CHALMERS

Overlay Networks and Distributed Denial of Service Attacks: Overview, study and evaluation of an application-enabled approach.

Master of Science Thesis in Computer Science and Engineering

NEGIN FATHOLLAH NEJAD ASL
RICARDO MOSCOSO ROMERO

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, March 2010

Overlay Networks and Distributed Denial of Service Attacks: An overview and evaluation of an application-enabled approach.

NEGIN FATHOLLAH NEJAD ASL
RICARDO MOSCOSO ROMERO

Examiner: Marina Papatriantafilou

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden March 2010

Abstract

Distributed denial-of-service attack (DDoS) as one of the most common Internet attacks today is an attempt to prevent legitimate network traffic from reaching the target and consequently to disable all services that this resource provides to the victim. The most common method to perpetrate DDoS attack is flooding the network with malicious packets to exhaust the network resources. This work is based on the fact that many network-based applications commonly open some known port(s) to communicate with their users; therefore, making themselves vulnerable to DoS or DDoS attacks. One of the main approaches to perform DDoS attack is to leverage the distributed network architecture (peer to peer networks) to create huge armies of zombies. These zombies are used to flood the victim with legitimate traffic. As there are large number of attacker machines in this method, defending against this attack is extremely complex. As peer to peer networks have become very important as one of the most popular content-delivery systems recently, the issue of defense against DDoS attack which use peer to peer network as their weapon turned into a big concern.

Considering this problem the main goal of this dissertation, after understanding the DoS and DDoS attacks deeply, is to simulate a DDoS defense system using a "pseudo-random port-hopping"" approach (called HOPERAA and BIG WHEEL algorithm) using ns-2, and analyze its performance under different circumstances. This "port hopping" approach is based on the work developed in [5]. The idea of this approach is to implement a solution capable of establishing a communication among the involved parties as well as hoping in a synchronized manner from port to port.

The analysis and evaluations performed in this dissertation include the overhead created by implementing the defense algorithm in a network under different defined conditions. Also the algorithms' behavior has been studied under variable clock drifts between the parties in the network. Simulating and analyzing the performance of these algorithms showed that this defense method behaves as expected and the results are consistent with the description given in [5].

Acknowledgements

It is a pleasure to thank those who made this thesis possible especially my parents who gave me the main support and encouragement I needed and my professor, Marina Papatriantafilou, and Zhang Fu whose supervision from the preliminary to the concluding level enabled me to develop an understanding of the subject. It would have been next to impossible to write this thesis without Marina and Zhang's help and guidance.

I also want to appreciate the help and support of my friend and colleague Ricardo Moscoso Romero and his great cooperation during almost one year of work on this thesis.

Finally I want to dedicate this dissertation to the brave people in Iran who proved their care about freedom of their country once again during recent summer while I was working on this project in Sweden.

Negin Fathollah Nejad Asl

February 2010

Acknowledgements

First of all, I cannot thank "The Swedish Foundation for International Cooperation in Research and Higher Education" enough for giving me their full support in this journey that I started quite a while ago; without them, things may have turned out different for me and, probably I would not have met such an amazing people, lived so many enriching experiences or even get to appreciate so many different cultures. I am also deeply grateful to one of my bachelor's professor, Gustavo Cervantes Ornelas; with whom, I will always be indebted not only for all the things I learned from him but, for all the unconditional help he has given me throughout these years.

My sincere gratitude goes to Marina Papatriantafilou and Zhang Fu; incredible hard-working people and amazing role models; their motivation and dedication inspired us to work harder and do our best in whatever task we were working on. They dealt with us for months not only with infinite patience but also with an absolute support and resolve; without their serenity, critique and continuous suggestions, this dissertation would have not been as entertaining and enlightening as it was.

I would like to thank all of my friends who played a huge role in my life; they have made me laugh with their irreverence but, also taught me the importance of taking a break whenever you feel like there is nothing left in you to go on… My special gratitude goes to Kasyab who let me crash at his place for what it was supposed to be "one month or two" and, turned out to be way much more; my roommates at Frölunda, who always were there for me even when I could not do the same for them; my friends back in Mexico whom, no matter how busy they were, always had time to spend and party with me whenever I went back home during Summer or Christmas; Shuvo, Rony, Cyril, Azadeh, Nojan, Molood, Sofia and so many other people that, even though I don't mention them, each one of them played a very important role not only during the time this dissertation took but, from the very moment I met them and realized how difficult it would be to carry on without them.

I will eternally be indebted and grateful to God for my loving family, they have always given me exactly what I need, their love, support and complete understanding. My sincere gratitude goes to them for, gladly and without any doubt in their hearts, letting me pursue every single one of the dreams I have had no matter how outrageous they were.

Finally, I would like to thank Negin for being there for me whenever I needed her, not only as my partner in this thesis but also, as one of my dearest friends…

Ricardo Moscoso Romero
December 2009

Table of Contents

List of Abbreviations

ACK	Acknowledgment
CBQ	Class-Based Queuing
CBR	Constant Bit Rate
CONSER	Collaborative Simulation for Education and Research
DARPA	Defense Advanced Research Project Administration
DDoS	Distributed Denial of Service
DFS	Depth First Search
DHS	Distributed Hash Tables
DoS	Denial of Service
FTP	File Transfer Protocol
HOPERAA	Hopping Period, Align and Adjust Algorithm
IP	Internet Protocol
NAM	Network Animator
ns or ns-2	Network Simulator
NSF	Network Specific Facility
Otcl	Object-Oriented extension of Tcl
P2P	Peer-to-Peer
RED	Random Early Detection
SAMAN	Simulation Augmented by Measurement and Analysis for Networks
Tcl	Tool Command Language
TCP	Transmission Control Protocol
TELNET	Teletype Network
UDP	User Datagram Protocol
VBR	Variable Bit Rate
VINT	Virtual InterNetwork Testbed

1
Introduction

"Cyber attacks have plunged entire cities into darkness."
- Barack Obama

In the past few years, Peer-to-peer networks (from now on we will use the acronym "P2P" when referring to this kind of networks) have become immensely important as one of the most popular Content-Delivery systems. Proof of this, is evident when looking at the Ipoque's Internet Study for 2008/2009, which provides an overview of the Internet's current state based on the analysis of 1.3 petabytes of Internet traffic in eight regions of the world (Northern Africa, Southern Africa, South America, Middle East, Eastern Europe, Southern Europe, Southwestern Europe, Germany). From this analysis it was concluded that even though the amount of P2P traffic has decreased, since the last time the study was conducted, it still generates the most traffic in all regions [1].

During the early beginnings, P2P networks and applications were mainly used in between home users, since it offered a way for people to share files among each other in a very reliable way; however, nowadays companies have been using the same approach for their own business strategy, shifting from the usual Client-Server model to the Peer-to-Peer model. Two examples of the previous statement are SKYPE which uses P2P protocols to forward phone calls around the net, and JOOST which offers "peer-to-peer" internet Television. The end-service might be different, but these and many other companies understood the importance of this approach and its many advantages, such as extensive object replication (due to the large number of peers) which at the same time increases availability, lowers cost of ownership and achieves fault tolerance, all in one.

The importance in between P2P Networks and the topic from this thesis is that nowadays, Attackers are capable of create huge armies of "zombies" by taking advantage of this kind of architecture. Perpetration requires little effort on the attacker's side, since a vast number of insecure machines (peers) provide fertile ground to create "zombies" and, prevention of the attack is extremely complicated due to the large number of attacking machines and the similarity between legitimate and attack traffic. [2]

Distributed Denial of Service attacks (DDoS) had been aimed to organisms like government organizations, commercial enterprises, banks and social networking sites [3] with the porpoise of disable, or diminish, the services they provide to millions of end-users; but also, they have targeted specific users such as the case of a Georgian blogger whose accounts on Twitter, Facebook, LiveJournal, Google's Blogger and YouTube were targeted in a denial-of-service attack, disabling his/her Twitter account and raising several problems at the other sites. According to Max Kelly, chief security officer at Facebook, "It was a simultaneous attack across a number of properties, targeting him to keep his voice from being heard". [4] As you can see, just anyone can be targeted as a victim, and the next one could easily be you or me.

This thesis will explore the problem of DDoS defense from two directions: (1) it strives to understand the problem by analyzing DDoS attacks, first how they work in a minor scale (DOS attacks) to then move on in studying them at a wider scale (DDoS attacks); this will give the reader a more specific knowledge of the problem being faced and denote the importance of the given solution. (2) It presents the analysis and implementation of a DDoS defense system using a "Port-Hopping" approach. Our study is based on the work developed on [5]. In this paper, the authors suggest an approach to deal with "Application level floods" on a Client – Server Scenario however, since each Client interacts with the Server on an individual basis, just as TCP does, the algorithm is highly suitable for P2P uses as well.
The work in this thesis aims to take it further, by understanding and evaluating the algorithm's behavior under different circumstances to the ones presented in [5].

