Smart User Control in Wi-Fi Access Points

Bart Leenheer
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
b.leenheer@student.utwente.nl

ABSTRACT

Current cellular networks use a method called "slicing" to improve speed and stability per user. Each slice has different performance and service requirements, and each user can be assigned the slice which comes closest to these specifications. This functionality is not present on the current state of the art routers, which can impact user experience significantly, especially when multiple users are connected to the same access point. This research aims at investigating and collecting metadata from Wi-Fi packets in specific use cases, which can act as a basis for an implementation similar to slicing in mobile networks. Furthermore, a proof of concept implementation will be presented that includes a wireless optimization process and can act as a basis for a possible "slicing" implementation.

Keywords

Wi-Fi, slicing, cellular, networks, improvement, stability, metadata, channel, bandwidth, optimization

1. INTRODUCTION

Wireless Fidelity networking (Wi-Fi) is an invention which is currently being used all across the globe and the internet is becoming more available to people as we speak [1]. However, Wi-Fi access points still lack in some aspects, such as the control over how much bandwidth is assigned to specific devices [2]. This means that someone who is watching Netflix [3] might get information with a very low latency, whilst it does not need it due to its buffering capabilities; in the meantime, someone who is playing a video game might have sub-optimal performance, because this requires low latency, but not as much bandwidth. This problem becomes increasingly noticeable when a high amount of devices are connected to the same access point.

Cellular networks, on the other hand, already have this problem solved. Most of this optimization is done by slicing the single network into multiple smaller networks that each have their own performance and service requirements [4]. For example, one layer (or slice) could focus on low latency and another could focus on having a low error rate. Assigning the correct slice to the user, based on their requirements, will optimize the user experience by providing what they need.

It should be possible to optimize Wi-Fi access points in a similar way to this cellular optimization. To achieve this, a good start would be made by analyzing the metadata present in Wi-Fi packets, as this could indicate performance and service requirements to make optimal use of the Wi-Fi connection [5]. With a clear overview of the metadata in specific use cases, statistics can be extracted. This properly analysed data could then be used to develop a smarter Wi-Fi access point - by implementing an algorithm similar to slicing - to improve overall speed and reliability.

2. PROBLEM STATEMENT

Even though Wi-Fi exists for quite some time now, there are currently no commercially available Wi-Fi access points that use optimizations similar to those of cellular networks; even simpler optimizations, such as optimal channel and bandwidth selection, are lacking. As mentioned before, Wi-Fi works pretty well when a small number of devices is connected, but the performance decreases significantly when this amount increases, Especially if these devices all have different use cases. Implementing network slices, comparable to those in cellular networks, should improve stability and performance. Furthermore, performance can be impacted when neighbouring access points have overlapping channels. Making use of better channel selection should improve speed and stability as well.

To try to improve these shortcomings, the following research questions have been composed:

RQ1 How can we extract useful information from a Wi-Fi access point and store it in a way such that it can be used to potentially improve network speed and stability?

RQ2 How can a Wi-Fi access point be improved using gained insight into metadata regarding specific use cases?

3. SLICING IN 5G

5G makes use of a technique called slicing. This technique allows one physical network to be split up into multiple logical networks, each containing its own performance and service requirements [4]. This way, the network is suitable for many different use cases, each of which can make use of a slice tailored for optimal performance. For example, high-resolution video streaming in dense areas requires a high average bit rate but does not (necessarily) need a low-latency connection. On the other hand, extreme real-time communication, such as remote robotic control in health care, requires an extremely stable connection with a very low latency [6]. Besides these two, there are a lot more use

35th Twente Student Conference on IT July. 2nd, 2021, Enschede, The Netherlands.

cases and it is likely that more will emerge. This makes using mobile slices in 5G seem very valuable.

5G seems to cover a bigger amount of use cases in comparison to Wi-Fi access points; for example, mobile internet in high-speed trains [6]. However, even within home networks, there can be a numerous amount of use cases, such as playing video games or streaming videos. Therefore, it could be valuable to implement a similar technique in Wi-Fi access points.

4. METHODOLOGY

In this section, the approach to tackling the research questions will be explained. The first step is to obtain an access point that can run custom firmware. This research has been conducted with OpenWrt, which is a Linux operating system that runs on routers and similar devices [7]. This operating system allows the user to add packages and modify the file system directly, which are useful features to have when doing research. The hardware used in this work is a Raspberry Pi 4 instead of an actual router or access point. This, unfortunately, means that there have been some hardware restrictions. This will be further elaborated upon in the **Discussion**.

