Understanding Backhaul Characteristics

Armando Flores and Aleksandar Yonchev

I. Introduction

We live in a time where most of the tasks that we perform are digitized. We most probably have a laptop or a tablet which we use for work. We also have a digital device, a smartphone, in our pocket which we use almost all of the time for communication, entertainment , etc. We even have smartwatches and all sorts of other devices that make us dive even deeper into the digital world. In order for the digital world to continue its existence one very important function of it needs to keep running smoothly and flawlessly - communication. In the last decades the number of digital devices has grown exponentially. With it the communication between these devices, as well as the amount of data that we send. Not only that, but also the security risks have increased tremendously. The world is full of fraudsters who take advantage of the existing vulnerabilities and try to steal our data. One of the most important features of the digital world that we require nowadays is secure communication. For that reason the notion of a Virtual Private Network (VPN) came into existence, which allows us to communicate securely even when we are connected to a non-trustworthy network. In order to achieve this most VPN protocols use encryption and establish a secure communication tunnel between the communicating parties. Some of the most popular examples of a VPN protocol are IPSec, OpenVPN and WireGuard. The first two protocols are known to nearly every system administrator, because they have been the go-to solution for years. The latter is a relatively new and modern software, which has many advantages over its competitors. WireGuard has higher performance and stronger security. However, it's most important feature is its small codebase. IPSec and OpenVPN are protocols which consist of between 400,000 and 600,000 lines of code [1]. This is extremely hard to support and maintain even for large teams. On the other side, WireGuard is developed using only around 4,000 lines of code, which makes it much easier to maintain and extend. That's one of the reasons why it's also easier to setup and control. Furthermore, WireGuard is integrated into the Linux kernel, but is also cross-platform and runs on Windows, MacOS, BSD, iOS and Android [2].

II. Issue

We outlined how advantageous WireGuard is and why it should be the preferred VPN protocol. However, WireGuard misses some important features. First, the protocol lacks multipath support. Currently there is no mechanism that allows us to transmit data over multiple concurrent paths. Second, WireGuard doesn't provide monitoring on the path performance. Currently there's no way for us to assess the packet loss, the delay or the capacity of a path. Third, because of the fact that WireGuard is now part of the Linux kernel, the process of developing and testing extensions for WireGuard takes more time as with each code modification we need to recompile the whole kernel.

III. Related Work

A team at Fraunhofer FOKUS worked on solving the multipath support challenge. In order to solve it, they introduced an additional set of new headers, which provide statistical data about the path performance. These headers include the Path ID, Timestamp and Path Counter [3]. With this header data, the receiver side can calculate the relative path latency and report it back to the client alongside with the path counter and a redundancy counter. With the help of the statistical data, the sender can choose the most efficient path. The team has developed a path scheduler which supports multiple load-balancing algorithms like Round-Robin, Minimum Round-Trip Time and Multi-armed bandit algorithms.

IV. Project Goals

The purpose of the current project is to calculate some important characteristics of a communication path. These characteristics include Capacity, Packet loss, Delay and Jitter. As it was mentioned the development and testing of WireGuard extensions is tedious because of the recompilation needed of the Linux kernel. For that reason we took an alternative approach - develop a communication protocol between a sender and a receiver which supports multiple paths and calculates the above-mentioned characteristics. A key requirement is to use UDP as the underlying data transfer protocol. This way we are able to more closely simulate the behaviour of WireGuard as it also uses UDP to transfer data. The last important goal is to develop a path scheduler which takes the calculated characteristics into account and chooses the most efficient path for the next data transmission.

V. Design

The Backhaul characteristics describe essential parameters to be taken in consideration when defining qualitatively how a network performs, how data is transferred and in general how its behaviour takes advantage of its optimal working state. Therefore it becomes of main interest to study the real approach which backhaul characteristics such as the capacity, package loss, delay, and the jitter intrinsically formulate in the transferring process.

For a simplistic, holistic and well documented research, the Python programming language was chosen for the purposes of this project as it is flexible, rich in libraries and programming methodologies for the manipulation of network resources through crucial tools, such as the socket library. Additionally, Python includes a dynamic and easy to use

mathematics operations and also allows faster and palpable testing methods for the steps in development to be followed up and asserted. Python also represents a fast and easy to grasp learning curve that provides an extensive variety of examples, guides and explanation resources available to be put in practice and exploited as orientation for reaching broader possibilities of methods and steps to optimize the programming experience and the code efficiency.

