Assessment of the performance of a special User Datagram Protocol
Identifying Packet Loss and Reordering Packets in Keyed UDP Transmissions

Fábio Gil Machado

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador: Prof. Doutor Nuno Manuel Garcia dos Santos

Outubro de 2022

Declaração de Integridade

Eu, Fábio Gil Machado, que abaixo assino, estudante com o número de inscrição M7061 de/o Engenharia Informática da Faculdade Engenharia, declaro ter desenvolvido o presente trabalho e elaborado o presente texto em total consonância com o **Código de Integridades da Universidade da Beira Interior**.

Mais concretamente afirmo não ter incorrido em qualquer das variedades de Fraude Académica, e que aqui declaro conhecer, que em particular atendi à exigida referenciação de frases, extratos, imagens e outras formas de trabalho intelectual, e assumindo assim na íntegra as responsabilidades da autoria.

Universidade da Beira Interior, Covilhã 10 /10 /2022

Assinado por: **Fábio Gil Machado**
Num. de Identificação: 14078610
Data: 2022.10.10 17:08:59+01'00'

Fábio Gil Machado

Acknowledgements

I would like to express my gratitude to my friend, professor and supervisor, Prof. Nuno M. Garcia for his guidance, useful critiques and his high expertise in the formulation of this dissertation.

A special thank you to Prof. Mário Freire, for giving me the opportunity to re enrol in this course and resume my studies.

In addition, I am grateful for my parents and the opportunities they have provided during my life.

Finally and most importantly, I wish to thank my loving girlfriend and my daughter, for all their encouragement, support and especially patience throughout my work and studies.

Resumo

O User Datagram Protocol (UDP) e outros protocolos semelhantes enviam dados de aplicativos da máquina de origem para a máquina de destino dentro de datagramas ou pacotes, sem qualquer controle sobre a transmissão ou métricas de sucesso. Esses protocolos são muito convenientes para transmissão em tempo real porque a inexistência de mecanismos complexos de confirmação torna a transmissão de dados muito rápida. Em oposição, para suportar a funcionalidade e os recursos aprimorados de um protocolo orientado à conexão, um conjunto de mecanismos é implementado com base em alguns campos específicos do cabeçalho da unidade protocolar de dados. Esses mecanismos resultam numa sobrecarga significativa em termos de aumento do número de pacotes transmitidos, e isso pode traduzir-se em atrasos significativos, devido ao número adicional de tarefas de comutação, roteamento e, eventualmente, devido a procedimentos de comunicação mais complexos, como por exemplo, redimensionar a janela de transmissão, e claro, atualizar os números de confirmação e a sequência. Os dois extremos dessas modalidades de comunicação, um que não tem controle e outro que permite controle total, resultaram na criação de um protocolo intermediário que permite um grau limitado de conhecimento sobre o sucesso de uma transmissão, e até mesmo para uma eventual reordenação de pacotes que chegam fora de sequência. Esta dissertação apresenta resultados de simulação que confirmam a eficiência do novo protocolo UDP quase confiável, chamado Keyed User Datagram Protocol (ou KUDP) para transmissão de dados que inclui a capacidade de identificar quais pacotes foram perdidos e reordenar os pacotes que foram recebidos na ordem errada apontando tarefas futuras a serem desenvolvidas nesta pesquisa.

Palavras-chave

Algoritmo de reconstrução de cadeias de dados; Keyed User Datagram Protocol; protocolo quase confiável; simulação de transmissão de dados

Abstract

The User Datagram Protocol (UDP) and other similar protocols send application data from the source to the destination machine inside datagrams, without any type of control on the transmission or success metrics. These protocols are very convenient for real time transmission as the absence of complex control mechanisms tend to make the data transmission adequately fast. In opposition, to sustain the increased functionality and features of the connection-oriented protocol, a set of mechanisms is implemented based on some specific fields of the segment header. These mechanisms result in a significant overhead in terms of the increased number of transmitted packets, which in turn may translate into significant delays, because of the additional number of switching and routing tasks, and eventually, because of more complex communications procedures, such as e.g. transmission window resizing, and of course, acknowledgement and sequence numbers updating. The two extremes of these communication modalities, one that has no control at all, and the other one that allows for full control, have resulted in the creation of an intermediate protocol that allows for a limited degree of knowledge on how successful a transmission was, and even for an eventual reordering of the packets that arrive out of sequence. This dissertation presents simulation results that confirm the efficiency of the new almost-reliable UDP protocol, named Keyed User Datagram Protocol (or KUDP) for transmission of data that includes the ability to identify which packets were lost and to reorder packets that were received out-of-order, and points future tasks to be pursued in this research.

Keywords

Algorithm for Stream Reconstruction; Keyed User Datagram Protocol; almost reliable protocol; data transmission simulation

Table of contents

List of figures

List of tables

Acronyms

KUDP	Keyed User Datagram Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
SYN	Synchronization
ACK	Acknowledgement
RFC	Request for Comments
HTTPS	Hyper Text Transfer Protocol Secure
HTTP	Hypertext Transfer Protocol
FTP	File Transfer Protocol
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
DHCP	Dynamic Host Configuration Protocol
TFTP	Trivial File Transfer Protocol
VoIP	Voice Over Internet Protocol
SRA	Stream Reconstruction Algorithm

Chapter 1

1. Introduction

The User Datagram Protocol (UDP) protocol is used around the world every day, to support connectionless communications for example for real time transmission. It is interesting to understand a little more how the two most used protocols work, TCP and UDP. UDP doesn't have any kind of control over the transmitted data and the destination host presents to the user the data exactly by the sequence that was received and even if some packets are lost the destination has no manner to assess that, so the destination application handles the data in a totally agnostic manner regarding the sequence it was transmitted. On the opposite side of this data transmission set of features, we have the Transport Control Protocol (TCP), with which the transmission of the different segments of data is totally controlled and confirmed. This allows for extremely reliable data transmissions, by means of a complex acknowledgment mechanism, and with the cost of much slower data transmission rates. The new Keyed-User Datagram Protocol (KUDP) [1] proposes a compromise between these two extremes, being termed an almost reliable protocol. This protocol has incorporated one algorithm that is only executed in the destination host, no acknowledgements are sent, and it allows some capacity to detect losses and to reorder packets that were received out-of-order. This dissertation presents the Stream Reconstruction Algorithm that is responsible for the features that power the almost reliable KUDP data transmission protocol. It also presents the initial simulation results regarding the operation of KUDP.

1.1. Objective

The main goal of this work is the study of the proposed stream reconstruction algorithm and understanding if it’s possible its usage in a real environment. This proposed algorithm will be capable of detecting packet loss and packets that are received out-of-order, but it is necessary to run some test simulations to calculate its effectiveness. The test results will be presented in this dissertation to make these public and motivate more research by the technologic community.

1.2. Motivation

On February 7, 2019, a paper was published with a proposal for a new protocol, the Keyed User Datagram Protocol, Concepts and Operation of an Almost Reliable Connectionless Transport Protocol [1]. I am one of the authors of this article and I will explain what the

KUDP is. After the submission of this article many questions were raised, such as the detection capability of the proposed algorithm. Who knows, in the near future, the proposed protocol can be used in a real environment.

1.3. Contributions

This work focuses on the study, analysis, implementation and test of a stream reconstruction algorithm for the Keyed User Datagram Protocol to enable lost and out-of-order packets detection. The main contribution of this work is to evaluate the proposed algorithm and make some conclusions.

Two papers have been published, the first one, describing the mechanisms for the new Keyed UDP protocol, published in IEEE Access [1], the second one, published in IEEE Globecom 2020 [2], containing the description of the Stream Reconstruction Mechanism and the KUDP simulation results.

1.4. Organisation

This dissertation is organised and structured in five chapters that demonstrate the process used to accomplish the work's objectives. The chapters are the following:

- **Chapter 1:** consists of a general introduction of the dissertation theme presenting the objectives, motivations and contributions of the for the presented evaluations and results;
- **Chapter 2:** consists in the state of the art looking for the OSI model and TCP/IP model with focus on the transport layer (layer 4) analysing the definition, work and applications of the TCP and UDP Protocol;
- **Chapter 3:** consists in a brief overview of Keyed User Datagram Protocol explaining the concept and the usage of them;
- **Chapter 4:** consists in a Stream Reconstruction Algorithm definition and in all steps to reproduce the evaluations and tests of the proposed algorithm for detecting lost and out-of-order packets;
- **Chapter 5:** presents the conclusions of this study and proposals for future work.

Chapter 2

2.State of the art

In the world of technology nothing is right, nothing is wrong, because there are a lot of ways to do the same thing. A long time ago, we had no rules, each one was able to create typologies and different protocols from each other. These creations and inventions were not compatible with the creations and inventions done by others so that's quickly understood that isn't the best approach.

In 1970, the OSI model was created, later standardised as the ISO (International Organisation for Standardisation) reference model in 1983, with the main objective of being the standard model for all communications between different devices, allowing end-to-end communication.