The first question requires the investigation of metadata from Wi-Fi packets and the extraction of the right data and statistics. At first, a couple of packages have been looked into such as Kismet [8] and aircrack [9]. However, it became evident that these packages (and similar ones) have a focus on hacking and are therefore quite extensive and complex. A simpler approach is to use the tcpdump package. With this package installed, it is possible to run the command via the Secure Shell Protocol (SSH) [10] and send the results of the tcpdump straight into your locally running instance of Wireshark [11] (see **Listing 1**). Once the capture is done, it is saved and then the packets can be inspected individually and the overall statistics can be extracted.

After this analysis, a possible slicing implementation has been researched (see **Section 5.2.1**), however, due to hardware limitations (see **Discussion**) this implementation is not possible using the current setup. Instead, a proof of concept implementation is created - using OpenWRT, SSH and bash scripts [12] - that improves channel and bandwidth selection. The first bash script executes on a laptop running Linux [13] and the second is placed on the access point and executed through the first script.

5. RESULTS

5.1 Packet capturing

As explained in the **Methodology**, the packets could be captured by running a tcpdump over SSH and then sending the output into Wireshark, as shown in **Listing 1**. When the relevant information is captured, it can be saved and inspected in Wireshark. Within Wireshark, the number of packets, capture time and total byte size can be extracted. From this data, statistics - such as average packet size, bytes per second or packets per second - can be calculated. Furthermore, Wireshark gives information about the protocols used and the amount of Transmission Control Protocol (TCP) Errors. The tables containing this information per capture can be found in **Appendix A**.

The packet captures show interesting details about the usage for different use cases. In **Table 2** it becomes evident that the packet sizes are significantly smaller when playing video games than downloading or uploading or even casually browsing the internet. This is likely due to the high importance of the speed at which the packets are received while gaming (reducing latency); to realize this, the packets are kept small. And for the other cases, the amount of transferred data is quite high, so in order to reduce loading times, more information is sent per packet. In the same table, we can see that the average bytes and packets per second are at their highest whilst downloading or uploading. This number is lower for casual browsing because, for that use case, the client only transmits and receives packets whilst loading the page (or when the page updates).

An interesting thing to note when inspecting the protocols used in the captures (see **Table 3**) is that a significant amount of traffic makes use of the QUIC [14] protocol. This protocol was proposed by Google and according to Qi et al., [15] it might replace the Transmission Control Protocol (TCP) [16] in the near future because of its advantages in encryption, low latency and multiplexing. QUIC is based on the User Datagram Protocol (UDP) [17] and - compared to TCP - it can establish a reliable and secure connection quicker, especially when re-establishing a connection. Whilst QUIC is based on UDP, it does allow for a reliable data connection and is therefore suitable for use cases such as data transfer. Qi et al. do mention that under high load, QUIC can perform worse than TCP. This also seems the case in the downloads and uploads in the captures, where the TCP connection achieves a higher average bitrate than the QUIC connection (see **Table 3**). Furthermore, Wireshark does not seem to recognize outgoing QUIC packets, as evident in the *Drive Upload (QUIC)* capture where it is classified as Data under UDP.

Besides this, the division between protocols shows some more information regarding different use cases. The gaming captures primarily show (non-QUIC) UDP traffic, this is likely due to the low-latency performance requirement for gaming applications. Whilst UDP is faster than TCP, it is not reliable. This means that packets sent from the server might not reach the client and vice versa. Usually, only the latest information is relevant for gaming applications, therefore retransmission of lost packets is not necessary. File transfers or website visits often use TCP or - as of recently - QUIC, as these protocols do allow for reliable data transfer, which is required for these use cases.

5.2 Implementation

5.2.1 User control

For a possible implementation, the statistics from the packet captures mentioned in **Section 5.1** can be further analyzed and a lookup table can be created. Using this lookup table, the connected client can be categorized and, with this knowledge, the access point could move the user to a separate network (possibly running on the same access point) with the settings tailored to the user. This approach would be similar to network slicing, as explained in **Section 3**. The separate "slices" could have different (non-overlapping) channels and different bandwidths in order to cater to the needs of the users. For example, someone who needs a low latency connection can be moved to a channel with few connected users to reduce congestion, whilst someone who is streaming video could be connected to a more populated slice with a higher bandwidth.