Once the transmission protocol was distinguished as main choice to implement in the communication process between server and client sides, the UDP protocol facilitates direct and faster data transmission through sockets incorporation. Thanks to its simple methodology in which the information is consistently and continuously transmitted without regarding the acknowledgement of received and recognized packets through data sending, the interval process sinks although it becomes more uncertain. Nevertheless, here is where the package loss simplifies it's study for a optimization and minimization of it's measurement after analyzing and comparing it's absolute and relative increase or decrease through time. Reading this metrics this way allows to identify the proportion of packets received by the server, when from the client side an amount of data represented as a data file is split and destined to be dispatched step by step to the server. The measurement of the package loss was therefore an action to be conceived by comparing the progressively total amount of packets sent. For this the packets were decided to be identified by a enumerating sequence number, for which after each packet sending the iterative increment of sequence number would be expected regarding the progression of packets received. Then if one of the enumerating sequence number of a packet doesn't match with the next expecting number, a packet loss would be identified and reported in the program. The accumulation of packet loss would be accumulated to determine the whole missing packets as expected sequence number which were lost in the process.

The following function is responsible for calculating the package loss.

```
def check_missing_packet(
    expected_seq_num: int,
    actual_seq_num: int,
    missing_packets: List):

print("expected seq num, seq num",
(expected_seq_num, actual_seq_num))

if(expected_seq_num != actual_seq_num):
  if(actual_seq_num in missing_packets):
    missing_packets
      .remove(actual_seq_num)
  else:
    missing_packets
      .append(expected_seq_num)
```

udpServer.py

As mentioned, the function calculates the package loss based on the sequence number of the packets. Thus can the number of missing packets be counted and organized in a list, afterwards divided by the whole packets sent per file deriving in the package loss or proportion of missing packets.

Another aspect to be considered as indispensable factor of the network characteristics is the delay which is the key metric to determine the efficiency of the network transmission. By measuring the delay for each path we can take intelligent decisions and choose the best performing paths for the data transfer. Our initial approach for load balancing was to use the Round-Robin algorithm, which just distributes the load equally between the different paths. However, the Round-Robin algorithm isn't the most effective one. For that reason we decided to design a minimum-delay path algorithm. Here is an overview of how we achieve this.

A. Client

1) The client stores a list of the different paths in the `path_array` variable with their characteristics - **Path ID**, **Target**, **Delay**, **Packet slots**. There is also a `control_path` socket created, which is used for sending control messages.
2) The client sends the packets in batches (20 messages by default)
3) The client gets the best path by calling the `get_best_path(path_array)` function and uses this path while there are available packet slots and after that it takes the next available path.
4) After the batch of packets is transmitted the client sends a **BATCH_FIN** message through the `control_path` socket
5) The client waits for a message from the server, containing a list of the average delays for each path.
6) After it receives the delays, it goes through the `path_array` and updates the delays of the paths.

B. Server

1) The server calculates the delays for each path and stores them in the `path_delays` variable
2) After the server receives the **BATCH_FIN** message it calculates the average delay per path.
3) The server then sends the delays back to the client as a string
4) The server clears the `path_delays` variable, so it can store the delays in the next batch

Measuring the delay can different network path be compared for the decision taking of which path takes the shortest amount of time for transmitting packages. Since the delay is defined by the difference of time between the initial timing in which the package was sent from the client side and the final timespan in which the package is finally received by the server and hereafter acknowledged, two different metrics defined in both different client and servers sides must be put all together in one same single common place to be processed and compared.

Following this trajectory is evident that for the delay calculation the sending timing must be transmitted within the packets sending to be collected by the server. Here is where header inclusion of further metrics in the packets message

plays a fundamental role. A timestamp can be easily measured in the code by the use of the python time library, importing this one and implementing the time measurement as follows:

```
import time

time_stamp = int(time.time())
```

udpClient.py

From here is derived the final compacted programming instructions for obtaining and sending the timestamp from the client side when starting the transmission of a packet. Here is vital the use of the header concept, which represents a variable in which we include all the meta data to be transmitted to and received by the server allowing the meta data intercommunication between both client-server sides.

```
headers = struct.pack('bii',
path.id, time_stamp, sequence_number)

path.socket.sendto(headers
+ bytes(subframe, encoding='utf-8'),
path.target)
```