The OSI Model is governed by seven layers, each one with a specific function:

- Application (Layer 7)
- Presentation (Layer 6)
- Session (Layer 5)
- Transport (Layer 4)
- Network (Layer 3)
- Data (Layer 2)
- Physical (Layer 1)

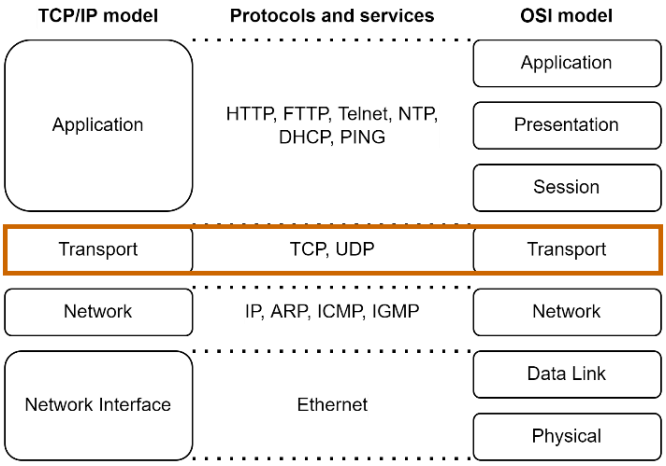

Figure 1 - TCP/IP Model vs OSI Model.

In this dissertation the focus is on Layer 4 (Transport) because that is where the TCP and UDP protocols apply.

The transport layer is responsible for enabling the temporary communication session between two applications and by data transmission between them. This layer is responsible for tracking the individual communication between the source and the target and managing the segmentation data to be transmitted to the target application.

TCP and UDP are internet protocols that determine how data is shared or transmitted over the internet. They have different characteristics, but they are used for the same purpose, sending packets to a remote IP, on the internet or on the local network. Both have advantages and disadvantages and would be used on a case-by-case basis.

2.1. TCP Protocol

2.1.1. Overview

TCP, described in RFC 793 [3], is the most used protocol, as it is a connection-oriented protocol and guarantees the delivery of packets between a sender and a receiver. When a sender and receiver initiate communication, they establish a connection before sending some data, there is a "pre-agreement" named Three-Way Handshake (SYN, SYN-ACK, ACK).

This one is considered the most reliable delivery protocol as when some data is corrupted or lost it can retransmit that specific data. This process of sending and receiving packets occurs whenever you perform an action on the internet that uses the TCP protocol, for example, opening a website, sending a message and others operations. It adopts a delivery system that enumerates all packets and sends them in order and when one of these packets is not sent correctly, the receiver sends a message to a sender to resend this packet and soon after receiving this packet is able to receive the next one until the last. They could sort and reconstruct data given by segment identification numbers and sequences.

Flow control is used to regulate the amount of data transmitted to be more efficient and adjust flows regardless of available throughput.

The most important feature of this protocol is error checking, to ensure that all information sent is not corrupted at the destination. This verification and the TCP sending process itself make it a very reliable protocol that is widely used by everyone.

2.1.2. Three-Way Handshake

How this “pre-agreement” called three-way handshake works, for example between a client and a server:

- The session between the client and the server is always initiated by the client, sending a connection request through a packet with the SYN flag enabled.
- Client sends to a random sequential number in this packet.
- Server responds using a SYN, ACK packet with its own random sequence number and acknowledgment number (same as client number +1, e.g., client was sending number 250 and server responds with number 251).
- To end the synchronisation, the client responds with an ACK packet with the acknowledgment number (equal to the server number +1).

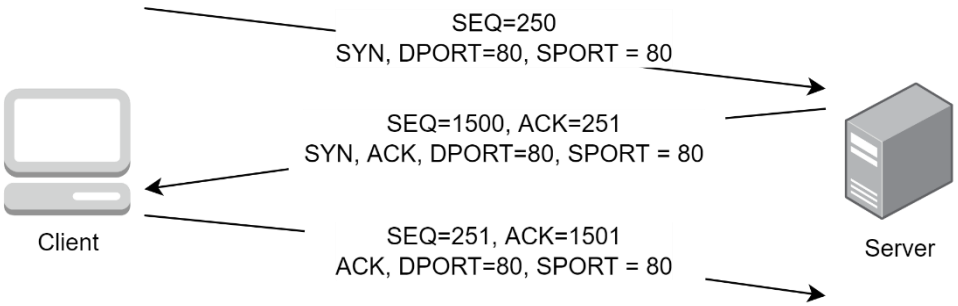

Figure 2 - TCP Three-Way Handshake.

Client: Server, are you there? (SYN)

Server: Yes, I am (ACK) and you, are you there? (SYN)

Client: Yes, I am (ACK)

2.1.3. Reliability

Reliability is one of the advantages when users need to ensure delivery of packages that cannot be lost, and ordering is important. This data, if not in the correct order, is unusable and unreadable.

All packets are marked with a number and a sequence to be able to reorder at the destination. You can use different routes for delivering data to the destination, based on different network paths, and depending on the best route for delivery. The best route may be different during connectivity.

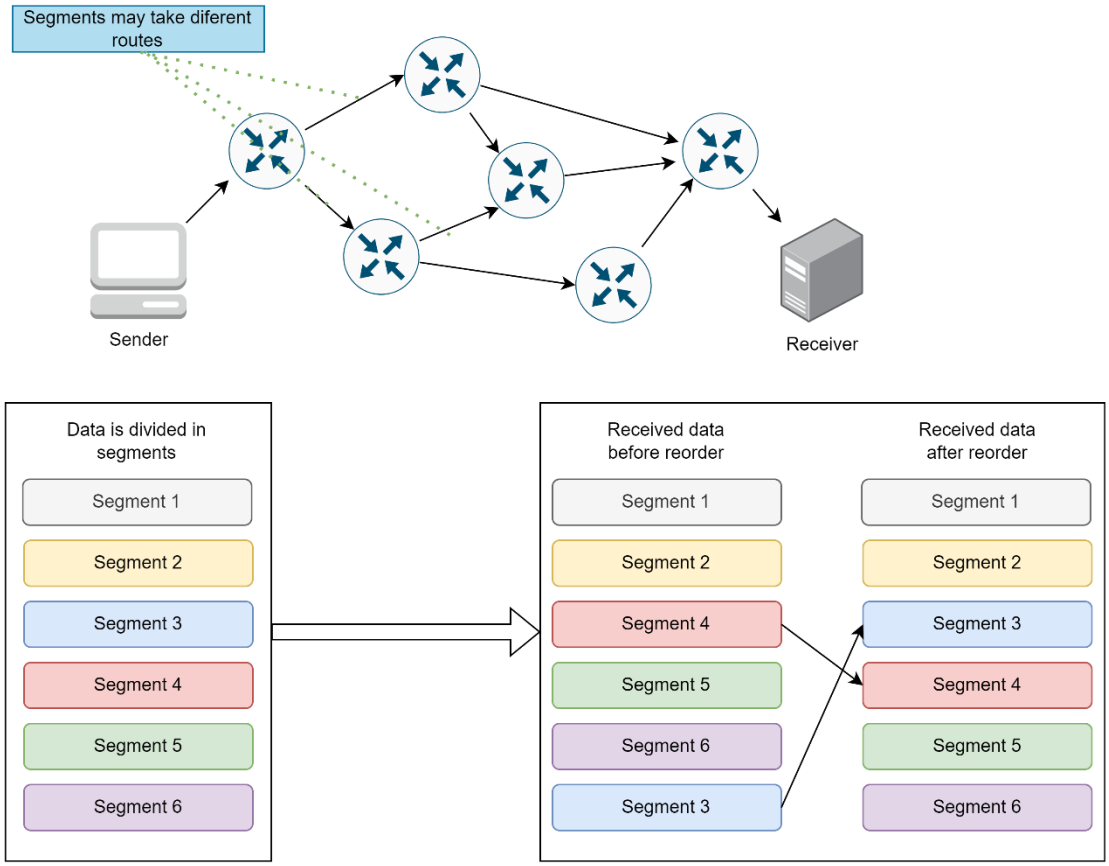

Figure 3 - TCP ordered delivery.

2.1.4. Acknowledgement and Window Size

The window size is the limit on how many bytes the sender can send without an acknowledgment and is used to avoid congestion at the receiver. The receiver informs the sender of its window size (buffer size). On the sender's side, the sender tries to make sure that at any time it does not have more bytes in transit than the received one informed it to send. The sender will wait for the acknowledgment message to transit more bytes and continue data transmission. This process repeats until all transmission packets are complete.

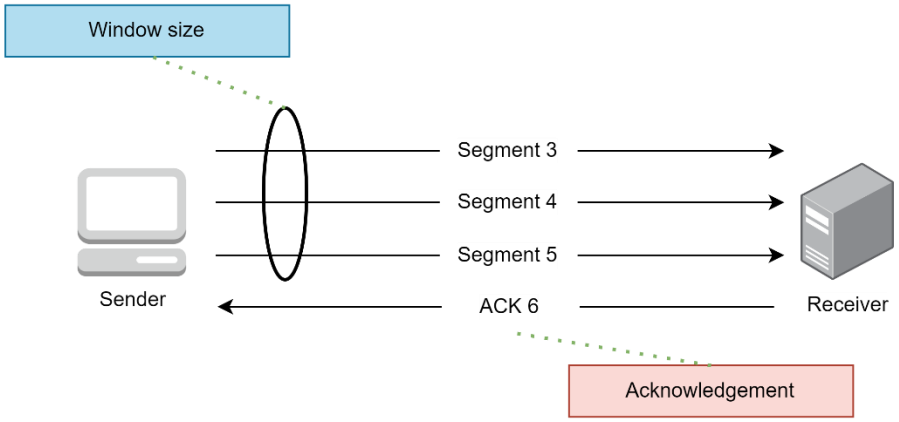

Figure 4 - Acknowledgement and Window Size.