This implementation requires the ability to create multiple wireless networks. Unfortunately, during this specific research, the hardware used was not able to do this (see **Discussion**), therefore this is unfortunately not a viable solution for this specific work.

```
ssh USER@IP tcpdump -i wlan0 -U -s0 -w - 'not port 22' | sudo wireshark -k -i -
```

Listing 1. Packet capturing command

5.2.2 *Optimal channel and bandwidth selection*

The final solution[1] is a proof of concept that acts as a start of what is possible with devices running OpenWRT. The basic idea is to run this script on a mobile device such as a laptop, which then changes the bandwidth and channel on the access point, waits until the device reconnects and then it tests the speed and stability of the connection through a series of ping commands. This is then done for each possible bandwidth and channel combination and a health score is calculated per combination (the lower the score, the better the connection). This health score is based on the average round trip time (RTT) and the amount of packet loss (see line 14 in **Algorithm 1**). After this is done, it selects the combination with the lowest health score and sets that as the channel and bandwidth on the access point. The pseudo-code for this is shown in **Algorithm 1**. Before the script starts determining the health score, it first establishes whether or not the remote channel and bandwidth are correctly set, by comparing them to the local channel and bandwidth. When this is not the case, it means that the script running on the access point has reset the channel and bandwidth to its default settings and therefore it will not include this combination in its results. A table containing the health scores of one run can be found in **Table 1**. In this run, channel 108 with 40MHz bandwidth has the lowest health score and is therefore the combination which provides the best connection.

Setting the channel on the access point is done by sending a command over SSH. The issue here is that commands sent over SSH stop executing as soon as the connection is lost. To solve this issue, a script should be placed on the access point that starts a process that is disowned. This moves the disowned commands to the background, which means that it keeps on running, even when the SSH connection is dropped. The pseudo-code for this script can be found in **Algorithm 2**. This script also works as a fail-safe; in case the access point switches to a channel and bandwidth combination that cannot be reached by the device executing the main script, it switches to a predetermined combination from which it is known that it works. Examples can be seen in **Table 1**; channel 116 and 140 did give a health score for 20 MHz, whilst they did not for 40 MHz.

Bandwidth	Channel	Health
20	36	568.96100
	40	772.36177
	44	611.48184
	48	654.13126
	52	736.18413
	56	655.73734
	60	638.62693
	64	775.84108
	100	789.22741
	104	934.07736
	108	735.817
	112	756.25678
	116	784.18870
	132	650.13761
	136	916.02584
	140	663.17615
40	36	459.49020
	40	621.42414
	44	854.44830
	48	579.90649
	52	441.90240
	56	612.80973
	60	755.65695
	64	430.49920
	100	459.70740
	104	490.30027
	108	368.57049
	112	393.04006
	132	757.25991
	136	461.04846

Table 1. Results optimal channel and bandwidth selection

6. DISCUSSION

Whilst one of the main goals of this research was to find a way to enable a form of smart user control in regular Wi-Fi access points, this goal was not feasible with the hardware available for this work. Within OpenWRT, it is possible to create multiple wireless networks, but the Raspberry Pi does not support this. While OpenWRT runs on this device, it is not made to be used as an access point. Regular access points, on the other hand, often have multiple antennas and are therefore able to create multiple wireless networks.

Besides limitations, the Raspberry Pi does have its benefits. The main one is that it is essentially a small computer and therefore contains features such as expandable storage, which allows bigger packages to be installed. For future research, the user control implementation could be further investigated by either using different hardware or adding wireless interfaces to the Raspberry Pi such as USB Wi-Fi dongles.

Because this implementation became infeasible in this research, live collection of statistics was not researched further as well. This would likely need to be investigated in order to determine the performance and service requirements of a specific user and assign them the correct "slice".