udpClient.py

Calculating the delay is fairly easy. We have taken some of the lines of code which constitute the procedure. What we are doing - we are taking the time when the packet was sent from the headers. Later we take the current time and we subtract the time of the transmission. This way we calculate the difference in milliseconds.

```
path_id, start_time, sequence_number =
struct.unpack('bii', headers)

end_time = int(time.time())

delay = end_time - start_time
```

udpServer.py

The delays are stored in the `path_delays` variable. After the server receives the **BATCH_FIN** message it calculates the average delay for each of the paths, using the *numpy* library. It stores the result as a string in a comma-separated values (CSV) format and reports it back to the client.

```
data_str = data.decode('utf-8').strip()

if(data_str == constants.BATCH_FIN_MSG):
result_array = []
for delay_arr in path_delays:
result_array.append(round(np.mean(delay_arr)))

statistics_message = ','.join(
  map(str, result_array)
).encode('utf-8')
reverse_path
  .sendto(statistics_message,
  constants.CLIENT_ADDRESS)

path_delays = initialize_path_delays()
continue
```

udpServer.py

Once the delays are received the client calls the `update_path_delays` function which goes through the available paths and updates their delays.

```
def update_path_delays(
    path_array: List[Path],
    delays: List[int]):
    for curr_path in path_array:
      curr_path.delay = delays[curr_path.id - 1]
```

udpClient.py

Then on the next batch transmission the function `distribute_packets(path_array, packets_count)` is executed. The primary purpose of this function is to distribute the load between the best-performing paths. In order to make it more understandable, code comments have been added which explain what the different lines do.

```
    def distribute_packets(
        path_array: List[Path],
        packets_count: int):
    # Sort the paths array by the delay
    path_array.sort(key=lambda x: x.delay)

    # Take the first item (the best path)
    best_path_delay = path_array[0].delay

    # Find the number of paths
    which have the same delay
    best_path_count =
      [path.delay for path in path_array]
        .count(best_path_delay)

    # Split the packets among the best paths
    packets_per_path =
      packets_count / best_path_count
```

```
for index in range(best_path_count):
    path_array[index].packet_slots =
      packets_per_path

# Assign the remaining packet slots
to the first path
path_array[0].packet_slots +=
  packets_count % best_path_count
```

udpClient.py

For the average calculation is the mean of the list of delays utilised. The mathematics definition describes following manners the mathematical formula of the mean value:

$$Mean = \frac{\sum_{i=1}^{N} i}{N}$$

With this is then the selection of the proper optimal path according to the minimal delay average conceived. This is since for choosing the fastest path, a historical data must be taken in considerations and compared for the best path to be chosen. After 50 initial attempts, will each path 10 times proved, so that right afterwards will from then on the path with the minimal calculated path average be preferred as optimal path to transmit data.

This algorithm relies strongly on the first 10 average data obtained for the different paths, since a first low mean means a preferable path to transmit information quicker.

Having once the delay, becomes obvious for next steps to measure the capacity of the network, which is defined by the total amount of bits that can be transmitted per second through the network. This way using the sys library will the total size of the packet be calculated as amount of bits, and with this is next the capacity calculated resulting from the division operation of the obtained size divided by the delay which defines the seconds required for the amount ob data to be sent. Therefore the programming instructions for the measurement of the capacity are as follows represented:

```
import sys

packet,addr = sock.recvfrom(1024)

size = sys.getsizeof(packet)
# get size of data

capacity = size / delay
```

udpServer.py

Finally but not least will the jitter metric be expected to be figured out. The jitter characteristic of the network represents the average deviation of delay within a collection of presented delays.

As the timestamp for each packet is sent and received by the server, they result essential for the conception of the jitter determination. Hereto are diverse delays compared to the mean output obtained as a result of accumulated time differences which define as follows the final jitter:

```
t1.append(start_time)
t2.append(end_time)

if count > 1:
for i in range(1,count):
  Dj.append(
    (t2[i]-t1[i]) - (t2[i-1]-t1[i-1])
  )
j = np.array(Dj)
jitter = j.mean()
```

udpServer.py

As a result we obtain the mean deviation of delays for each time a new packet with a new delay value is sent and added to the list of time differences.