Many applications use this protocol when they need to ensure that all data is transmitted, for example: HTTPS, HTTP, FTP, Telnet, SMTP, etc...

2.2. UDP Protocol

2.2.1. Overview

The UDP protocol, described in RFC 768 [4], is a connectionless protocol, but it allows users to establish communication between a client and a server and allows the transmission of data (packets) between them. In terms of functionalities, such as TCP, this protocol is used to transmit data between two applications, but it has differences.

This protocol does not retransmit lost or corrupted data and if the destination does not have all the packets the sender will not send the data again. It has no order data reconstruction, no flow control and is a stateless protocol.

When a user needs to communicate with another user, for example, making a call just with audio or with audio and video, there is an important premise, to establish this communication in real time. In this case, it is impossible to use the TCP protocol, because if there is a drop packet during the communication, the packet will be resent and the communication and transmission, for example, in the call with audio and video, will be not in real time.

UDP protocol enables communication between hosts and there is no packet checking to guarantee real time.

2.2.2. Connectionless and Unreliable

The advantage of the UDP protocol is the connectionless and unreliable transmission to be used in case of real-time communications when lost datagrams or data received out of order does not matter.

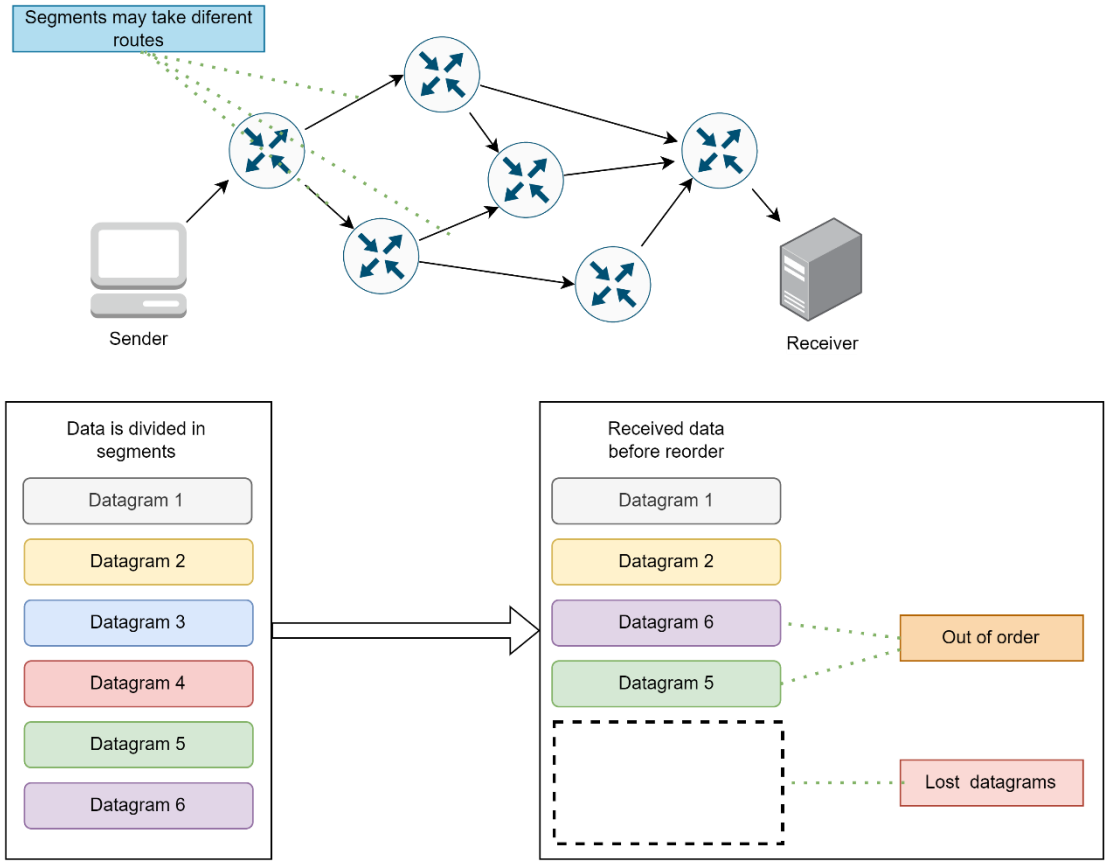

Figure 5 - UDP, connectionless and unreliable.

Many applications use this protocol when they need to transmit information, but don't care if the receiver receives it or not, for example: SNMP, DHCP, TFTP, VoIP, etc...

Chapter 3

3. KUDP Protocol

The KUPD protocol [1] proposed a compromise between the connectionless UDP and the connection-oriented protocol TCP, without any changes in the UDP header structure. Instead, it uses the port numbers in an innovative way. This chapter describes the workings of the KUDP protocol.

3.1. Overview

Keyed User Datagram Protocol is an extended UDP protocol that applies Layer 4 of the OSI model and uses standard UDP datagrams. When using the UDP protocol, we select the port to use for sending datagrams and sent from the IP address of the source using a pre-configured port 9000 (for example).

Considering the UDP protocol, KUDP was designed to reduce transmission overhead and use different ports to send datagrams and not just between a source port and a destination port.

The proposal of the KUDP Protocol is to send datagrams through multiple destination ports, reserved and opened by the destination host, defined in the configuration parameters of the specific application software to receive datagrams. KUDP protocols can use three approaches:

- One sends port and multiple receive ports

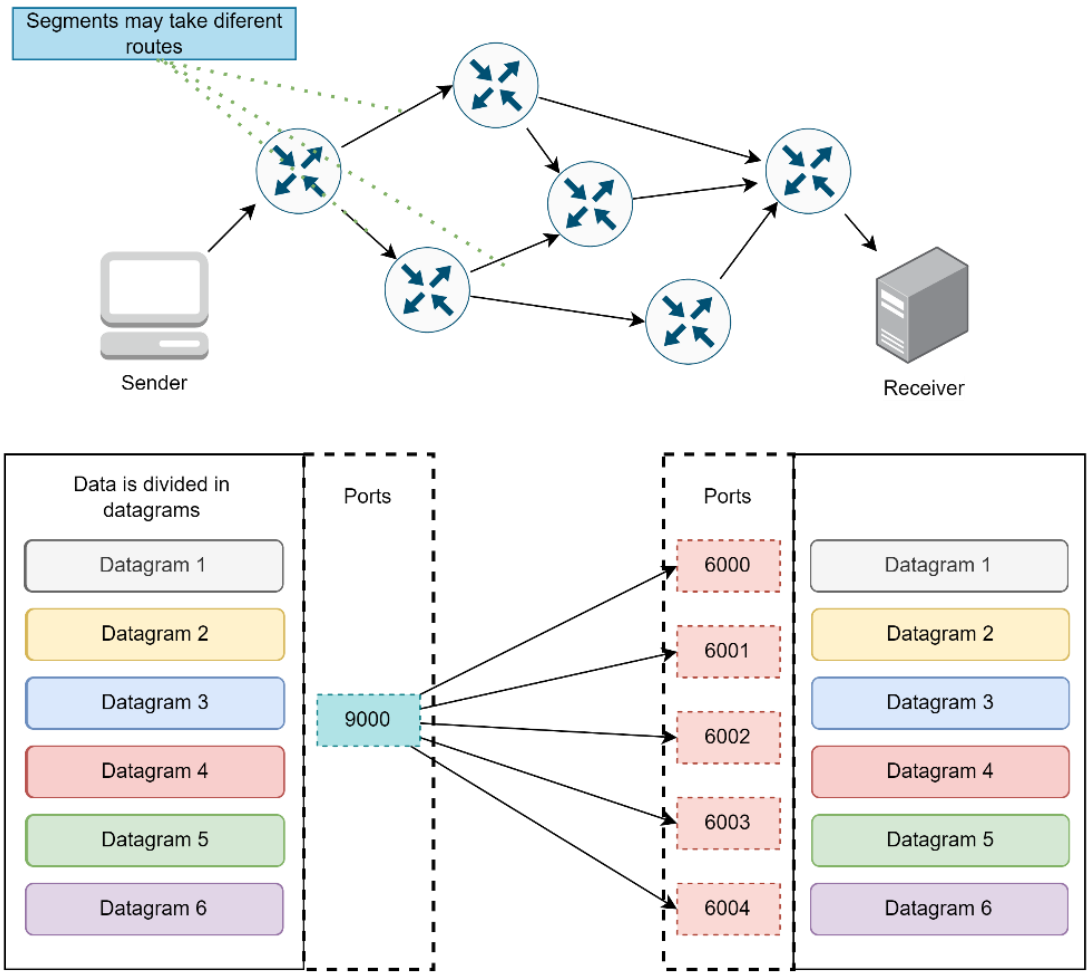

Figure 6 - KUDP, using one sending port and multiple receiving ports.