In the implementation proposed in this work, the health score is calculated by looking at the average round trip time together with the packet loss in a series of ping commands (see line 14 in **Algorithm 1**). Whilst this does give a good indication of the strength and stability of the connection, it might be better to calculate this health score differently, for example by taking the minimum and maximum

[1]The code used in this work is made available in this repository: `https://github.com/bartleenheer/Optimal-channel-selection`

Algorithm 1 Best channel and bandwidth selection

channels ← *All possible channels*
bandwidths ← *All possible bandwidths*
for all *bandwidth* ∈ *bandwidths* **do**
    for all *channel* ∈ *channels* **do**
        Change access point to *channel* and *bandwidth*
        Wait for reconnect
        if *local channel*, *bandwidth* ≠ *remote channel*, *bandwidth* **then** ▷ No connection established within the given timeout
            Skip this combination
        end if
        health ← 0
        for *interval* ← 0.1, 0.01, 0.001 **do**
            for *size* ← 10*kB*, 30*kB*, 60*kB* **do**
                Ping access point with *interval* and *size* 100 times
                health ← *health* + (*avg rtt* ∗ (1 + *pct. packet loss*))
            end for
        end for
        Store *health*, *channel*, *bandwidth* in result file
    end for
end for
bestchannel, *bestbandwidth* ← channel and bandwidth corresponding to lowest health
Change access point to *bestchannel* and *bestbandwidth*

Algorithm 2 Change channel (on access point)

channel ← *parameter -c*
bandwidth ← *parameter -b*
disown ▷ Making sure it still executes when SSH loses connection
    Turn wireless off
    Use *channel* and *bandwidth*
    Turn wireless on
    while No wireless connections **do**
        if Timeout reached **then**
            Return to default channel and bandwidth
            exit
        end if
        Sleep for 0.5s
    end while
end disown

round trip time into account or assigning more weight to packet loss. This way of determining the health score also does not take the different protocols into account, but only makes use of the Internet Control Message Protocol (ICMP) [18] used by the ping command. For a better health score, TCP, UDP and QUIC could also be taken into account as they can possibly have different performance under similar conditions. For future research, different ways of calculating the health score could be compared and the speed using different protocols could also be tested and incorporated in this score.

Another future research could look into including the preferred "slice" in the header of packets sent to the access point. This would require packet manipulation on the client-side, but it could speed up the categorization on the access point side because it only needs to read one packet header instead of gathering statistics from metadata per user.

7. CONCLUSION

Capturing packets and extracting data from access points running OpenWRT can give a good indication of certain performance and service requirements in specific use cases. Loading the data into Wireshark eases the process of data extraction, as it has capture statistics built-in. As mentioned in the **Discussion**, live statistic extraction was not researched in this paper, as it became irrelevant for the final implementation. In this case, it was sufficient to store the extracted data into a table. This gives a good overview of the data for different use cases.

With this data collected and inspected, it is possible to create a lookup table that links usage statistics to certain performance and service requirements. This could then be used to categorize connected clients in separate wireless networks that support these requirements. Unfortunately, as mentioned in the **Discussion**, this implementation became infeasible due to hardware limitations. Instead, this paper discussed a proof of concept implementation for optimal channel and bandwidth selection. This implementation acts as a basis in order to show what is possible within OpenWRT. Without the hardware limitations, the original implementation idea could likely be created, mostly because nearly everything within OpenWRT can be changed or monitored from the command line.

References

[1] H. R. Max Roser and E. Ortiz-Ospina, "Internet," *Our World in Data*, 2015, https://ourworldindata.org/internet.

[2] M. Nerini and D. Palma, *5g network slicing for wi-fi networks*, 2021. arXiv: 2101.12644 [cs.NI].

[3] *Netflix*. [Online]. Available: https://www.netflix.com/ (visited on Jun. 27, 2021).

[4] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina, "Network slicing in 5g: Survey and challenges," English, *IEEE Communications Magazine*, vol. 55, no. 5, pp. 94–100, 2017, Cited By :343. [Online]. Available: www.scopus.com.

[5] J. Hui and M. Devetsikiotis, "Metamodeling of wi-fi performance," English, in *IEEE International Conference on Communications*, Cited By :13, vol. 2, 2006, pp. 527–534. [Online]. Available: www.scopus.com.

[6] N. Alliance, "5g white paper," *Next generation mobile networks, white paper*, vol. 1, 2015.

[7] *Openwrt*. [Online]. Available: https://openwrt.org/ (visited on Jun. 27, 2021).