1.1 Motivation

As briefly discussed in the previous section, our study is based on the work by Zhang Fu, Papatriantafilou Marina and Philippas Tsigas in their paper "Mitigating Distributed Denial of Service Attacks in Multiparty Applications in the Presence of Clock Drifts".[5]

In the latter, the authors focused their efforts in developing a solution for one of the main methods used by an attacker, in order to deplete the computational resources of a specific target. Their work is based on the fact that many network-based applications commonly open some known port(s) to communicate with their users; therefore, making themselves vulnerable to Denial of Service (DoS) attacks. With the purpose of solving this problem, a "pseudo-random port-hopping" approach was followed; the goal behind this, was to implement a solution capable of establish a communication among the involved parties yet, being able to hop in a synchronized manner from port to port on the meanwhile. However, in order to achieve this, was also necessary to implement a practical, yet not intrusive, way to keep some sort of synchronization in between the time drift from the clocks of all the involved entities. Based on this scenario, two algorithms were proposed: (1) BIG WHEEL, which not only allows servers to communicate with multiple clients in a port-hopping manner but, also accomplishes communication-independency among the Clients; and (2) HOPERAA, which is the algorithm that allows each Client to hop, in a synchronized manner with the server, taking in consideration the presence of clock-drifts.

The authors analyzed the algorithm's performance and, in the "Experimental Study" section, gathered promising results under a scenario in which a single adversary was capable of perform blind and directed attacks to a set of ports in the Server; however, they did not studied in detail the algorithm's behavior by looking at how the different variables involved are adjusted and how such adjustment is affected under specific circumstances. Based on [3], [6], [7] we can conclude that the adversaries are getting more creative with time and, attacks are getting more and more intensive; that's why, under the supervision of Marina Papatriantafilou and with the help of Fu Zhang, we decided to take their work one step ahead and analyze how will these algorithms perform under different circumstances.

Since creating the right environment for our study cases, in the real world, will require of access to specific resources and particular conditions, we decided to set up our implementation and test-case scenario on a network simulator. For this purpose, we are using “ns” an event-based simulator, well-established within the research community, which uses
C++ and an object orientated version of Tcl called, OTcl. Simulations are written using Tcl scripts and the protocols are implemented in C++; since ns does not include any visualization tools by default, we will use the network animator NAM so as to obtain a graphic representation of the simulated scenario.

1.2 Outline

This dissertation is structured as follows. First we begin by giving an overview about all the related topics that will be discussed throughout this thesis. In chapter 2 we discuss about the DoS dilemma in order to denote the magnitude of problem and give the reader a more concrete knowledge about how it works. Also in chapter 2, we explain what a P2P networks is, its definition, architecture and benefits but more importantly, for the focus of this thesis, how can they be exploited to launch DDoS Attacks; after this chapter, the Reader will have all the necessary knowledge to understand how the solution, studied in this work, helps to mitigate the DDoS problem.

Since our study is based on the paper "Mitigating Distributed Denial of Service Attacks in Multiparty Applications in the Presence of Clock Drifts" [5] by Zhang Fu, Papatriantafilou Marina and Philippas Tsigas, in chapter 3 we lay the foundation to understand how the defense framework works. In this chapter we explain in detail the two algorithms suggested in [5]; first we explain all the steps and variables involved when executing the HOPERAA algorithm, used for synchronous port hopping in the presence of clock-drifts; and then, we explain how the BIG WHEEL is used for servers to support communication with multiple clients, in a port-hopping manner.

In chapter 4 we present a brief background and important highlights concerning the simulator; also, we introduce a practical approach of how the HOPERAA algorithm would function on a single Client/Server model (by using hand-derived calculations), explaining the outcome derived from each step and how each variable calculated would behave; but more importantly, an extensive and detailed explanation of how the framework, explained in chapter 3, was implemented and which scenarios were considered in order to gather significant data for this work. Finally, based on the knowledge and experience gathered throughout the development of this dissertation, in chapter 5 and 6 we present our points of view regarding future work and conclusions, respectively.

2
Background

"Technological progress is like an axe in the hands of a pathological criminal."
- Albert Einstein

In this chapter, we give an overview about all the related topics that will be discussed throughout this thesis. In order to really understand the magnitude of the DDoS problem, the reader should have some knowledge about how does it work; but above all, its possible repercussions.

We have discussed how P2P Networks provide fertile ground for an adversary to create "zombies"; since in most of the cases, attackers had taken advantage of insecure networks to deploy DDoS attacks, we consider is important to understand what they are, how they operate, but specially how are they related to the DDoS problem. Also in this chapter, we will analyze the problem to be mitigated by the implemented solution [5]; in this part we will study DDoS attacks by first, comprehending how they are launched in a minor scale (DOS attacks) to then understand their taxonomy at a wider extent (DDoS attacks). After this chapter, the Reader will not only have all the necessary knowledge to understand how the presented solution helps to mitigate the DDoS problem but also, a general knowledge of how these attacks work and the damage they can cause.

2.1 P2P Networks

As the title asserts, this section talks about the main features of Peer-to-Peer (commonly referred as P2P) Networks and how they work. In the subsequent segments, we will give an extensive explanation about the topic by covering areas such as its definition, architecture and benefits but more importantly, for the focus of this thesis, how can they be exploited to launch DDoS Attacks.

2.1.1 P2P Network Definition

Peer-to-peer (P2P) systems are distributed systems in which nodes of equal roles and capabilities exchange information and services directly with each other. [9] Let's disassemble this into pieces in order to understand it better; first of all, all users or components in the network are called "peers", and each one of them is capable of retrieving objects directly from each other without intervention of a centralized server. Unlike the traditional Client-Server architecture, in a P2P network each component within the system, depending on the situation, can play the role of a Server or a Client at any time and simultaneously. For example, when a peer allows others to download a file from its hard drive, the peer plays the role of a server; conversely, this peer could be also obtaining files from other peers hence acting as a client. Whenever a peer is in the lookout for a specific file, messages are interchanged among the parties in order to discover other peers and determine which one of them has the desired object. Once the object has been found, the peer who started the search can download the file directly from the peer providing it since, as discussed earlier, in a peer-to-peer topology all transfers are always done directly between the peer sharing the file and the peer requesting for it; nevertheless the control process, prior to the file transfer, can be implemented in many different ways.

Once we have understood what P2P networks are and how they work, we employ [8] and [10] to identify some of the most common models:

- **Centralized**: In this model, each peer publishes information about the content they offer for sharing, along with a peer ID, IP address, and other type of information, into a well-known central directory. (See figure 1) A TCP connection is established with the central server and whenever one of the peers wants to retrieve a file, the first thing it does is to send a query message to the server; this query usually includes some keyword or identifier of some kind which, at the server's side, is evaluated against the directory. The server then creates a list with all the peers that best match the request and sends it back to the client; on receiving the list, the peer selects a peer from which it directly retrieves objects. When a peer leaves the P2P network, the server detects the disappearance through the termination of the TCP connection. Since this model requires a centralized infrastructure to store the information of all peers, it is prone to show some scalability limits, since it might require bigger servers when the number of requests escalate, or larger storage when the number of users increase; also, in the topic this thesis concerns, it becomes a vulnerable point which could be targeted by an attacker in order to disable the whole network.

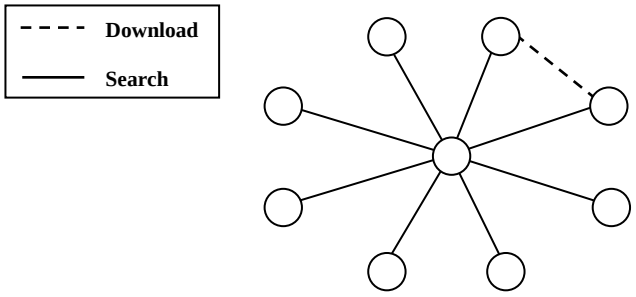


Figure 1: Centralized Peer-to-Peer network.

Some popular applications using this model are Napster, OpenNap, and instant messaging applications such as ICQ, Yahoo messenger, and MSN messages.

- **Flooded requests or Decentralized:** In this architecture, all peers are equal and there is no directory server so, is commonly referred as "pure P2P" (See figure 2). When a peer joins the network, it first sends a request to a bootstrapping node which provides him with a list of IP addresses of peers that have already participated in the network; then the new peer, advertises its address to the other parties thus creating a "neighborhood". When a peer wants to retrieve an object, the request is flooded (broadcasted) to directly connected peers, whom consequently flood their neighbors thus distributing the request throughout the whole parties; process is terminated when the request is answered or, a maximum number of flooding steps has occurred.

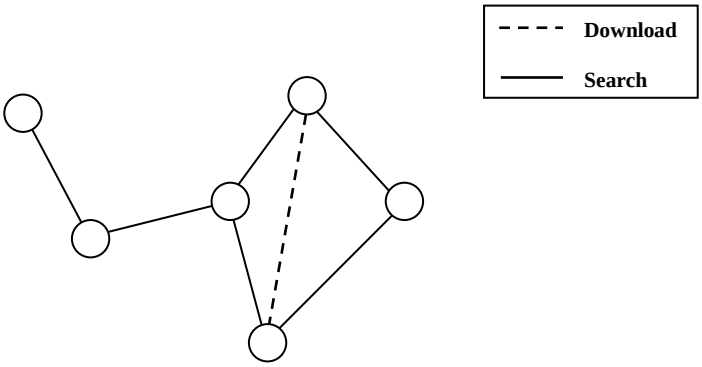


Figure 2: Decentralized P2P Network. Each node is only connected to its direct neighbors; file download can be established with nodes not directly connected.

One of its main advantages, over the other models, is that the failure of one or even several of the nodes has little impact on the performance of the network since there is no single point of failure. Gnutella and Freenet are typical applications using this mechanism.