- Multiple send ports and one receive port

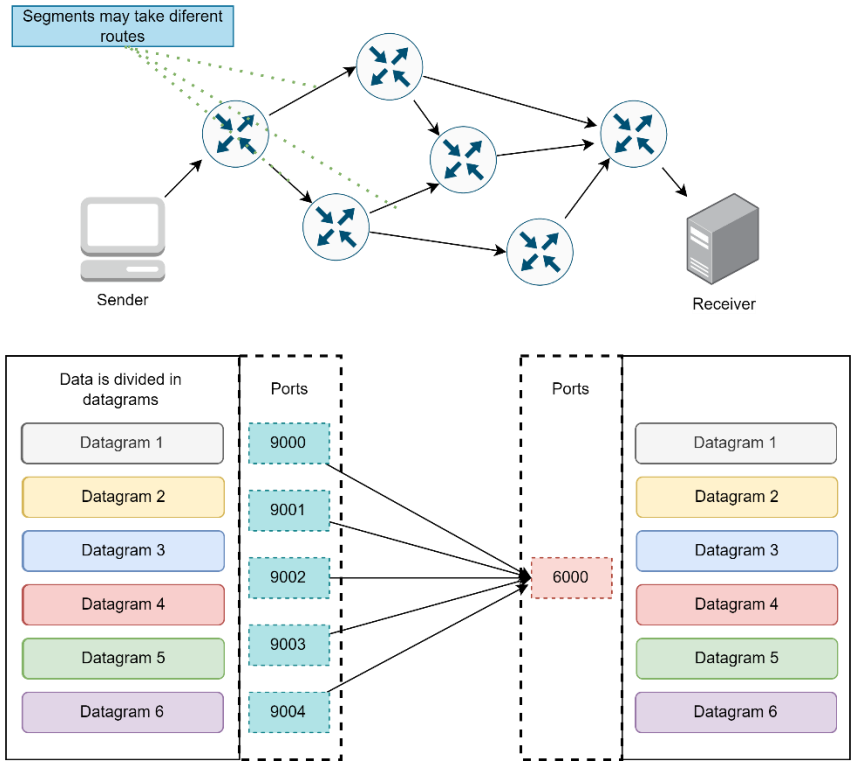

Figure 7 - KUDP, using multiple sending ports and one receiving port.

- Multiple send ports and multiple receive ports

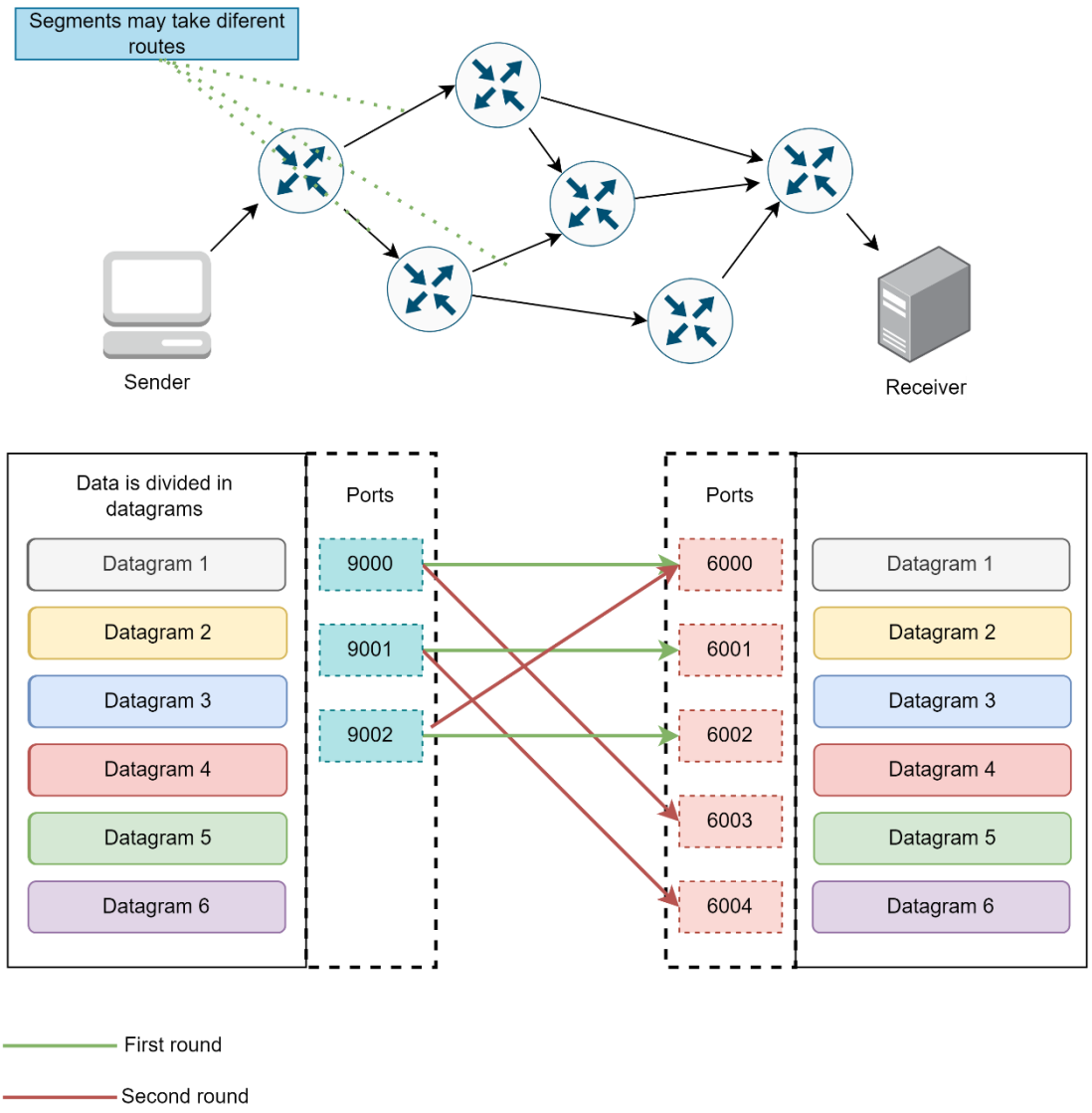

Figure 8 - KUDP, using multiple sending ports and multiple receiving ports.

The idea is to use multiple ports to create a sending sequence that allows the destination to identify if any datagrams are lost or out of order.

How the KUDP protocol works:

1. If the sender has the following pre-configured ports for sending datagrams: 9000; 9001; 9002; 9003; 9004
2. If the receiver has the following ports pre-configured to receive datagrams: 6000
3. The sender will always use the ports to create a sequence: 9000;9001; 9002; 9003; 9004
4. The receiver knows the sender's ports when it receives the datagrams and can identify lost and out-of-order datagrams.

5. After all sequence ports are used, the KUDP protocol repeats the process, reusing all sequence ports again.

Assuming possible UDP combinations, counting source ports used when transmit data is n, when count with destination port is n+1.

Using KUDP counting possible combinations counting source ports (s) and destination ports (d), the possible combinations are s+d = n.

3.2.KUDP Research needs

One of the research needs is to give the KUDP protocol the ability to detect lost and out-of-order datagrams.

To enable this capability, the proposal is to create an additional sublayer in the TCP/IP transport layer and use the KUDP Protocol with this sublayer when the Flow Reconstruction Algorithm is needed to ensure the reliability of this protocol.

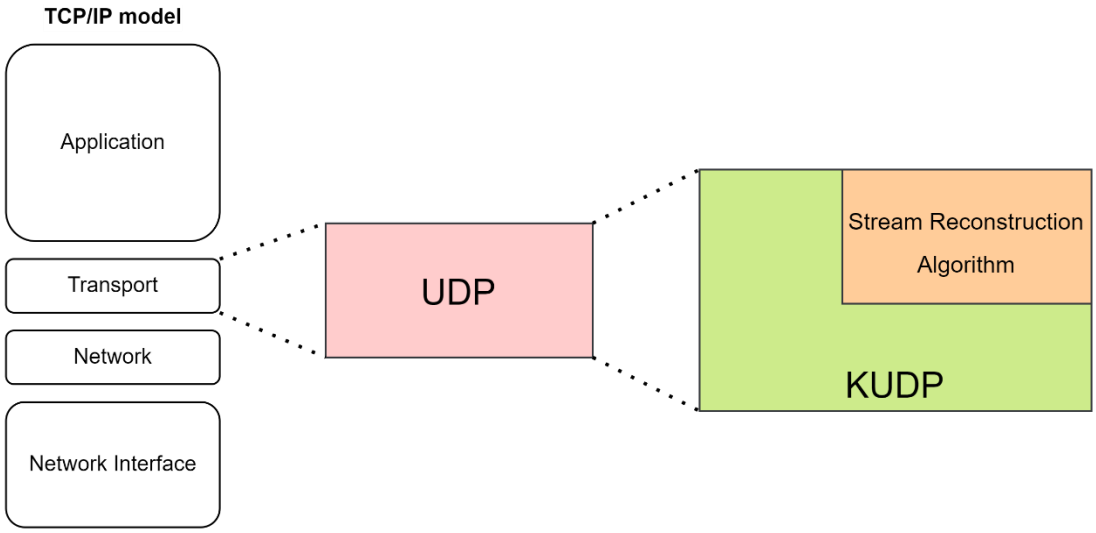

Figure 9 - KUDP sublayer map.