[8] *Kismet*. [Online]. Available: https://openwrt.org/docs/guide-user/network/wifi/wireless-tool/kismet (visited on Jun. 27, 2021).

[9] *Aircrack*. [Online]. Available: https://openwrt.org/docs/guide-user/network/wifi/wireless-tool/aircrack-ng (visited on Jun. 27, 2021).

[10] T. Ylonen, C. Lonvick, *et al.*, *The secure shell (ssh) protocol architecture*, 2006.

[11] *How to capture, filter and inspect packets using tcpdump or wireshark tools*. [Online]. Available: https://openwrt.org/docs/guide-user/firewall/misc/tcpdump_wireshark (visited on Jun. 27, 2021).

[12] C. Ramey, "Bash, the bourne- again shell," in *Proceedings of The Romanian Open Systems Conference & Exhibition (ROSE 1994), The Romanian UNIX User's Group (GURU)*, 1994, pp. 3–5.

[13] *Linux*. [Online]. Available: https://www.linux.org/ (visited on Jun. 27, 2021).

[14] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, *et al.*, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the conference of the ACM special interest group on data communication*, 2017, pp. 183–196.

[15] L. Qi, Z. Qiao, A. Zhang, H. Qi, W. Ren, X. Di, and R. Wang, *Performance Analysis of QUIC-UDP Protocol Under High Load*, English, ser. Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST. 2020, vol. 331, pp. 69–77. [Online]. Available: www.scopus.com.

[16] J. Postel *et al.*, "Transmission control protocol," 1981.

[17] J. Postel *et al.*, "User datagram protocol," 1980.

[18] A. CoNe, R. und Telematik, and C. Schindelhauer, "Internet control message protocol," 2008.

[19] Valve, *Counter-strike: Global offensive*. [Online]. Available: https://blog.counter-strike.net/ (visited on Jun. 27, 2021).

[20] *YouTube*. [Online]. Available: https://www.youtube.com/ (visited on Jun. 27, 2021).

[21] Ubisoft, *Trackmania*. [Online]. Available: https://www.ubisoft.com/en-us/game/trackmania/trackmania (visited on Jun. 27, 2021).

[22] Google, *Google drive*. [Online]. Available: https://drive.google.com/ (visited on Jun. 27, 2021).

APPENDIX

A. CAPTURE STATISTICS

In this section, three tables containing general data and statistics from the packet captures are presented: **Table 2**, **Table 3** and **Table 4**.

A short description of each capture:

- **Casual browsing (1)**: During this capture, some casual browsing was performed. In this case, it means that articles were visited and some online shopping was done.
- **Casual browsing (2)**: This capture was performed in a similar setting to Casual browsing (1).
- **Game (Counter-Strike)**: During this capture, the game *Counter-Strike: Global Offensive* [19] was played, this is a highly competitive game. During this game, it is crucial to have a low latency, because it is important to be able to act upon movement quickly. Having a high latency gives the opposing team a significant advantage.
- **Streaming YouTube**: This capture contains data from watching *YouTube* [20], which is a streaming platform where anyone can upload their content and - unless the video is protected - anyone can view this content.
- **Game (Trackmania)**: This capture contains data from playing the video game called *Trackmania* [21]. This is a racing game that is time-based, this essentially means that everyone finishes the track on their own and the time taken is compared. This setup means that latency is less important, as a small delay will not influence the results.
- **Download (Counter-Strike)**: During this capture, the download of the aforementioned video game *Counter-Strike: Global Offensive* took place.
- **Drive Download (QUIC)**: During this capture, a file of approximately five gigabytes was downloaded from the web interface from Google Drive [22]. This specific capture had the chrome flag "Experimental QUIC protocol" [14] turned on.
- **Drive Download (TCP)**: During this capture, a file of approximately five gigabytes was downloaded from the web interface from Google Drive [22]. This specific capture had the chrome flag "Experimental QUIC protocol" [14] turned off and therefore uses the regular Transmission Control Protocol (TCP) [16].
- **Drive Upload (QUIC)**: During this capture, a file of approximately five gigabytes was uploaded to Google Drive [22], using the web-interface. This specific capture had the chrome flag "Experimental QUIC protocol" [14] turned on.
- **Drive Upload (TCP)**: During this capture, a file of approximately five gigabytes was uploaded to Google Drive [22], using the web-interface. This specific capture had the chrome flag "Experimental QUIC protocol" [14] turned off and therefore uses the regular Transmission Control Protocol (TCP) [16].