- **Controlled Decentralized:** This model employs a hybrid, combination of the centralized and decentralized frameworks; in this architecture peers are clustered into groups so, the entire P2P network is logically formed by a conjunction of different groups. When a peer joins the network, it has to become a member of a specific group. In a group, there is a leader ('super-node' or 'ultrapeer') which maintains information of the objects deposited by peers in the group; thus, registration, query and objects retrieval in a group, are similar to the one in the Centralized architecture. Additionally, to achieve more results, one super-node can forward queries from its client peers to another super-node. (See figure 3)
-

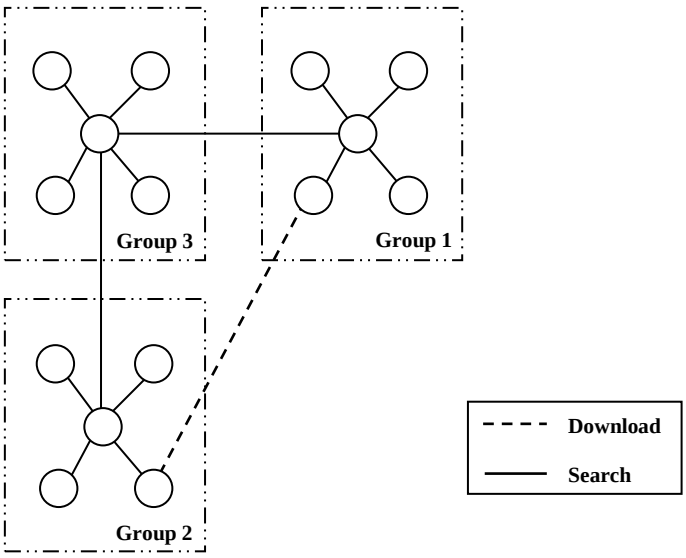


Figure 3: Controlled Decentralized P2P network. Each node in a group connected to a super-node. File download can be established with node from different groups.

Super-nodes change dynamically as bandwidth and the network topology change. A client-node keeps only a small number of connections open and each of those connections is to a super-node. Third generation P2P such as Morpheus, KaZaA, eDonkey2000, Groove, WinMX and FastTrack are typical applications using this model.

2.1.2 P2P Benefits

In this section we want to denote the benefits derived from using P2P networks, in order to magnify the reason why, in recent years, they have become a popular way to share huge volumes of data; increasingly receiving attention, from both industry and research community.
According to [8], selecting a P2P approach is often driven by one or more of the following benefits.

- **Cost distribution/reduction**
  In centralized systems, as they grow bigger, it becomes more expensive to support all the clients; however in P2P networks, such cost is distributed among all the peers. For example in the case of Napster, the centralized directory was only responsible for keeping the index required for sharing; while the cost for file storage, was never a problem since it was taken care of by the members in the network.

- **Resource aggregation and interoperability**
  Another great benefit derived from using P2P networks is that they can grow depending on the required needs so, attributes such as high compute power or storage space can be achieved relatively easy. P2P structures can be used to solve larger problems by segmenting them into small pieces and assigning them to each peer rather than, trying to solve the whole problem just by using a single machine. "File sharing systems, such as Napster, Gnutella, and so forth, also aggregate resources. In these cases, it is both disk space to store the community's collection of data and bandwidth to move the data that is aggregated." [8]

- **Improved scalability/reliability**
  This feature is achieved in conjunction with the algorithm implemented within the network, whether is centralized, decentralized or a hybrid of some kind, the network can become highly reliable (Capable of tolerating high node failures while maintaining connectivity and resolving searches within few messages [12]) and scalable (The ability of an unbounded number of new peers to join in the system [11]).

- **Anonymity and privacy**
  By using a P2P structure, all activities are local to the peers. The latter allows them a greater degree of autonomy and control over their data and resources; once in the system, users can avoid having to provide any information about themselves to anyone else; this is not the case in a central server approach, where the server will typically be able to identify the client, at least by Internet address.

- **Enables ad-hoc communication and collaboration**
  It instinctively offers dynamism and certain degree of freedom, since all members can connect and disconnect from the network at any time. The system does not rely on an established infrastructure, rather than a logical assembly, thus collaboration is achieved and highly scalable by just adding more peers into the network.

- **Greater Autonomy**
  Since each peer within the network is independent, the P2P model allows a high degree of autonomy for its peers, because it eliminates the need to rely on, and follow the rules set by, a single central resource provider.

2.2 Distributed Denial of Service (DDoS)

A DoS attack is an attempt to prevent legitimate users of a service or network resource from accessing that service or resource. DoS attacks usually make use of software bugs to crash or freeze a service, resource, or bandwidth limits by saturating all bandwidth. [15]

Based on this definition, we will explore the problem first by defining what a DOS attack is. A Denial of Service (DoS) attack is one of the most common attacks today. Different to many other threats, these attacks are not targeted at stealing, modifying or destroying information but to prevent legitimate network traffic from reaching the target thus, disabling all services the latter provides to its users. Although there exist many forms or methods to perpetrate a DOS attack the most common form consists in flooding the network with bogus packets, hence preventing legitimate network traffic. Another method is to drown the victim in fastidious computation so that it is too busy to do answer any other queries; in a DoS attack, only one machine is used to generate malicious traffic. Distributed DoS (DDoS) on the other hand, is an attack concerning a big number of subverted machines (zombies), coordinated by a central intelligence (attacker), launching simultaneous DOS attacks.

2.2.1 DDoS Attacks Classification

Now that we have a better understanding of what DDoS attacks are, we present the following taxonomy in which attacks are classified under common characteristics; for this, we use [15] and [16].

- **Degree of automation**

  In order to perform a DDoS attack, the adversary must achieve 3 basic objectives:

  1. Recruit multiple machines, known as zombies (recruit phase)
  2. Acquire certain control level over them (exploit phase)
  3. Instruct them to launch an attack over a specific target (attack phase).

In the early stages of DDoS attacks these phases were performed manually, by scanning remote machines for vulnerabilities, breaking into them, installing attack code and then command the attack; however, nowadays the process has been automated thus reducing, or even avoiding, the need for any communication between attacker and agent machines. After the recruit and exploit phases, the agent machine may propagate the attack code as follows:

- *Central source approach:* The attack code resides on a central source from where it is downloaded by the compromised host.
- *Back-chaining approach:* The attack code is downloaded from the machine that was used to exploit the system rather than from a centralized location.
- *Autonomous approach:* The attack code is injected during the exploit phase.

- **Weaknesses Exploited**

DDoS attacks exploit different weaknesses in order to achieve their goal; based on this, we can classify them into:

- *Vulnerability attacks:* Known also as 'semantic attacks', their aim is starvation of resources in the victim by exploiting implementation bugs, specific features or applications running in the victim's end.
- *Flooding attacks:* Known also as 'brute force attacks', their aim is starvation of resources in the victim by exasperating it with many 'seemingly legitimate transactions' up to the point, when it becomes unable to accept any more transactions.

- **Victim's Type**

As discussed previously, attacks are not necessarily aimed against single host machine. Depending on what they target, victims can be applications running on the target host, part of a network infrastructure or resources, such as bandwidth, or resources into the victim's network like router or a bottleneck link.

- **Attack Rate Dynamics**

  During the attack, each zombie is instructed to send a stream of packets to the victim; depending on the way these are sent, we differentiate between:

  - *Constant rate attacks:* After the attack is commanded, the zombies generate attack packets at a fixed rate, usually as many as their resources permit. The main feature of this approach is that is capable of disrupt the victim's services quickly, due to the sudden increase of packets.
  - *Variable rate attacks:* This attack aims to gradually degrade the victim's performance by changing the frequency rate at which the zombie sends packets to the victim.

- **Impact on the Victim**

  Depending on the impact of a DDoS attack in the victim, attacks can be disruptive, in which the goal is to completely deny the victim's service to its clients; or degrading, in which the objective is to strategically consume portion of a victim's resources, in order to degrade the service's quality offered to legitimate customers; since these attacks do not lead to total service disruption, they could remain undetected for a long time.

The reason why DDoS attacks are so popular nowadays is because they are very difficult to counterattack since, as mentioned before, they use a large number of computers to generate malicious traffic and these attacker machines could easily be spread all over the Internet or even be legitimate users; but also, as mentioned in [2], DDoS traffic very difficult to detect since it is highly similar to legitimate traffic so, it blends completely with the small amount of legitimate client traffic thus making detection very difficult until is too late.

Taking in consideration the previous, one would think that because DDoS attacks are distributed threats, the best approach would be to implement a distributed solution; however, wide deployment of any defense system is very difficult to enforce because Internet is administered in a distributed manner; and this is where the framework studied in this dissertation falls into place.

To conclude our discussion about DDoS attacks, we would like to answer one very important question, "Why do people perpetrate DDoS attacks?"
The main goal, no matter the case, is to inflict as much damage as possible on the victim or, to whoever relies on the target's correct operation. Ulterior motives may include personal reasons (such as revenge or merely for fun), or prestige (successful attacks on popular Web servers gain the respect of the hacker community [19]). However, some DDoS attacks are performed with darker motives in mind, such as material gain (like for example, the attacker could target possible competitors in order to disable their capability of offering a service or their corporate image [20]; as well as blackmail a company trying to obtain a financial benefit [18]), political reasons (aiming to create instability in the country [17] or simply to discredit and silence an specific public figure, like the case presented in [4]) and even could be used as a weapon during war times.