Chapter 4

4.Stream Reconstruction Algorithm

4.1. Definition

The stream reconstruction algorithm (SRA) is an additional sublayer that allows the KUDP protocol to detect lost and out-of-order datagrams. This sublayer requires the implementation of an additional algorithm that will be executed at the receiver, without any recognition, control or success metrics provided by the receiver to the sender. To include more datagram headers, we will increase transmission overhead, so this is not the best practice when we talk about a UDP based protocol, as this protocol is used for real-time transmission and the main requirement is efficiency.

In this case, I will only use one port to send datagrams and ten ports to receive (one sends port, and several receive ports, as shown in Figure X). I will simulate the lost and out-of-order datagrams and identify the datagrams with a number (key of the port that receives the datagram) and with a letter (to identify the sequence), for example 1a, 2a, …, 5a, 1b, 2b, … 5b, …, 5d for better understanding of the algorithm. In the following simulation we will use 20 datagrams and only show the interaction of the first 10.

For this example, we have the following transmitted (Tx) and received (Rx) datagrams:

Tx = {1a, 2a, 3a, 4a, 5a, 1b, 2b, 3b, 4b, 5b, 1c, 2c, 3c, 4c, 5c, 1d, 2d, 3c, 4c, 5c}

Rx = {1a, 2a, 4a, 1b, 5a, 2b, 3b, 5b, 2c, 1c, 3c, 5c, 1d, 3d, 4d, 5d}

Four datagrams are lost (3a, 4b, 4c, 2d) and there are out-of-order datagrams 1b with 5a and 2c with 1c.

With a key port 5 (n) and running each iteration with n-1 we can get amazing results and most datagrams can be identified as lost or if they are out of order the algorithm can identify and reorder these datagrams and put them in the correct order.

We can identify some points in the simulation using the presented algorithm:

At the point identified by 1 all occurrences are 2a then the algorithm will assume this datagram.

The point identified by (2) the datagram 3b has more occurrences and the algorithm will assume this, as it appears more times than 3c.

At the point identified by (3) for the key port 4 there are no datagrams to place here, so the algorithm will assume it is a lost datagram.

f identifies all datagrams that the algorithm does not detect, and may reflect different reasons, the datagram was actually lost or in the n-1 interaction it could be detected, and the datagram exists.

For this proposal of a Flow Reconstruction Algorithm there are many tests and proofs to be done to guarantee the feasibility of its use in the real concept.

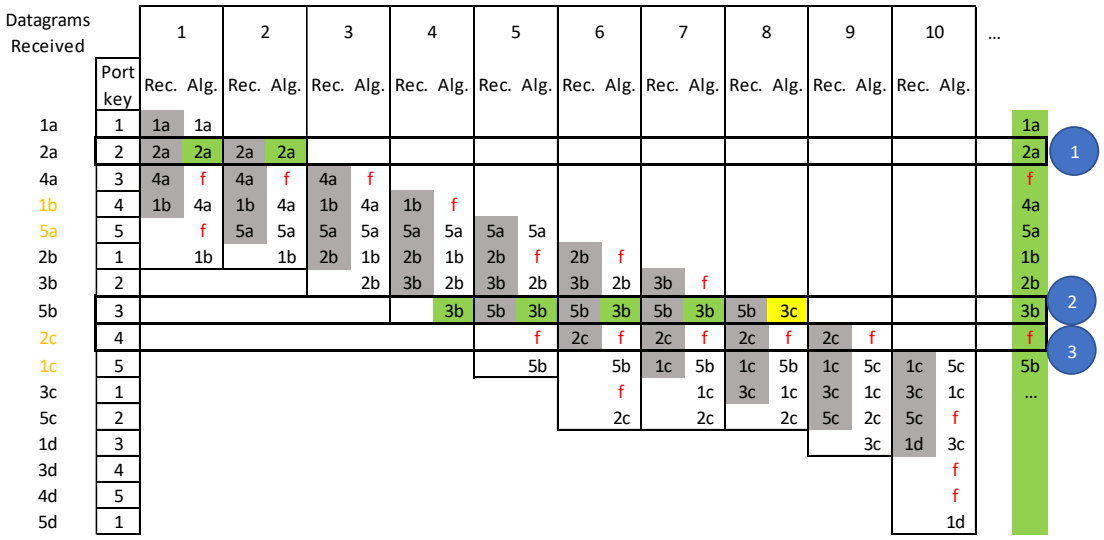

Figure 10 - KUDP Algorithm interactions.

4.2. Evaluations and tests

After the definitions, it's time to run tests and get concrete results on the use of the proposed algorithm.

For this chapter, the Ruby language was used to implement the algorithm and create some tests to ensure the feasibility of the proposed algorithm as a sublayer for the KUDP protocol.

4.2.1. Prepare tests

The first step was the installation of RubyMine IDE (ruby's IDE) to assist in the implementation and design the code to simulate the use of the algorithm and its implementation, and this can be installed using the student licence provided by JetBrains for UBI students.

After that, the code for the simulation has been implemented, using the best practices by the creations of different functions.

To enable the capacity to run a lot of simulations all inputs were generated automatically and the unique manual configuration is the test configuration to define some parameters for the execution. The use of 5 configuration variables allowed the execution of all tests:

- `@num_exec`: number of executions to be run for after being possible the average calculation of all executions. The default value used in all simulations will be 100
- `@key`: defined key to be used for each simulation, in this case to simulate the number of destination ports. In these simulations the following values will be used: 5, 10, 15, 20, 50, 100, 200.
- `@length`: this configuration variable indicates the number of sequences to be transmitted between all defined destination ports, e.g. if the k=5 and length=4 the number of transmitted packets will be 20 (1a, 1b, 2c, 4d -> port 9000 and 2a, 2b, 2c, 2d -> port 9001). The default value used for these simulations will be 4
- `@drop_percentage`: number of drops to be generated into the initial array with all packets. Values that will be used for the simulation will be 1%, 3%, 5%, 10%, 15%, 20%, 25%
- `@switch_percentage`: number of switches to be generated into the initial array with all packets. Values that will be used for the simulation will be 1%, 3%, 5%, 10%, 15%, 20%, 25%, 60% and 70%

The function `generate_packets` enables the creation of an array with the packets to be used in the simulation. This function as inputs has the `key` and the `length` previously configured as configuration variables. The `char = 97` represents the letter a to be incremented using the length value. As return value of this function and array to be used in the simulation.

if `key=10` and `length=3` the total size will be 30 packets and the output array will be:

```
packets = [1a, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9a, 10a, 1b, 2b, 3b, 4b, 5b, …,10c]
```

```ruby
def generate_packets(key, length)
  char    = 97
  packets = []

  length.times { |i|
    key.times { |j|
      packets.push("#{j+1}#{(i+char).chr}")
    }
  }
  packets
end
```

Figure 11 - Generate packets function.

After creating the packets array, to simulate dropping of some packets, the `generate_drop` function was implemented. As input variables for the execution of this function the previously created array (`packets`) and the percentage of drops previously configured as a configuration variable (`percent_drop`). This method returns the array of packets without the dropped packets, but for debug information in the following example, the dropped packets will be printed to the console log just for knowledge and to help the simulation.

Example using the previously generated array:

`packets = [1a, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9a, 10a, 1b, 2b, 3b, 4b, 5b, …,10c]`

`percent_drop = 10`

At 30 total packets if you want 10% of drops the resulting array will be minus three packets, like some that:

`packets = [1a, 2a, 3a, ~~4a~~, 5a, 6a, 7a, 8a, 9a, 10a, ~~1b~~, 2b, 3b, 4b, 5b, …,~~10c~~]`

```ruby
def generate_drop(percent_drop, packets)

  packets.size.times { |i|
    if (rand(0.00...1.01) <= (percent_drop / 100)) && packets[i] != nil
      puts "#{packets[i]} -> DROPPED"
      packets.delete(packets[i])
      @drops += 1
    end
  }
  return packets
end
```

Figure 12 - Generate drops function.

The `generate_switch` was implemented too, to create the automatism to put packets out-of-order. As input for the execution of this function the packets array previously generated (`packets`) and the percentage of switched packets previously configured as a configuration variable (`percent_switch`). In this function the result will be a new array with the exchanged packets:

```
packets = [1a, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9a, 10a, 1b, 2b, 3b, 4b, 5b, …, 10c]

percent_switch = 10
```

At 30 total packets, if you want 10% of switches, the resulting array will have to go up and down 3 switched packets:

```
packets = [1a, 3a, 2a, 4a, 5a, 7a, 6a, 8a, 9a, 10a, 1b, 2b, 3b, 5b, 4b, …, 10c]
```

```ruby
def generate_switch(percent_switch, packets)

  packets.size.times { |i|
    if (rand(0.00...1.01) <= (percent_switch / 100)) && packets[i+1] != nil
      puts "V - #{packets[i]}"
      puts "^ - #{packets[i+1]}"
      switch        = packets[i]
      packets[i]    = packets[i+1]
      packets[i+1] = switch
      @switch += 1
    end
  }
  return packets
end
```

Figure 13 - Generate switches function.