This way the mathematical approach of average jitter is as follows defined:

$$Avgjitter = \frac{\sum_{i=1}^{N} |D|_i}{N}$$

Once recognized the formula for the average jitter it becomes clear that this characteristics indicates how correlated holds together the interval of sending timing through continuous iteration of packets sending in the network. This characteristic describes how consistent or volatile the network intrinsic state may behave through time and therefore presenting how well a network connection may secure the data transferring expected to be caught in a foreseen bounded average timespan.

Then with all the metrics deterministically conceived in and assessed in the server side, next steps that prosecute are the multipath algorithm sorting for which a predicted shorter route must stand out as most intelligent sorting for smartest process of data communication.

Once the communication was successfully established among the communicating devices was as next precised to study all different possible options for resolving the path choosing dilemma.

In first instance were five different possible roads set to analyze variations between the different paths and this way compare resulting backhaul characteristics observed in the receiving server side. This helped to delimit extremes of maximal and minimal values, as well as average volatility and diversion of the varying phisical characteristics. Some correlation were this way recognized and interpreted as the minimally varying delay between short periods of interval and greater oscillation between longer periods of time in which connection resulted in retarded protocol communication.

Jitter is therefore evidently affected and influenced as well as the capacity and depending on the network instability also the packet loss. Reason why after studying the backhaul characteristics behaving in time directs us to the conclusion that the path must be optimally be selected regarding the average measured metrics in time of the network. In other

words the behavior of the network must be contrasted in time and the average results must be collected, accumulated and regarded in time for deducting which path represents optimal route for the transmission of packets to arrive in the shortest predictable time to their destiny.

Considering that the optimal path may vary through time, since the network whole road may differ radically according to interconnected devices performances, overloading and saturation which they may experience, the different network paths must be continuously and consistently be judged and proved for a right on time proper reaction and fix of current path to be implemented and selected for the continuing of packets transferring. Here arise nevertheless the following questions: how should the algorithm be developed, so that the multiple paths are still evaluated for it's optimal performance timing recognition and immediate selection? Or how should the algorithm acknowledge regarding which metrics and characteristics a route is the ideal for sending and delivering the packets in time? Even the question of how and based on which metrics should the path selection be determined and be hold in time for it's maximal optimal benefit in terms of seconds savings?

These and further inquiries had to be pondered evaluating the priorities and desired and preferred measures to get optimized and interpreted according to given observations and results boarding these handed characteristics. A single simple but effective approach was then envisioned and applied to the program. Collecting all measures of delay and deriving it's mean value through time and storing these data for each applied path. These values had to be reallocated in the client side for its direct comparison. Therefore was decided that the data should be structured and packed for being sent through the bidirectional established connection for its further manipulation. Then once handed into the client side the average delay could be put in comparison for obtaining a trustworthy determiner of best current route with best timing performance.

Enforced approach facilitates the conception of a deterministic solution for the multipath selection, enhancing slowdown reduction in the communication process and augmenting delivery time overall.

VI. Conclusion

The study of backhaul characteristics is fundamental for a detailed analysis of network communication. Measuring different parameters of the system enables a broader understanding of the multiple possible actions that can be handled for improving the ultimate performance of the data transmission. Regarding the miscellaneous variables that constitute the telecommunication approach produces a spectrum of responses to be investigated for a better decision taking in the architecture and design of optimum selected routes in time. Allowing betimes reactions for handling alternating performance metrics bring security and coordination to the system, which can hereby be controlled and set for according data delivery.

Through the development of the algorithm and project components, a scheme proposal was boarded for the integral comprehension of the protocol setting and further socket configuration for the information in bytes to be processed and accordingly structured and recognized by the target destination where the package be due to be handed over.

Rhetorical assay of software integration contemplates a didactic and formative overview of considered elements to be retrospectively pondered for the envision and creation of intelligent path-sorting algorithms. Following this progression becomes evident, that the study of the backhaul characteristics opens doors in the scientific field for researches to recognize and formulate new patterns in the approach of data transmission and network communication through carefully regarded path selections. In this ambit new software integration generates advances and feasible methods of improving the nowadays delivery of packets through effective protocols as UDP.

References

[1] https://en.wikipedia.org/wiki/WireGuard. Accessed: 21.08.2022.
[2] https://www.wireguard.com/. Accessed: 21.08.2022.
[3] Konrad-Felix Krentz and Marius-Iulian Corici. Poster: Multipath extensions for wireguard. In *2021 IFIP Networking Conference (IFIP Networking)*, pages 1–3, 2021.