginning of the loading process to the display of the image, it took about 23.2 s, with an unknown loading time. We compare the results with the image C, which is slightly larger. Loading took 5.06 s, and processing and uploading 0.034 s, which totals about 5.1 s. This is the amount that the user would have to wait in case both activities are run sequentially immediately after each other, which is usually not the case. The user first starts loading the image, slides the image he wants to display and clicks on the image retrieval button. In this case, the upload time is obscured by the user's activity when selecting the image, and the image display after clicking on the retrieval button is almost instantaneous.

IV. Conclusion

In this paper we presented a distributed, hierarchical system for remote rendering of DICOM images. Our motivation was to create a system which is simple, easy to install and use, and which provides functionality without investments in additional hardware or telecommunication services.

The benchmark test of DICOM image loading from disk to OpenGL texture array showed that the latency of loading files up to several hundred megabytes in size can be successfully hidden from the user and thus provides a seamless interaction with the system. The benchmark test of file rendering and transport of the final image from the backend to the frontend showed that the total time is in the range of 100 ms, even for images with large dimensions, which is quite satisfactory.

For future work, we plan to extend the functionality of the system with more complex algorithms, e.g., volume rendering from OpenGL texture array. We also plan to support images loading from encrypted file systems (such as ZFS) for increased data security, as well as make further improvements to the frontend usability in terms of more advanced image viewing and handling functions.

References

[1] NEMA, "DICOM." http://dicom.nema.org/ (accessed Jul. 29, 2021)

[2] M. Mustra, K. Delac, and M. Grgic, "Overview of the DICOM standard," in Proceedings of the 50th International Symposium ELMAR 2008, 2008, vol. 1, pp. 39–44.

[3] B. F. Tomandl, P. Hastreiter, C. Rezk-Salama, K. Engel, T. Ertl, W. J. Huk, R. Naraghi, O. Ganslandt, C. Nimsky, and K. E. Eberhardt, "Local and Remote Visualization Techniques for Interactive Direct Volume Rendering in Neuroradiology," RadioGraphics, vol. 21, no. 6, pp. 1561–1572, Nov. 2001.

[4] M. Zhang, "Server-Aided 3D DICOM Image Stack Viewer for Android Devices," International Journal of Digital Content Technology and its Applications, vol. 6, issue 1, pp. 208–217, Jan. 2012.

[5] K. Thilagavath and M. Kavitha, "Sparkmed: A framework for Multimedia Medical Data Integration of Adaptive Mobile Object in Heterogeneous Systems," International Journal of Science and Modern Engineering (IJISME), vol. 1, no. 12, pp. 1–4, 2013.

[6] Q. Min, X. Wang, B. Huang, and L. Xu, "Web-Based Technology for Remote Viewing of Radiological Images: App Validation," J. Med. Internet. Res., vol. 22, no. 9, e16224, Sep. 2020.

[7] A. Arbelaiz, A. Moreno, L. Kabongo, H. V. Diez, and A. García Alonso, "Interactive Visualization of DICOM Volumetric Datasets in the Web - Providing VR Experiences within the Web Browser:," in Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications, Porto, Portugal, 2017, pp. 108–115.

[8] B. Blazona and Ž. Mihajlović, "Visualization Service Based on Web Services," CIT. Journal of Computing and Information Technology, vol. 15, no. 4, pp. 339–345, 2007.

[9] N. T. Karonis, "High-Resolution Remote Rendering of Large Datasets in a Collaborative Environment," Future Generation Computer Systems, vol. 19, pp. 909–917, Aug. 2003.

[10] G. Wheeler, S. Deng, N. Toussaint, K. Pushparajah, J. A. Schnabel, J. M. Simpson, and A. Gomez, "Virtual interaction and visualisation of 3D medical imaging data with VTK and Unity," Healthcare Technology Letters, vol. 5, no. 5, pp. 148–153, Oct. 2018.

[11] S. Belyaev, V. Chukanov, and V. Shubnikov, "Bump Mapping for Isosurface Volume Rendering," Journal of Image and Graphics, vol. 5, no. 2, pp. 68-71, December 2017. doi: 10.18178/joig.5.2.68-71

[12] S. Sondur, K. Kant, S. Vucetic, and B. Byers, "Storage on the Edge: Evaluating Cloud Backed Edge Storage in Cyberphysical Systems," in 2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems (MASS), Monterey, CA, USA, Nov. 2019, pp. 362–370.

[13] RedHat, "Whatis network-attached storage?," https://www.redhat.com/en/topics/data-storage/network-attached-storage (accessed Jul. 29, 2021)

[14] IETF, "Hypertext Transfer Protocol Version 3 (HTTP/3) draft-ietf-quic-http-34,"https://datatracker.ietf.org/doc/html/draft-ietf-quic-http (accessed Jul. 29, 2021)

[15] J. Ćurguz, "Vulnerabilities of the SSL/TLS Protocol," in Proceedings of the 6th International Conference on Computer Science, Engineering and Information (CS & IT), May 2016, pp. 245–256.

[16] R. B. Miller, "Response time in man-computer conversational transactions," in Proceedings of AFIPS Fall Joint Computer Conference, December 9–11, 1968, pp. 267–277.