After executing the previous functions and the designed algorithm, it's time to compare the results and to make this comparison the `compare_ini_final` was implemented. As input variables of this function the initial array generated by the function `generate_packets` (`packets_ini`) and the final array after the execution of the drops or switch function and the algorithm (`packets_final`). This function will do a complex comparison between these two arrays and calculate the percentage of matches.

The expected value is 100 % that will reflect the capacity of the tested algorithm to recover from all drops or all switched packets.

```ruby
def compare_ini_final(packets_ini, packets_final)
  sum = 0
  packets_ini.size.times{ |i|
    if packets_final[i] != nil && packets_ini[i] == packets_final[i][0].to_s
      sum += 1
    end
  }
  result = (sum * 100) / packets_ini.size

  result
end
```

Figure 14 - Compare initial array with final array function.

4.2.2. KUDP Algorithm Implementation

To implement the algorithm, the first step is to seek their definition, understand the interactions and apply the definition according to the code that reflects them.

First, two input variables are needed, the array of packages (packets) after drops and switches and the key used to generate these.

Follow the algorithm:

```
def kudp_algorithm(packets, key)
  iterations = Array.new(packets.size) { Array.new(20) }
  packets.size.times { |i|
    (key/2).times { |j|
      if packets[i+j] != nil
        packet = packets[i+j]
        insert  = i+1

        while packet != nil
          if iterations[i][insert-1] == nil && ((insert%key) == packet.to_i || ((insert%key) == 0 && key == packet.to_i))
            iterations[i][insert-1] = packet
            packet                  = nil
          end
          insert = insert + 1
        end
      end
    }
  }
  ordered = []
  (iterations[iterations.size-1].size).times { |i|
    lines = {}
    (iterations.size).times { |j|
      if iterations[j][i] != nil && lines[iterations[j][i].to_sym]
        lines[iterations[j][i].to_sym] += 1
      elsif iterations[j][i] != nil
        lines[iterations[j][i].to_sym] = 1
      end
    }
  ordered.push(lines.max_by{|pack,value| value})
  }
    i = (ordered.size) - 1
    while i > 0
      j = 0
      while j < i
        if (ordered[i] != nil && ordered[j] != nil)
          if (ordered[i][0] == ordered[j][0]) && (ordered[i][1] >= ordered[j][1])
            ordered[j] = nil
          elsif (ordered[i][0] == ordered[j][0]) && (ordered[i][1] < ordered[j][1])
            ordered[i] = nil
          end
        end
        j += 1
      end
      i -= 1
    end
  ordered
end
```

Figure 15 - KUDP Reconstruction Algorithm function.

4.2.3. Run Tests

After implementing all the below method, you can run tests and it is only necessary to change 4 variables to run it:

- Define the number of executions for this test (`@num_exec`)
- Define the key to be used (`@key`)
- Define the number of sequences to be used (`@length`)
- Define the percentage of drops (`@drop_percentage`)

- Define the percentage of switch (`@switch_percentage`)

```
#Config Variables ---------------------------------------------------------
@num_exec = 100
@key = 100
@length = 4
@drop_percentage = 15.00
@switch_percentage = 15.00
#---------------------------------------------------------------------------
```

Figure 16 - Configuration variables.

After that it is necessary to execute all the methods in the following sequence:

- `generate_packets`
- `generate_drop`
- `generate_switch`
- `kudp_algorithm`
- `compare_ini_final`

Follow the usage example:

```ruby
#-------------------------------------------------------------------------
exec = 0
result = 0
while exec < @num_exec  do
puts "EXEC: #{exec}"
#Generate packets-------------------------------------------------------
packets = generate_packets(@key, @length)
packets_ini = packets.clone
#-------------------------------------------------------------------------
# Drop (Percent Drop ex: 10.00 % | 20.00 % | 30.00 %)--------------------
puts "DROPPED PACKETS:"
packets_dropped = generate_drop(@drop_percentage, packets)
puts "DROPPED FINAL ARRAY"
puts packets_dropped
#-------------------------------------------------------------------------
# Switch (Percent Switch ex: 10.00 % | 20.00 % | 30.00 %)----------------
puts "SWITCHED PACKETS:"
packets_switched = generate_switch(@switch_percentage, packets_dropped)
puts "SWITCHED FINAL ARRAY"
puts packets_switched
#-------------------------------------------------------------------------
# KUDP Algorithm ---------------------------------------------------------
final = kudp_algorithm(packets_switched, @key)
#-------------------------------------------------------------------------
# Sum of all executions--------------------------------------------------
result = result+compare_ini_final(packets_ini, final)
#-------------------------------------------------------------------------
# RESULTS ----------------------------------------------------------------
puts "-----------------------------------"
puts "Packets (each execution)"
puts packets_ini.size
  exec +=1
end
puts "Final Packets - TOTAL (all executions)"
puts ((@key * 4) * @num_exec)
puts "Final Result % (efficiency)"
puts ((result / @num_exec)+ ((@drops * 100)/ ((@key * 4) * @num_exec)))
puts "Generated Drops %"
puts ((@drops * 100)/ ((@key * 4) * @num_exec))
puts "Generated Switches %"
puts ((@switch * 100)/ ((@key * 4) * @num_exec))
puts "Final out-of-order %"
puts ((@packets_out_of_order * 100)/ ((@key * 4) * @num_exec))
puts "Final lost packets %"
puts ((@packets_null * 100)/ ((@key * 4) * @num_exec))
#-------------------------------------------------------------------------
```

Figure 17 - Simulation execution.

If the file with all the code like the name kudp.rb is needed, run this command:

```
# ruby kudp.rb
```

How to interpret console output results:

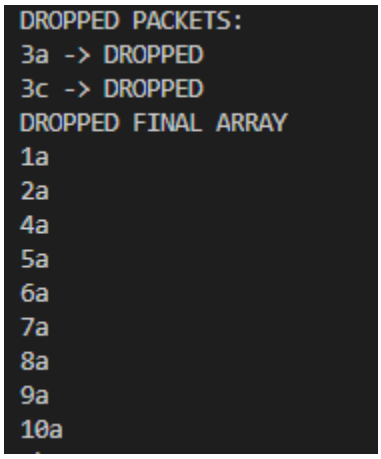

Figure 18 - Console Output (Dropped packets).

- **DROPPED PACKETS**: Packets dropped identification (Figure 18).
- **DROPPED FINAL ARRAY**: Final packets array after running the `generate_drop` function (Figure 18).

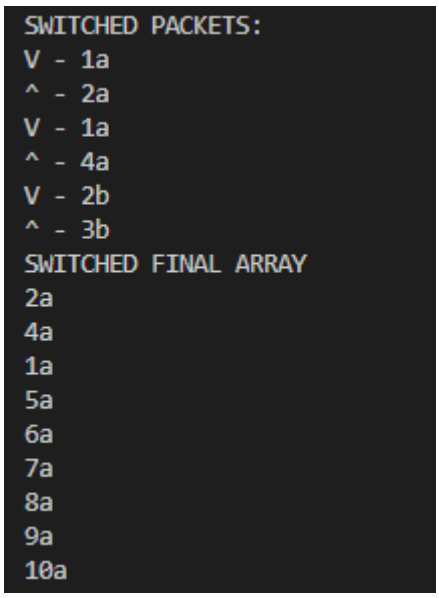

Figure 19 - Console Output (Switched packets).

- **SWITCHED PACKETS**: Switched packets identification and if the move was to up or down (Figure 19).

- **SWITCHED FINAL ARRAY**: Final packets array after running the `generate_switch` function (Figure 19).

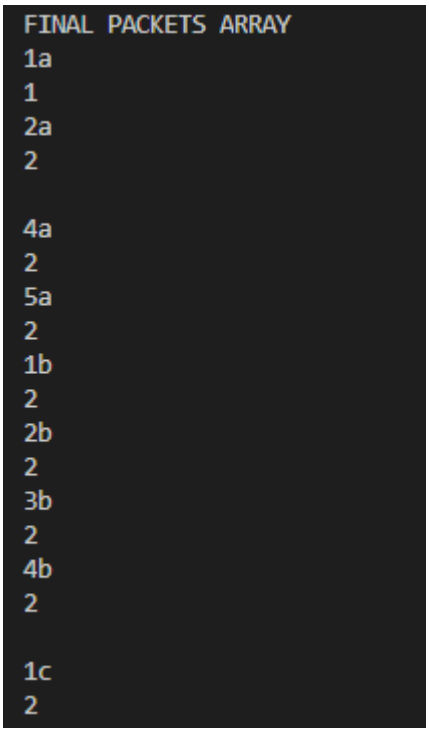

Figure 20 - Console Output (Final Array).

- **FINAL PACKETS ARRAY**: Array of final packages after executing the `kudp_algorithm` function with the number of occurrences that find the package at this position (Figure 20):
    - 5a – packet identification
    - 2 – found in 2 occurrences

If you fund an empty space, this package is lost. In this case, packet 3a is a lost packet.