2.3 P2P & DDoS Attacks

So far we have discussed the many benefits derived from using a P2P strategy; however, now it is time to analyze how these same advantages, turn P2P networks into the weapon of choice for many attackers. With such a huge user base and lack of any authentication, P2P networks can be leveraged by an adversary to launch a DDoS attack against a victim machine on the Internet. The victim need not be a participant in the P2P network, and could be a web server, a mail server or even a home user's desktop. In this section, using [13] we focus on DDoS attacks triggered from exploiting P2P systems. The algorithms in a P2P system enable a peer to join the group, and maintain information about other members, even though nodes may join or leave the system. To scale to large group sizes, nodes maintain knowledge of only a small subset of group members and, this is where two of the most common attack approaches are used, index poisoning and routing table poisoning. [14]

2.3.1 Index poisoning attack

In index poisoning attack, the aim of the attacker is to make several peers believe that some popular file is present with the victim. To achieve this, the attacker A sends a false index record with the victim's IP address and port number to all the other nodes. The attacker usually uses the file hash of a very popular file so that there will be a large number of requests for it. On receiving the false index record, the peer B adds it into its index along with the location of the victim. B does not verify whether the victim has the corresponding file or even that A or victim is a participant in the P2P network. When some other peer C searches for that file, B will send V's record to C and the latter will try to establishing a TCP connection and retrieve the file from V. (See Figure 4) Since V could or not be part of the P2P network, it may not understand the message, thus ignoring it, replying with some error message or even terminate the connection. Unable to download the file, the downloading peer C may retry after some time.

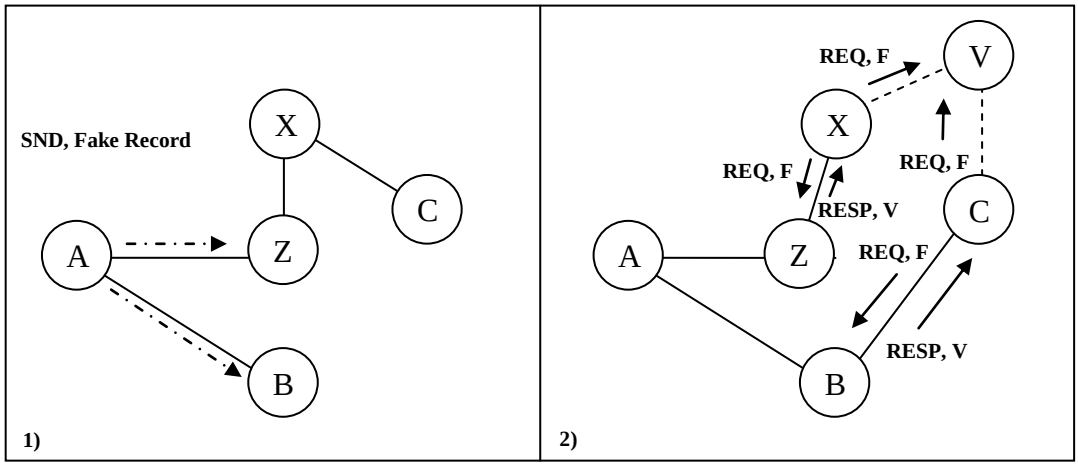


Figure 4: (1) Attacker 'A' sends fake record to every crawled peer, the latter add it into its index. (2) 'C' requests file 'F' from B and tries to retrieve 'F' from 'V', who might not even have the requested file.

As we stated before, since it is popular file the victim will receive a large number of requests from others peers whom, just like C, believe A has the desired file hence, making V unavailable to accept connections from legitimate users. (See Figure 4.2) Making things worse, Index poisoning can become a resilient attack since the fake records persist in the indexes for hours; even after the peers have failed on retrieve the requested file from the target.

2.3.2 Routing table poisoning

In routing table poisoning attack, the aim of the attacker is to make the peers add the victim as their neighbor. To achieve this, the attacker A sends node announcement message to every crawled peer. In these messages, the attacker includes the victim's IP address and port number so that B includes it into its routing table. Whenever a peer receives a search query or a maintenance message, it may select the victim from its routing table and forward the message to the victim. If the attacker poisons the routing table of a large number of peers, the victim may receive a flood of search queries and maintenance messages, saturating the victim's link. (See at figure 5)

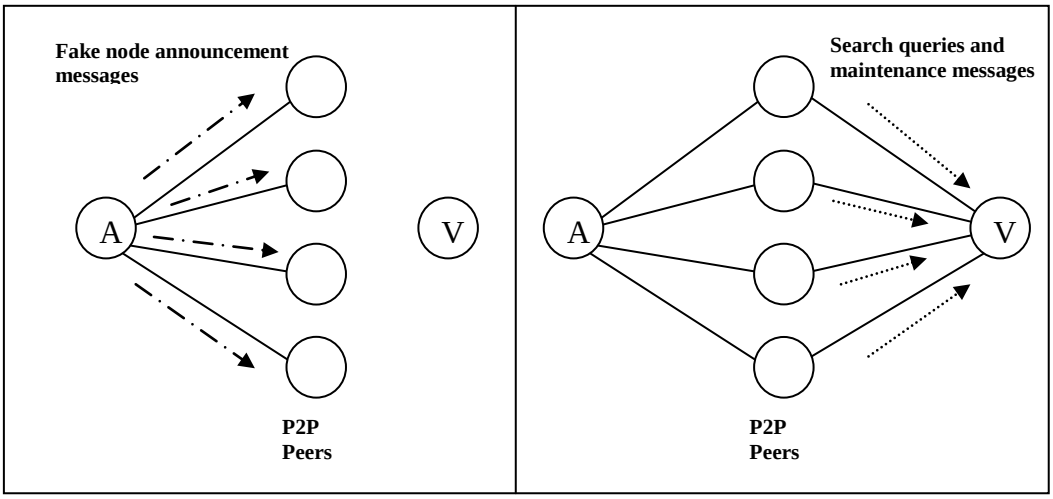


Figure 5: (1) Attacker 'A' sends node announcement message (V's information) to every crawled peer. (2) Whenever any of those peers receives a search query or a maintenance message, it forwards the message to the victim flooding its links.

Since the victim is not a participant in the P2P network, it will typically respond with a 'Port unreachable' message, flooding not only on upstream but also on downstream direction. The routing table poisoning generates a burst of messages directed at the victim but, different to the previous attack, after the victim fails to respond it may be removed from the poisoned routing table.

3

A port hopping approach against DDoS

"Defense is the stronger form with the negative object, and attack the weaker form with the positive object."

- Ernest Hemingway

In this chapter, we give an extensive and detailed explanation of how the framework designed in [5] works.

To begin with, the solution was designed to overcome a very common vulnerability, present at the application layer, in which certain programs may open ”well-known ports” in order to perform whatever action they're meant to do thus, an attacker, can eavesdrop some packages, discover which port is being used and launch a directed attack over such port; or, even if it can't discover which port is being used, I can still perform blind attacks (sending packets to a largely random set of ports, to then target any of the ports who responded) and eventually accomplish the same objective. Taking this scenario in consideration, the solution studied is based on the idea that the parties involved are capable of communicating with each other "hopping" in between different available ports over time; thanks to this, an attacker is not able to perform a attack over a particular/vulnerable port (used for communication) since the latter is always changing in a synchronized manner. In order to achieve such behavior, and overcome the need of a global synchronization mechanism in the system, two algorithms were proposed; (1) BIG WHEEL, which is used for servers to support communication with multiple clients, in a port-hopping manner and; (2) HOPERAA, used for synchronous port hopping in the presence of clock-drifts. In the following sections, we will discuss in detail how these algorithms work.

3.1 Hopping Period, Align and Adjust Algorithm

The Hopping Period, Align and Adjust algorithm (Referred as to HOPERAA in this thesis) is an adaptive algorithm, which is executed by each client when its hopping period length and alignment drift apart from the server's; the latter, ensure synchronization among the parties, without having to rely on a "common synchronization" server.
Within a network, is very common for the clients to have local clocks which differs from the one of the server, sometimes it could be slower and sometimes faster; since the ports being used for communication become available and unavailable over time, the periods of the Client and Server may start drifting apart from each other after some time, causing messages loss due to the fact that the Client may send messages to some of the Server's ports that has been closed or not yet open due to asynchronous clocks. The HOPERAA algorithm fulfill its objective by dealing with problems as the previous scenario and, avoiding messages losses due to unsynchronized port hops.

3.1.1 HOPERAA's Description

Before explaining how the algorithm works, first we set the following ground rules and assumptions:

- Each communication party has its own clock and the clock rate of each local clock is constant.
- The server's clock is used as reference for the whole operation hence each client's clock drift is defined as the ratio between its own clock and the server's clock rate.
- The client and server share a "common secret", which is a pseudo-random function *"f_ψ"* used to generate the port number for transmission
- "μ" is the maximum round-trip delivery latency for the messages.
- This solution mitigates attacks based on the application layer; therefore it's assumed that network is always available and attacks depleting the bandwidth of the server's network are not considered.
- The server has a set of N ports (Port Number Space) available for communicating with legitimate clients.