Capture name	No. of packets	Capture time (s)	Total size (bytes)	Avg pps	Avg packet size	Avg Bps	TCP Errors
Casual browsing (1)	283,511.00	1,914.57	268,654,000.00	148.08	947.60	140,321.17	1799
Casual browsing (2)	90,297.00	1,364.67	54,713,107.00	66.17	605.92	40,092.67	1302
Game (Counter-Strike)	161,736.00	1,162.43	76,935,092.00	139.14	475.68	66,184.65	842
Streaming YouTube	508,927.00	1,800.39	612,206,536.00	282.68	1,202.94	340,041.82	651
Game (Trackmania)	290,877.00	1,398.07	59,172,589.00	208.06	203.43	42,324.63	692
Download (Counter-Strike)	265,641.00	107.23	282,479,611.00	2,477.42	1,063.39	2,634,456.62	5769
Drive Download (QUIC)	4,029,126.00	736.70	4,988,564,487.00	5,469.13	1,238.13	6,771,473.02	253
Drive Download (TCP)	3,490,767.00	667.62	4,900,986,808.00	5,228.68	1,403.99	7,340,993.60	369
Drive Upload (QUIC)	4,069,280.00	767.62	5,007,141,946.00	5,301.19	1,230.47	6,522,977.56	166
Drive Upload (TCP)	4,905,406.00	715.70	4,984,263,390.00	6,854.03	1,016.08	6,964,208.86	8974

Table 2. Statistics extracted from the captures.

Capture name	Protocol					
	UDP				TCP	Other
	Total	QUIC	Data	Other		
Casual browsing (1)	74.9%	13.2%	56.0%	5.7%	25.0%	0.1%
Casual browsing (2)	55.8%	15.3%	39.4%	1.1%	44.2%	0.0%
Game (Counter-Strike)	90.8%	0.1%	90.3%	0.4%	9.2%	0.0%
Streaming YouTube	96.9%	96.8%	0.0%	0.1%	3.1%	0.0%
Game (Trackmania)	95.1%	0.1%	94.9%	0.1%	4.9%	0.0%
Download (Counter-Strike)	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%
Drive Download (QUIC)	100.0%	99.9%	0.0%	0.1%	0.0%	0.0%
Drive Download (TCP)	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%
Drive Upload (QUIC)	100.0%	0.0%	100.0%	0.0%	0.0%	0.0%
Drive Upload (TCP)	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%

Table 3. Protocol division in the captures.

Capture name	Packet length									
	0-19	20-39	40-79	80-159	160-319	320-639	640-1279	1280-2559	2560-5119	5120 and greater
Casual browsing (1)	0.00%	0.00%	9.76%	8.93%	4.30%	0.93%	50.45%	25.63%	0.00%	0.00%
Casual browsing (2)	0.00%	0.00%	24.95%	9.62%	29.28%	2.20%	4.21%	29.74%	0.00%	0.00%
Game (Counter-Strike)	0.00%	0.00%	3.39%	22.19%	26.57%	9.59%	32.02%	6.25%	0.00%	0.00%
Streaming YouTube	0.00%	0.00%	12.31%	1.35%	0.55%	0.24%	0.40%	85.14%	0.00%	0.00%
Game (Trackmania)	0.00%	0.00%	4.66%	60.83%	12.71%	20.11%	0.22%	1.46%	0.00%	0.00%
Download (Counter-Strike)	0.00%	0.00%	29.86%	0.23%	0.46%	0.45%	0.19%	68.81%	0.00%	0.00%
Drive Download (QUIC)	0.00%	0.00%	10.36%	0.79%	0.02%	0.18%	1.34%	87.31%	0.00%	0.00%
Drive Download (TCP)	0.00%	0.00%	5.52%	0.07%	0.00%	0.00%	0.00%	94.40%	0.00%	0.00%
Drive Upload (QUIC)	0.00%	0.00%	11.98%	0.07%	0.00%	0.04%	0.27%	87.63%	0.00%	0.00%
Drive Upload (TCP)	0.00%	0.00%	28.82%	0.96%	0.00%	0.00%	5.86%	64.36%	0.00%	0.00%

Table 4. Packet lengths division in the captures.