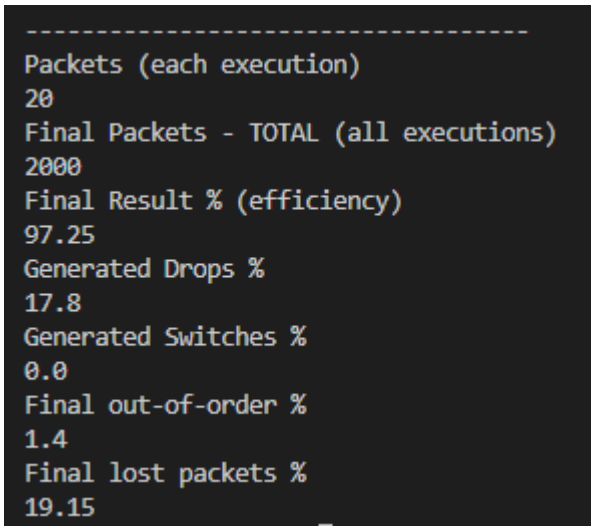

Figure 21 - Final simulation results.

- **Packets (each execution):** Generated packets for each defined execution in configuration variable `@num_exec`.
- **Final Packets – TOTAL (all executions):** Number of total packets used to run the simulation.
- **Final Result % (efficiency):** SRA efficiency percentage.
- **Generated Drops %:** Real drops percentage used in the simulation.
- **Generated Switches %:** Real switches percentage used in the simulation.
- **Final out-of-order %:** Real percentage of out-of-order packets identified by SRA.
- **Final lost packets %:** Real percentage of lost packets identified by SRA.

4.2.4.Test results

In this point will be presented the simulation results that reflect the efficiency of the Keyed User Datagram Protocol (KUDP) for transmission data that includes the capability to identify lost and out-of-order packets and reorder them.

For this simulation a large number of tests will be run and the average will be calculated to ensure the reliability of the results obtained.

First tests are just swapping packets for testing the algorithm capability to detect the out-of-order packets. Different relevant keys (`k`) will be assumed and the value of packet switches will also be incremented to see how far the algorithm is able to detect them. Assume only one source port and multiple destination ports, `k` reflects the number of destination ports used in this simulation.

In the following simulation it is possible to see the ability to recover the packets to the correct order in percentage based on the relationship of a key and a certain percentage of switches.

Table 1 - Efficiency of SRA for packets received out-of-order vs key length.

	1.00%	*3.00%*	*5.00%*	*10.00%*	*15.00%*	*20.00%*	*25.00%*	*60.00%*	*70.00%*
k=5	99.95%	99.75%	99.20%	96.95%	94.40%	91.45%	88.55%	75.50%	75.10%
k=10	100%	100%	100%	100%	99.95%	99.90%	99.75%	92.37%	90.07%
k=15	100%	100%	100%	100%	100%	99.98%	99.93%	97.61%	94.24%
k=20	100%	100%	100%	100%	100%	100%	99.97%	99.06%	97.53%
k=50	100%	100%	100%	100%	100%	100%	100%	99.99%	99.97%
k=100	100%	100%	100%	100%	100%	100%	100%	100%	99.99%
k=200	100%	100%	100%	100%	100%	100%	100%	100%	100%

From the simulation described above in the table, it is possible to extract a lot of relevant information:

- `k<10`: There is no ability for the algorithm to retrieve all packets
- `k=15`: The algorithm has the availability to detect and reorder 15% of switches where this percentage of switches is the acceptable value.
- `k>100`: Packets recover increase exponentially and more than 60% can be recovered.

Now you are thinking, to have `k=100`, a lot of ports are needed to receive packets, that's true, but to have 60% of exchanges something is wrong with the communication between the two machines. In my opinion until 15% of switches are acceptable, if there are more, something is wrong.

In the next graph (Figure 22) you can better assimilate the previous information provided in the table.

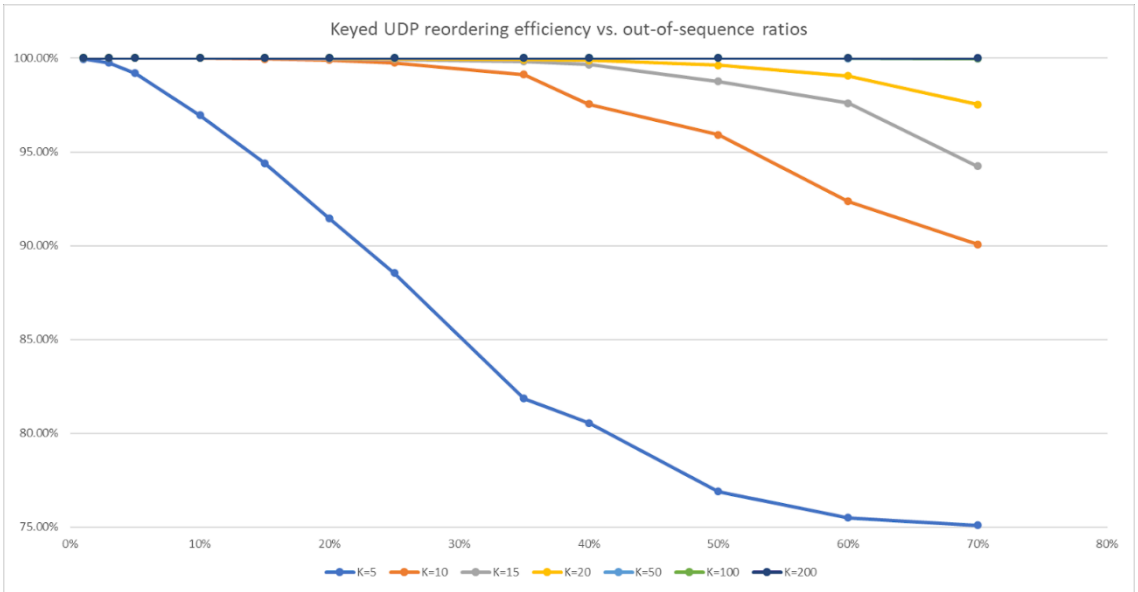

Figure 22 - Efficiency of SRA for out-of-order packets and different key lengths.

Next tests are just simulating drop packets for testing the algorithm capability to detect these drops. Like in previous tests, different relevant keys (`k`) will be assumed and the value of packet drops will also be incremented to see how far the algorithm is able to detect them. Assume only one source port and multiple destination ports, `k` reflects the number of destination ports used in this simulation.

In the following simulation it is possible to see the ability to detect drop packets in percentage based on the relationship of a key and a certain percentage of drops.

Table 2 - Efficiency of SRA for packets lost vs key length.

	1.00%	*3.00%*	*5.00%*	*10.00%*	*15.00%*	*20.00%*	*25.00%*
k=5	100%	100%	100%	99.75%	99.10%	97.50%	95.40%
k=10	100%	100%	100%	99.88%	99.50%	97.82%	96.00%
k=15	100%	100%	100%	99.97%	99.90%	99.25%	96.30%
k=20	100%	100%	100%	100%	99.98%	99.25%	97.15%
k=50	100%	100%	100%	100%	100%	99.70%	97.57%
k=100	100%	100%	100%	100%	100%	99.93%	98.38%
k=200	100%	100%	100%	100%	100%	100%	99.10%

From the simulation described above in the table, it is possible to extract a lot of relevant information:

- `k<15`: The algorithm has the capability to detect up to 5% of drops

- `k=50`: up to 15% of drops can be detected
- `k=200`: up to 20% of drops can be detected

As said in the switches test results, until 15% of drops are acceptable, after that something wrong with the connection is assumed.

In the next graph (Figure 23) you can better assimilate the previous information provided in the table.

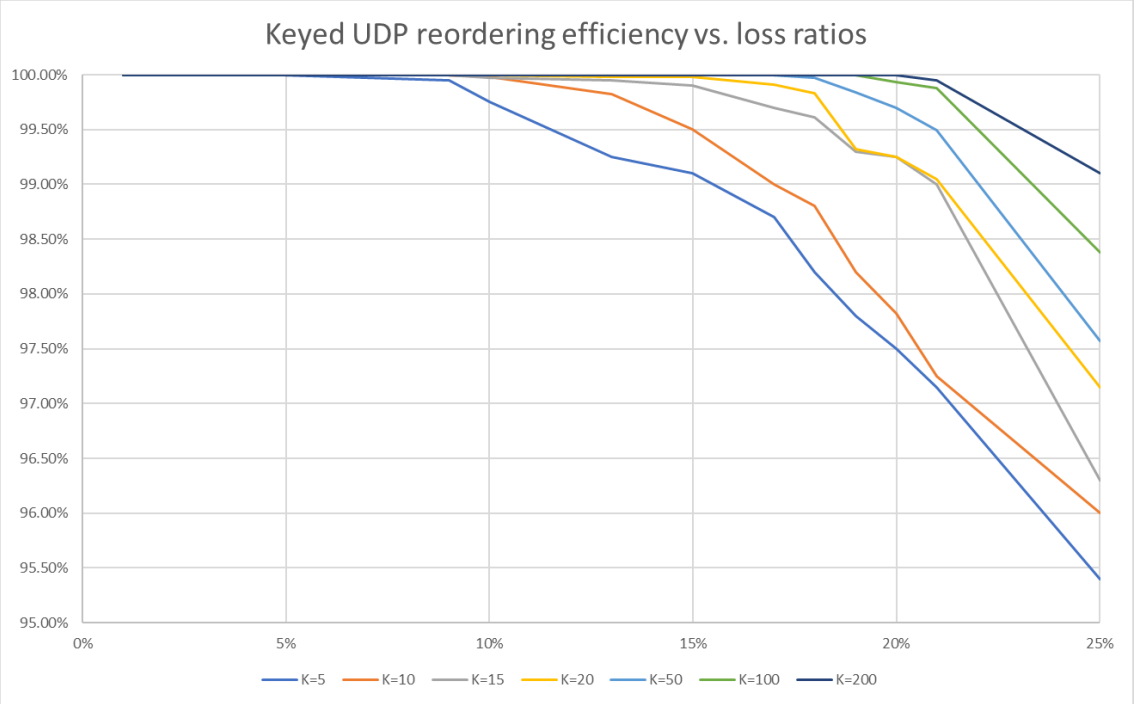

Figure 23 - Efficiency of SRA for lost packets and different key lengths.