- Ports opened at the server's side can be of two types, based on their function. (1) Worker ports, used for receiving data messages from the client or (2) Guard Ports, used for receiving coordination messages from the client. Guard Ports can become worker ports after some time.
- Worker ports are opened every "L" time units and, they remain open for "L+ μ" time units. (figure 6)

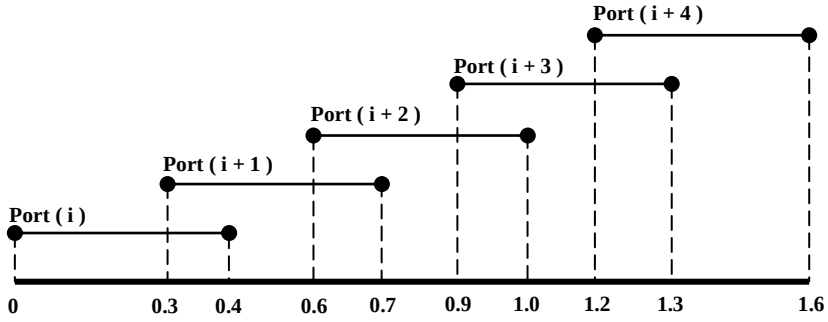


Figure 6: Assuming that L=0.3 and μ = 0.1, the worker ports at the server side will open and close as above presented.

Now that we have established a common ground, let's study how the solution operates. Communication is achieved throughout these phases:

- **Phase 1: Contact-Initialization**

  During this phase, the Client contacts the Server without any "well-known" port being opened at neither the server's side nor, "C" having to rely on a third-party to get the port information.
  In order to achieve this, the server must do the following:

  1. Divide the range of port numbers into "k" intervals evenly.
  2. Open "k" different guard ports at the same time, one of them per one interval, and changes them every "τ" time units.

After the server has executed the previous actions, the algorithm then performs like this:

1. Client tries to contact the Server by sending "contact-initiation messages" to all the ports in a randomly-chosen interval. In this message the Client includes a timestamp "time" with the time at which the message was sent.
2. When the Server receives the "contact-initiation message", it waits until the next worker port opens, open a session for the Client who contacted it and replies with the following information:
    a. **"σ"**, seed used to compute the next worker port.
    b. **"time"**, Timestamp at which the reply was sent.
    c. **"t1"**, time at which the Server received the contact-initiation message from the Client.
    d. **"h1"**, timestamp at which the Client sent the contact-initiation message
3. If the Server doesn't receive any message from the Client by the next worker port, the session opened in the step 2 will be closed; on the other hand, if the Client doesn't receive any reply from the Server, after "2μ +L" time units it will send "contact-initiation messages" to another randomly chosen interval.

The actions described above, from both Client and Server, are presented as an algorithm in figure 7.

Tc = *undef*;
Reply = *false*;

- Sending contact-initiation messages:

```
while (Reply == false) do
    I = SELECT (Ii | i ε {1, 2, ..., k})
    for (all ports in 'I') do
        SEND (init, time, p)
    end for
    WAIT (2μ +L)
end while
```

- After receiving ***(init, time, p)***:

```
t1 = time(now);
If (session == undef) then
    OPEN (session, C);
    h1 = time;
end if
WAIT (Next worker port opens)
SEND (reply, σ, timestamp, h1, t1)
```

Figure 7: (1) Algorithm for the Client in the Contact-Initiation Phase
(2) Algorithm for the Server in the Contact-Initiation Phase

- **Phase 2: HOPERAA Execution**

"Roughly speaking, after the contact-initiation phase, the application data from C to S is sent out through ports of S that change with period L time units of S's clock, corresponding to 'Pc' time units in C's clock (initially Pc = L)" [5] however, before actually start sending data to the Server, the Client has to perform the actions described in this section. As previously mentioned, this phase is reached after the Client has received the reply from the Server and, before it starts sending Data Packets; in this phase, the HOPERAA algorithm uses the clock's information, from the exchanged messages, to determine whether the Client's clock is slower or faster than the Server's and, based on such, it takes the proper actions to ensure successful data transmission. The reply sent by the Server to the Client is structured as follows:

Pkt (*reply, h1, t1, timestamp, seed***)**

In order to keep synchronous communication in between the parties, the following actions are performed:

1. The "HOPERAA execution interval" is initiated to 0.
2. The Client initializes the following variables:
   a. **Hc (t4)** = Time at which the Client received the reply from the Server.
   b. **Hc (t1)** = h1
   c. **t2** = t1
   d. **t3** = timestamp
3. The Client, bounds its clock drift using the following:

   $$\rho Low \leq \boldsymbol{\rho} \leq \rho Up$$

   Where:

   $$\rho Low = \frac{hc(t4) - hc(t1)}{(t3 - t2) + 2\mu} \qquad \rho Up = \frac{hc(t4) - hc(t1)}{t3 - t2}$$

4. The "HOPERAA execution interval" is calculated based on the following conditions:

   a. *If (ρLow < ρUp < 1) then*

   $$\text{Hoperaa Execution Interval} = \frac{\rho Low \Delta}{1 - \rho Low}$$

b. *If (1 < ρLow < ρUp) then*

$$\text{Hoperaa Execution Interval} = \frac{\rho Up\Delta}{\rho Up - 1}$$

c. *If (ρLow < 1) and (1 < ρUp) then*

$$\text{Hoperaa Execution Interval} = \min\left\{\frac{\rho Low\Delta}{1 - \rho Low}, \frac{\rho Up\Delta}{\rho Up - 1}\right\}$$

5. The "HOPERAA execution interval" and the value of "Pc" are both adjusted based on the following conditions:

a. *If (1 ≤ ρLow ≤ ρUp) then* **Pc** = L (ρLow) *and*

$$\text{Hoperaa Execution Interval} = \frac{(\rho Up)(\rho Low)\Delta}{\rho Up - \rho Low}$$

b. *If (ρLow ≤ ρUp ≤ 1) then* **Pc** = L (ρUp) *and*

$$\text{Hoperaa Execution Interval} = \frac{(\rho Up)(\rho Low)\Delta}{\rho Up - \rho Low}$$

c. If none of the conditions above are fulfilled then, do nothing.

- Receiving ***(reply, σ, timestamp, h1, t1)***:

```
if (Reply == false) then
    Reply = True;
    Tc = 0;
    Pc = L;
        /* Start sending DATA */
        Seq = 0;
        Pold = f ψ (σ);
        Pnew = f ψ (σ + 1);
        While true do
          SEND (Data, Pold)
            If (i(Pc) - μ ≤ Tc ≤ i(Pc)) then
              SEND (Data, Pold)
            end if
            If (Tc == i(Pc)) then
              Pold = Pnew;
                Pnew = f ψ (σ + i + 1);
                i ++;
            end if
        end while
end if
```

Figure 8: Algorithm used by the Client to send Data to the Server.

- **Phase 3: Data Transmission**

  This phase is executed immediately after the Client has finished with the calculation from "phase 2" and, the following actions are taken:

  1. As soon as the Client receives the reply, it performs the following:
     a. Sets its internal timer "Tc" to 0. This variable increases at the same rate as the client's local clock.
     b. Uses the seed "σ" and pseudo-random function "f_ψ" to generate the worker ports Pi = $f_\psi(\sigma)$ and Pi+1 = $f_\psi(\sigma+1)$.
  2. After calculating the worker ports, the Client will send the data messages immediately to Pi.
  3. During the interval "[i(Pc) - μ ≤ Tc ≤ i(Pc)]", messages will be sent to both "Pi" and "Pi + 1".
  4. When "Tc becomes equal to "i(Pc)", "Pi" changes its value for the one of "Pi+1" and "Pi+1" is recalculated by using "$f_\psi(\sigma+i+1)$", at every i ε N*. Roughly speaking, we can say that "i" acts as an index, whose initial value is 1, and it increases every time "Pi" and "pi+1" are updated.

  Depending on the value of the HOPERAAA execution interval, the transmission may be stopped to execute Phase 1 and 2; however, data transmission will be resumed after the latter two phases accomplish their purpose.
  Actions described above, are presented as an algorithm in figure 8.

- **Phase 4: Termination**

  The Client will end the communication, by sending a "termination-message" and getting it acknowledged by the Server.

We tried to develop this section so that it would be as comprehensible as possible however; if something is not clear and to avoid confusions, we strongly recommend that the reader refers to reference [5] for more details on specific issues.

3.2 BIG WHEEL Algorithm

In this section the BIG WHEEL algorithm is considered to deal with multi-party communication, supporting several Clients connected to the same Server. Since each Client follows the Server's hopping procedure, and take the Server's clock as reference, they are capable of communicate independently from each other.

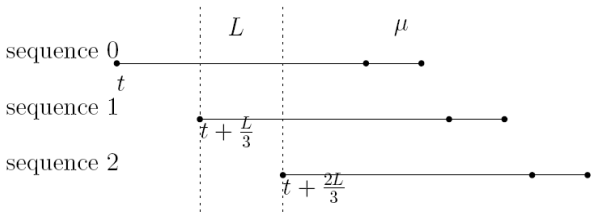


Figure 9: Shows the situation of m = 3 and the open time of P_0^i is t.

When using BIG WHEEL, worker ports still remain open for "L + μ" units of time but now the Server will support "m" port number sequences instead of just one, as in the previous section (see figure 9); this afford more clients and also decrease the maximum waiting time for each one of them. In the Clients side, by using λ and the pseudo-random function "f_ψ" it is possible to generate different port number sequences if different values of λ are given.
Apart from these changes, the phases previously explained and the actions performed in each one of them are the same, so when the server receives a contact-initiation message from the Clients, it will send the reply at the closest opening time of a worker port (considering all "m" sequences) along with the corresponding value of λ for the sequence to which that worker port belongs. The pseudo-code is the following:

- Buffer B stores all contact-initiation messages received
- Whenever is time to open a new port from any of the "m" intervals:

```
if (time(now) == OpenTime P^i_J) then
    λ = Corresponding value for sequence "j"
    σ = Corresponding value for P^i_J;
    for all clients in B do
        if (session == undef) then
            OPEN(session, C)
            h1 = timestamp of the corresponding contact-initiation message
        end if
        SEND(reply, σ, timestamp, h1, t1,λ)
    end for
  CLEAR(buffer B)
```

4

Implementation in ns-2

"There are no secrets to success. It is the result of preparation, hard work, and learning from failure."

- Colin Powell

In this chapter, we present an extensive and detailed explanation of how the framework, explained in the previous chapter, was implemented in the Network Simulator. First, we give a general impression on the network simulator and briefly discuss its basic structure, components, capabilities, etc. Since HOPERAA was implemented in ns by introducing a new Agent, throughout the second section we give an extensive explanation of how HOPERAA was implemented, the steps followed to simulate the algorithm and how to configure the simulator in order to support this new agent. One of the major breakthroughs of the HOPERAA algorithm is its ability to "hop" synchronously among different ports, when the parties involved have different clock rates; the strategy followed, in order to attain the latter, is also described in this chapter. In the last section, we present a practical approach of how the HOPERAA algorithm would function on a single Client/Server model (by using hand-derived calculations) then, we explain the outcome from each step and how each variable calculated would behave depending on the situation. We believe this is important since, it will help the reader to understand how the pseudo-code described in chapter 3 "looks like" in a practical environment, what sort of adjustment are expected from each time HOPERAA is executed but more importantly, this will set a baseline in understanding what the values obtained mean and what the expected performance will be.

4.1 The Network Simulator (ns)

In this section we use [26] to give a general impression on the network simulator (commonly known as ns-2, in reference to its current generation). Ns is a discrete event network simulator, developed at UC Berkeley, capable of simulate a wide variety of IP networks. It implements network protocols such as TCP and UPD; it can model traffic sources such as FTP, Telnet, Web, CBR and VBR; router queue management mechanism like Drop Tail, RED, CBQ and more. Apart from the already mentioned, one of the main reasons why ns was chosen as a tool for this thesis is because of its flexibility, since its open source and it offers a plentiful online documentation, it is possible to extend its original capabilities into fulfilling a special purpose. (As we will demonstrate further in this chapter)

Ns was built in C++ and provides a simulation interface through OTcl, an object-oriented dialect of Tcl. The user describes a network topology by writing OTcl scripts, and then the main ns program simulates that topology with specified parameters. Currently ns development is supported through DARPA with SAMAN and through NSF with CONSER, both in collaboration with other researchers including ACIRI.

4.1.1 Ns' Fundamentals

The simulator's version used in this thesis is ns - 2.31. This section talks briefly about the basic structure of ns and most of the information, used in describing the ns basic structure and network components, can be found in the 5th VINT/ns Simulator Tutorial/Workshop slides and the ns Manual (formerly called "ns Notes and Documentation") [25] and [22] respectively. In its most basic definition, ns is an Object-oriented Tcl script interpreter which offers several components such as a simulation event scheduler, network component object libraries, and network setup module libraries. In other words, to setup and run a simulation network, a user has to write an OTcl script that initiates an event scheduler, sets up the network topology using the network objects and execute specific actions throughout the simulation time.

All the different elements, necessary to build a specific topology, are defined in the simulator's object library, so users have the possibility of using a predefined object from this library or create a new network object to accomplish a specific task.

The event scheduler in ns is the one responsible for keeping track of the simulation time and, when the time is right, firing a particular event from the event queue; even if two or more events were scheduled to execute at the same time, since ns offers a single thread of control, the risk of locking or race conditions is inexistent. Each event scheduled, happens in an instant of virtual (simulated) time but, takes an arbitrary amount of real time to execute.

As we previously mentioned in the first section, ns uses two languages (OTcl and C++); the reason for this, is because it has two different kinds of things it needs to do. For instance, a detailed simulation of protocols requires of a programming language which can efficiently manipulate bytes, packet headers, and implement algorithms that run over large data sets. For these tasks, run-time speed is important and turn-around time (run simulation, find bug, fix bug, recompile, re-run) is not; on the other hand, a large part of network research involves slightly varying parameters or configurations, or quickly exploring a number of scenarios and for these tasks, turn-around time becomes a priority thus need for another language. In simpler words, C++ is fast to run but slower to change which makes it suitable for detailed protocol implementation; conversely, OTcl runs much slower but can be changed very quickly (and interactively), making it ideal for simulation configuration. As recommended by [22], having two languages raises the question of which language should be used for what purpose; their basic advice is:

C++	Tcl
• Anything that requires processing each packet • If it is necessary to modify the behavior of an existing module or create a new one.	• Simple Configuration and Setup Scenario. • If the problem can be modeled using any of the already defined Tcl modules.

In order to obtain NS simulation results, the first step is to create a Tcl script, feed it to the simulator and, when the simulation of such script is finished, NS will produce one or more text-based output files containing detailed simulation data; the data then, can be used for simulation analysis or as an input to a graphical simulation display tool called Network Animator (NAM).

In this work, we used Nam as an aid to obtain a visual demonstration of NS output and have a general overview of how the implementation behaves. NAM was developed as a part of VINT project and it offers a graphical user interface similar to that of a CD player (play, fast forward, rewind, and pause) as well as a display speed controller; although it can graphically present information such as throughput and number of packet drops at each link, the graphical information derived from NAM alone, cannot be used for accurate simulation analysis. [26]

4.2 HOPERAA implementation in ns-2

In this section we will discuss in detail how HOPERAA was implemented in ns-2; we will cover different topics from what changes were made to the ns source files in order to add a new agent, how the clock's drifts were simulated, the structure of the packets created for the HOPERAA agent, etc.
First of all, HOPERAA was implemented in ns-2 as a whole new agent due to the nature of the algorithm, from chapter 3 is easy to notice that in order to make it work, it is necessary to define a new type of packet which includes all the information related specifically to the algorithm also, this new entity should be able of processing each packet and derive an specific action based on the time it was received; because of these, and many other reasons, we decided that the easiest way to do it was to implement a new agent capable of dealing directly with all the actions involved in the HOPERAA Algorithm. In the following section, we explain more in deep how the whole simulation was defined and all the steps taken in order to cover these specific needs.

4.2.1 Adding the new Agent into ns-2

Whenever is necessary to add new agents into ns-2, there are several steps that have to be followed; reference [24] is the one we used as guideline to create our new agent and, to configure the simulator so that this new hoperaa agent could be invoked from the OTcl script.
As previously discussed, we will need to create a new type of packet which will include all the information necessary for the HOPERAA algorithm (struct 'hdr_hoperaa' in 'hoperaa.h'); for this, we defined a new packet type for the hoperaa agent. The first step to achieve the latter is to edit the file 'ns-2.xx/common/packet.h' and include the new definition for PT_HOPERAA:

```
enum packet_t {
        …….....
        // insert new packet types here
        …….....
        PT_HOPERAA,     //Packet protocol ID for HOPERAA
        PT_NTYPE        // This MUST be the LAST one
};
```

Also in this file, is necessary to edit the pinfo() as follows:

```
class p_info {
public:
        p_info() {
                name_[PT_TCP]= "tcp";
                name_[PT_UDP]= "udp";
                .....
                name_[PT_TFRC]= "tcpFriend";
                name_[PT_HOPERAA]="hoperaa";

                name_[PT_NTYPE]= "undefined";
        }
```

The file 'ns-2.xx/tcl/lib/ns-default.tcl' has to be edited too. This is the file where all default values for the Tcl objects are defined; it is necessary to make sure that in this file insert the following lines to set the default value for the packet size and the variables bound in 'hoperaa.cc' (see Appendix section), otherwise when the Otcl script is interpreted by ns, warning messages will be displayed.

```
Agent/hoperaa set packetSize_ 40
Agent/hoperaa set periodL_ 0
Agent/hoperaa set maxDeliveryLatency_ 0
Agent/hoperaa set totalPackets_ 0
Agent/hoperaa set noIntervals_ 0
Agent/hoperaa set cDrift_ 0
Agent/hoperaa set deltaTime_ 0
Agent/hoperaa set noSequences_ 0
Agent/hoperaa set totalPorts_ 0
Agent/hoperaa set sessionStatus_ 0
```

Finally, add into the list of object files, in the 'Makefile', one entry for each new cc file defined and recompile ns by typing 'make' in the ns directory. Remember to do a 'make depend' before you do the 'make', otherwise these two files might not be recompiled.

4.2.2 HOPERAA Packet taxonomy

This new agent will communicate with other agents alike by sending hoperaa packets; in these packets we include the following information:

1. A control variable which takes different values depending on the step the HOPERAA algorithm is in.
2. A time-stamp which is used by the involved parties to determine the sending time from each packet and initialize variables from the HOPERAA algorithm such as ‘h1’ or ’t3’ depending the case. (Look at chapter 3 for more information)
3. The seed value with which the receiver can calculate the next port that will be used for data transmission. (known as worker port)
4. The time at which the Server received the first contact initialization message from the Client (t1).
5. The time-stamp from the first contact initialization message received by the Server (h1).
6. And in the case of the BIG WHEEL algorithm, we include also the value of ‘λ’ which is used, in conjunction with the seed, to calculate the next worker port.