After these simulations results the nice to have k is 50 to at least be able to recover up to 15% of switch packets and 15% of dropped packets.

The UDP is used for fast data transmission, without any kind of control or success metric and when used there are no mechanisms to detect lost or out-of-order packets and the receiver only will show the packets that have been received by the received order.

With the Keyed UDP the fast transmission will be achieved like UDP, because it uses the same datagrams headers and size, but with one difference, that is the capability that KUDP have to detect lost and out-of-order packets without any kind of control or success metric, just looking for packets received in a lot of ports with a simple algorithm that enables this detection.

Chapter 5

5. Conclusions and future work

5.1. General conclusions

The UDP is used when fast data transmission is necessary and doesn't require any kind of control or success metric. In the case of the TCP the transmission is more expensive because of the control mechanisms implemented to the packets control, but it allows lost packets retransmission and grants that packet will be received by the correct sequence using acknowledgements numbers.

For the Keyed User Datagram Protocol the recommended usage is for multimedia real time communications, where the receiver machine can apply the proposed algorithm to grant some success metrics without any kind of additional features or changes into the datagrams headers. The KUDP can be used when data integrity is relevant, but if some packets are lost or the algorithm can't reorder them the system is not compromised and is able to recover in the next transmission slot.

The proposal of KUDP, as described in [1] will use the format of the UDP datagram with zero overhead or any datagram headers changes.

In the previous simulation a lot of scenarios were covered, using different keys with different kinds of severity when talking about lost and out-of-order packets. As expected, longer keys return higher efficiency results.

5.2. Future work

As additional research, the combined occurrence of packets received out-of-order and packets lost needs to be addressed, keeping the ratios for these two types of events at a realistic level. Unfortunately, it was not possible to search the literature for realistic levels of packets arriving out-of-order or for packets that were lost.

References

[1] N. M. Garcia, F. Gil, B. Matos, C. Yahaya, N. Pombo and R. I. Goleva, "Keyed User Datagram Protocol: Concepts and Operation of an Almost Reliable Connectionless Transport Protocol," in IEEE Access, vol. 7, pp. 18951-18963, 2019, doi: 10.1109/ACCESS.2018.2886707.

[2] F. M. Gil, N. M. Garcia, B. Matos, N. Pombo, R. Goleva and C. Dobre, "Identifying Packet Loss and Reordering Packets in Keyed UDP Transmissions," 2020 IEEE Globecom Workshops (GC Wkshps, 2020, pp. 1-5, doi: 10.1109/GCWkshps50303.2020.9367443.

[3] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, https://www.rfc-editor.org/info/rfc793

[4] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, https://www.rfc-editor.org/info/rfc768

Attachments

The code to generate the simulation is present in the next attachment:

kudp.rb file

```
#Counts (Global Vars)
@drops = 0.00
@switch = 0.00
@packets_out_of_order = 0.00
@packets_null = 0.00


def generate_packets(key, length)
 char    = 97
 packets = []


 length.times { |i|
  key.times { |j|
   packets.push("#{j+1}#{(i+char).chr}")
  }
 }
 packets
end


def generate_drop(percent_drop, packets)


 packets.size.times { |i|
  if (rand(0.00...1.01) <= (percent_drop / 100)) && packets[i] != nil
   puts "#{packets[i]} -> DROPPED"
```

```
    packets.delete(packets[i])
    @drops += 1
   end
  }
  return packets
end

def generate_switch(percent_switch, packets)

  packets.size.times { |i|
   if (rand(0.00...1.01) <= (percent_switch / 100)) && packets[i+1] != nil
    puts "V - #{packets[i]}"
    puts "^ - #{packets[i+1]}"
    switch       = packets[i]
    packets[i]   = packets[i+1]
    packets[i+1] = switch
    @switch += 1
   end
  }
  return packets
end

def kudp_algorithm(packets, key)
  iterations = Array.new(packets.size) { Array.new(20) }
  packets.size.times { |i|
   (key/2).times { |j|
    if packets[i+j] != nil
     packet = packets[i+j]
```

```
      insert  = i+1



      while packet != nil

       if iterations[i][insert-1] == nil && ((insert%key) == packet.to_i || ((insert%key) == 0 && key ==
packet.to_i))

         iterations[i][insert-1] = packet

         packet               = nil

        end

        insert = insert + 1

      end

    end

  }

 }

 ordered = []

 (iterations[iterations.size-1].size).times { |i|

  lines = {}

  (iterations.size).times { |j|

    if iterations[j][i] != nil && lines[iterations[j][i].to_sym]

      lines[iterations[j][i].to_sym] += 1

    elsif iterations[j][i] != nil

      lines[iterations[j][i].to_sym] = 1

    end

  }

 ordered.push(lines.max_by{|pack,value| value})

 }

  i = (ordered.size) - 1

  while i > 0

   j = 0

   while j < i
```

```
      if (ordered[i] != nil && ordered[j] != nil)
       if (ordered[i][0] == ordered[j][0]) && (ordered[i][1] >= ordered[j][1])
         ordered[j] = nil
       elsif (ordered[i][0] == ordered[j][0]) && (ordered[i][1] < ordered[j][1])
         ordered[i] = nil
       end
      end
      j += 1
    end
    i -= 1
  end
ordered
end


def compare_ini_final(packets_ini, packets_final)
 sum = 0.00
 (@key*4).times{ |i|
  if packets_final[i] != nil && packets_ini[i] == packets_final[i][0].to_s
   sum += 1
  end
  if packets_final[i] != nil && packets_ini[i] != packets_final[i][0].to_s
   @packets_out_of_order += 1
  end
  if packets_final[i] == nil
   @packets_null += 1
  end
 }
 result = (sum * 100) / packets_ini.size
```

```
 result
end


#Config Variables ---------------------------------------------------
@num_exec = 100
@key = 5
@length = 4
@drop_percentage = 20.00
@switch_percentage = 0.00
#---------------------------------------------------------------------
exec = 0
result = 0
while exec < @num_exec  do
puts "EXEC: #{exec}"
#Generate packets----------------------------------------------------
packets = generate_packets(@key, @length)
packets_ini = packets.clone
#---------------------------------------------------------------------
# Drop (Percent Drop ex: 10.00 % | 20.00 % | 30.00 %)-----------------
puts "DROPPED PACKETS:"
packets_dropped = generate_drop(@drop_percentage, packets)
puts "DROPPED FINAL ARRAY"
puts packets_dropped
#---------------------------------------------------------------------
# Switch (Percent Switch ex: 10.00 % | 20.00 % | 30.00 %)-------------
puts "SWITCHED PACKETS:"
packets_switched = generate_switch(@switch_percentage, packets_dropped)
```

```
puts "SWITCHED FINAL ARRAY"

puts packets_switched

#-------------------------------------------------------------------

# KUDP Algorithm ----------------------------------------------------

final = kudp_algorithm(packets_switched, @key)

#-------------------------------------------------------------------

# Sum of all executions---------------------------------------------

result = result+compare_ini_final(packets_ini, final)

puts "FINAL PACKETS ARRAY"

puts final

#-------------------------------------------------------------------

# RESULTS -----------------------------------------------------------

puts "-----------------------------------"

puts "Packets (each execution)"

puts packets_ini.size

  exec +=1

end

puts "Final Packets - TOTAL (all executions)"

puts ((@key * 4) * @num_exec)

puts "Final Result % (efficiency)"

puts ((result / @num_exec)+ ((@drops * 100)/ ((@key * 4) * @num_exec)))

puts "Generated Drops %"

puts ((@drops * 100)/ ((@key * 4) * @num_exec))

puts "Generated Switches %"

puts ((@switch * 100)/ ((@key * 4) * @num_exec))

puts "Final out-of-order %"

puts ((@packets_out_of_order * 100)/ ((@key * 4) * @num_exec))

puts "Final lost packets %"
```

```
puts ((@packets_null * 100)/ ((@key * 4) * @num_exec))

#---------------------------------------------------------------------
```