Each packet coming out from a hoperaa agent has this structure; however in some cases, like for example during the contact initialization phase, fields like ‘h1’,’seed’ and ‘t1’are not filled in since those are only used for when the Server sends the reply back to the client.
We are aware that the code might not be the best possible implementation and, it could always be improved or extended, that is why the definition of the packet can be found in ‘struct hdr_hoperaa’ inside ‘hoperaa.h’ at the appendix section.

4.2.3 HOPERAA in relation to ns

So far we have pointed out the presence of hoperaa agents and hoperaa packets however, in this section we will explain the role of this agent into the simulation and, more importantly, how does it relates to the rest of the objects from ns.

In our simulation, the topology is basically constructed using the following elements:

- **Hoperaa Agents**

  This agent is responsible of transmitting all the information required by Hoperaa thus, is the one sending "hoperaa packets" from one point to the other. This entity is responsible of sending all the contact messages during the contact initialization phase also, is the one who calculates the "Hoperaa execution interval" when the reply from the server has been received and, it schedules the next time at which the client must re-synchronize with the server. (There is an agent of this kind attached to the Server and, to each Client in the simulation)

- **TCP Agents**

  For our simulation, we are considering TCP as a protocol so we are using ns-2 Tahoe TCP agent "Agent/TCP" to simulate such behavior. (This agent is present at the client's side only)

- **FTP applications**

  We are using this entity, attached to the TCP agent, to generate traffic for our simulation. (This agent is present at the client's side only and, has to be attached to the TCP agent)

- **TCP-Sinks**

  This agent is used to receive the traffic generated from the FTP application and transmitted through the TCP agent; for this we are using a base TCP sink object "Agent/TCPSink" which is responsible for returning ACKs to a TCP source object for each packet received. (This agent is attached at the Server's side only)

- **Duplex Links**

  Used to connect the nodes with each other and achieve data communication in between the parties.

Now that we know which elements are involved, we can start explaining how all of them relate to each other. The best way to explain this is with an example, let assume that we have the following topology in ns:

Figure 10: Shows a Single Client – Server Topology.

Internally, the Client node is constructed like this:

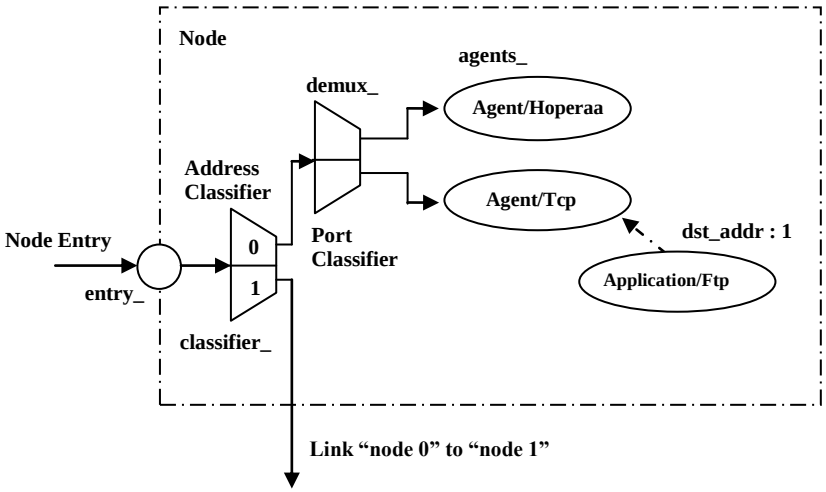


Figure 11: Shows the internal structure of a Client Node, its agents and internal communication links.

Alternatively the structure of the Server node is the following:

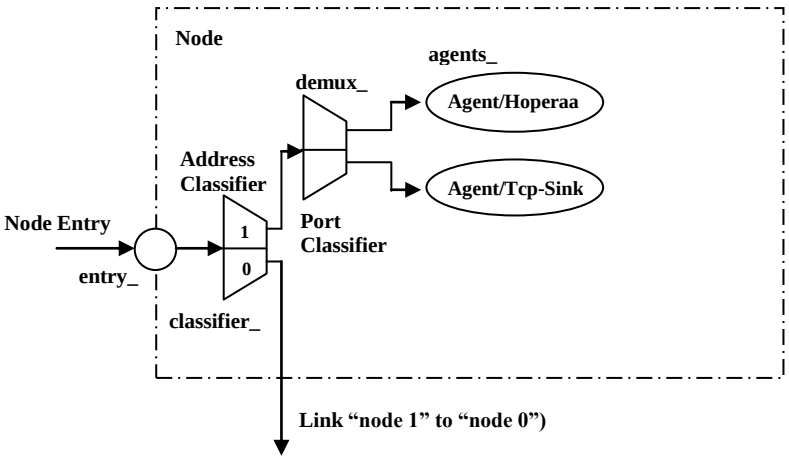


Figure 12: Shows the internal structure of a Server Node, its agents and internal communication links.

Now, let's say that the Client wants to communicate with the server so, the Server node first has to divide the range of port numbers into 'k' intervals evenly (See chapter 3). The latter is achieved through the hoperaa agent, which is also responsible of opening 'k' different ports (one port per one interval) and change them every 'τ' time unit; in each opened port, the hoperaa agent will be the responsible of receiving the contact initialization messages from the Client and store the timestamp values from such contact.

The first step for the Client node, is to reach the Server by starting the 'Contact-Initialization' phase, for this the node uses its own hoperaa agent which will send contact initialization messages to all the ports in a randomly chosen interval. When one of those messages arrive in one of the open ports, the hoperaa agent from the server will process such message accordingly and will send the reply at the moment the next worker port is opened hence, sending the Client all the data it needs to calculate the Hoperaa Execution Interval. When the packet is received at the Client's side, it is processed by the hoperaa Agent which will calculate the Hoperaa Execution Interval and, according to this value, will schedule when to re-synchronize with the Server. Once the hoperaa agent has finished its calculation, it will use the seed (received from the hoperaa packet) to calculate the value of the worker ports "Pold" and "Pnew". After all of this is done, the hoperaa agent configures the TCP agent so that this one send all the data to the specific port Pold, and hands over the control to the latter agent which will command the FTP application to start generating data packets. When it's time to re-synchronize with the Server, the hoperaa agent will "tell" the TCP agent to stop sending packets (which in consequence will stop the FTP application) so that it can send contact initialization messages to the Server and a new Hoperaa Execution Interval is calculated. When the hoperaa agent at the Server's side receives the contact-initialization message from the Client, it will check the stored values from the timestamps of the first contact initialization message received and will send the reply at the moment the next worker port is open thus, starting the described process all over again.

4.2.4 Modifications to ns Source Code

Now that we have a better idea about how all the agents and elements from ns relate to each other, there are some other changes we have to make. We didn't explain these changes in the previous sections because we wanted to set the proper background so that these modifications would make sense to the reader.
From the previous section, we know that the one responsible of transmitting the data packets in the TCP agent and, the one responsible of receiving, and sending the acknowledgment from each packet received is the TCP Sink; when we were creating the topology for our simulations, we came across with the following problems:

1. When it comes to TCP connection establishment in ns, the first packet sent by the Client's TCP agent is a control packet which is received by the TCP Sink at the Server's side and replied with an ACK, this allows the Client to start sending the data packets (It simulates the way TCP performs the Connection establishment in between 2 parties, however ns uses a two-way handshake); the problem arises when we have a scenario in which two or more clients contact the Server at the same time and, have to share the same worker port for a period of time; since for each TCP agent there has to be a TCP Sink associated, in the previous scenario we needed to attach the 2 TCP Sinks at the same port (so that it could receive data from Client 1 and Client 2) and the predicament was, that when you do that in ns (setting two TCP Sinks at the same port), although both sinks will receive such contact packet only one reply will be sent. The problem with this is that since only one of the Clients received the reply, the other one will have to wait unnecessarily and contact the Server at another time hence, interfering with the correct development of our implementation. (Look at "test1.tcl" in the appendix section for an example of this situation)
   To give solution to this problem, we modified "tcp-sink.cc", by including the following code inside "void TcpSink::ack(Packet* opkt)":

```
void TcpSink::ack(Packet* opkt)
{
    Packet* npkt = allocpkt();
    // opkt is the "old" packet that was received
      ……..
    // Andrei Gurtov
     acker_->last_ack_sent_ = ntcp->seqno();
     // printf("ACK %d ts %f\n", ntcp->seqno(), ntcp->ts_echo());

      hdr_ip *iph = hdr_ip::access(opkt);
      hdr_ip *iph_new = hdr_ip::access(npkt);
      iph_new->daddr() = iph->saddr();
      double t = Scheduler::instance().clock();
      printf("\n      [ TIME: %f : Sending ack to Packet #%d received in port %d
              from Node %d]",t,ntcp->seqno(),iph->dport(),iph_new->daddr());

     send(npkt, 0);
     // send it
}
```

2. In contrast, another problem we came across with was how to achieve packet replication when the HOPERAA algorithm requires it? As we know from the previous chapter, in HOPERAA there will be an interval in which the Client will have to send the exact same packets to "Pold" and "Pnew" alike; this becomes a problem when simulating in ns because the original TCP agent does have the capability of sending an exact replica of the packet to 2 different ports at the same time.

   The solution to this problem required for us to bound into "tcp.cc" a new variable called "replicate_"; the default value of this variable must be "-1" and whenever it takes a different value, the TCP agent will send a copy of the packet to the port given by the new value of "replicate_". Once the variable "replicate_" has been included in the code, the following has to be included in "void TcpAgent::output (int seqno, int reason)":

```
void TcpAgent::output(int seqno, int reason)
{
      int force_set_rtx_timer = 0;
      …….
      send(p, 0);

      if (replicate_ != -1) {
                  // Declare "t" at the beginning of the file
                  t = Scheduler::instance().clock();
                  Packet* pa = allocpkt();
                  pa=p->copy();
                  hdr_tcp *tcph_pa = hdr_tcp::access(pa);
                  hdr_ip *iph_pa = hdr_ip::access(pa);
                  iph_pa->dport() = replicate_;
                  printf("\n      [ TIME: %f : Node %d sends replicate (from packet#
                        %d) to port %d ]",t,addr(),tcph_pa->seqno(),replicate_);
                  send (pa,0);
       }
       if (seqno == curseq_ && seqno > maxseq_)
       …….
 }
```

4.2.5 Simulating Clock Drifts in ns

Simulating different clocks for each node was another problem that we faced when implementing the algorithm in ns. Since the simulator is running in a single computer, we had to find a way to ensure that each Client would have a different clock than the one used by the Server so that the time-stamps from the packets, would diverge thus obtaining relevant results. In order to achieve the latter, we used a very simple approach; let's assume that we have 3 different nodes, one of them is the Server and the other two are Clients, one of the clients is running two times faster than the Server's clock (Client 1) and the other one two times slower (Client 2).

From the previous statement we can conclude that we will use the Server's Clock as a reference to simulate Clients running faster or slower depending on the situation and, that we will need to simulate 3 different clocks in order to get different timestamps out of all the hoperaa packets being transmitted. What we did, was to use the simulator's clock for the Server and also as a reference for the rest of the "Application Clocks" in each client so, for example, when we say that the Client is two times faster than the Server then it means that for each unit of time that passes for the Server two will have passed for the Client and the same logic is applied for the Client that is two times slower.

Graphically speaking we will have the following timeline:

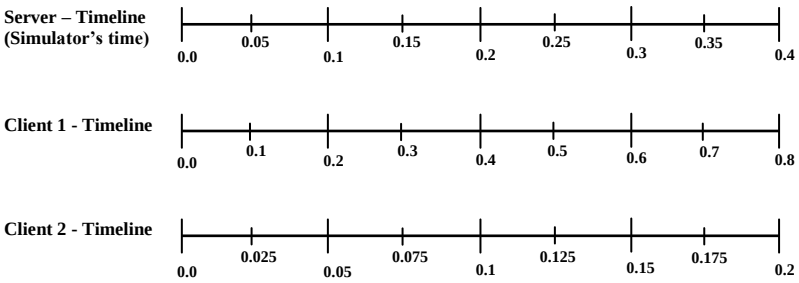


Figure 13: Shows the relation in between the "application time" from each node in contrast with the simulator's time.

Based on the above, we "play" with the simulation time in order to obtain the desired behavior out of each node. For example, every packet sent by the Client 1 will be stamped with (simulator's time) * (2) and the same applies to each action this agent performs; for instance, if the Client 1 has to send contact initialization messages at time = 0.5 (on the client's clock) then it would mean that ns will schedule such action to be executed at simulation time = 0.25. The time-stamp from the packets received by the Server will have the value 0.5 and the Server, when sending the reply, will stamp the packet with the actual simulation time (since the Server's and the simulator's clock are the same); if we assume that the server sent the reply at simulation time 0.3 and, it takes 0.05 to reach the Client 1, then it means that the Client will receive the packet at 0.35 "simulation time" but at 0.65 according to the Client's clock; we do this by subtracting 0.05 (the time it takes for the packet to travel from one point to the other) from 0.35 and then multiplying the result 0.30 by 2 which is equal to 0.60 (this value represents the time on the Client 1's clock at which the packet was sent by the Server). Finally we add the 0.05 that we subtracted at the beginning hence obtaining 0.65 which is the time at which the reply was received, according to the "application clock" in Client 1. This same logic applies to the Client 2 although, instead of multiplying the simulator's time by 2, we divide it by such value. By doing this, we can achieve different time-stamps in each packet that is being transmitted thus, obtaining significant results whenever the Hoperaa Execution Interval is calculated.

We can use the timeline presented above, to map a relation in between the "application time" from each node and the simulation time at which it corresponds in ns.

4.2.6 Example: Single Client/Server Scenario

Before presenting the results obtained from implementing HOPERAA in ns, we want to give a brief explanation of how the calculations are performed; this will set a baseline in understanding what the values obtained mean and what the expected performance will be.
For simplicity purposes in this case, let assume the following:

1. We will use a topology as the one showed in figure 10. (A single Client/Server)
2. The Client will be 2 times faster than the Server. (See previous section for further explanation)
3. We are assuming the following values of:
    a. **μ** = 0.1
    b. **L** = 0.3

Now, at time = 0 Client 1 will start the "Contact-Initialization" phase by sending packets to all the ports in a randomly chosen interval. These packets will have the following format:

Pkt (*init, timestamp, port***)**

Where timestamp is equal to 0; the Server will receive a contact initialization message at time = 0.05 (remember that μ is the round-trip maximum delivery latency) and will wait until the next worker port is opened (t = 0.3). Since the Client is 2 times faster than the Server, it means that at time =0.25, the Client's clock will be 0.5 (2μ + L) and the Client will start the contact initialization phase again (Since so far it hasn't gotten any reply from the Server) now with the timestamp value 0.5. The server will receive another contact Initialization message at time 0.30 but, it will send a reply with the information of the first contact initialization message received:

Pkt (*reply,h1, t1, timestamp, seed***)**

Where:

- h1 = 0.0
- t1 = 0.05
- Timestamp = 0.3

The client will receive the replay at time = 0.65 and will give value to the following variables:

- t2 = 0.05
- t3 = 0.30
- Hc (t1) = 0.0
- Hc (t4) = 0.65

Now, it will calculate the values of "ρUp" and "ρLow":

- rhoUp = 2.60
- rhoLow = 1.45

Next, it will calculate the value of the hoperaa execution interval, which is the variable responsible of determining when to execute Hoperaa again:

- Since both "rhoUp" and "rhoLow" larger than 1, the hoperaa execution Interval is 0.162441 (Look at chapter 3 for more detail of how this result was obtained.)

Finally, it will adjust the value of the variable Pc and will calculate the value of the hoperaa execution interval if necessary. In this case, since "1 ≤ rhoLow ≤ rhoUp" the updated values of Pc and Hoperaa Execution Interval are:

- hopExeInt = 0.327
- Pc = 0.435

From the above calculation, we can conclude following:

1. The next resynchronization will be at time = 0.65 + 0.327 = 0.977 which means that it will be scheduled in ns at time = 0.977 / 2 = 0.489. (Look at previous section)
2. The original value of Pc was 0.3 (Pc = L) so, this new value is the algorithm's adjustment in order to determine at which rate the Server's Worker ports are being opened.
3. The values of "rhoUp" and "rhoLow" will determine just how big the drift in between the Client and the server's clock is.

Let's assume that we allow Hoperaa to execute for a second time at time = 0.977. At this point, Client 1 will start the "Contact-Initialization" phase by sending packets to all the ports in a randomly chosen interval. These packets will have the following format:

Pkt (*init, timestamp, port***)**

Where timestamp is equal to 0.977; the Server will receive a contact initialization message at time = 0.539 and will wait until the next worker port is opened (t = 0.6). At time 0.60 the Server will send a reply with the information of the first contact initialization message received:

Pkt (*reply,h1, t1, timestamp, seed***)**

Where:

- h1 = 0.0
- t1 = 0.05
- Timestamp = 0.6

Once again, the client will receive the replay at time = 1.25 and will give value to the following variables:

- t2 = 0.05
- t3 = 0.60
- Hc (t1) = 0.0
- Hc (t4) = 1.25

Now, it will calculate the values of "ρUp" and "ρLow":

- rhoUp = 2.27
- rhoLow = 1.66

Next, the value of the hoperaa execution interval will be:

- Since both "rhoUp" and "rhoLow" larger than 1, the hoperaa execution Interval is 0.179

Finally, it will adjust the value of the variable Pc and will calculate the value of the hoperaa execution interval if necessary.
In this case, since "1 ≤ rhoLow ≤ rhoUp" the updated values of Pc and Hoperaa Execution Interval are:

- hopExeInt = 0.62
- Pc = 0.50

Even though Hoperaa was simulated only 2 times in this brief example; based on [5], we can expect the following behavior:

1. If we were to continue with the simulation, we will notice that the Hoperaa Execution Intervals are increasing every time the resynchronization phase is executed; this means that, as the simulation goes on, the Client is capable of sending more and more data before it has to stop to resynchronize with the Server.
2. We also notice that the value of Pc is being updated every time; however, if we were to continue with the simulation we will notice that eventually "Pc ≈ 0.6" The reason for this is because the Client is 2 times faster than the Server and, Pc represents the period of time (for the client's clock) at which the worker ports are being opened on the Server's side. (Roughly speaking such value would be "L * 2" for this case)
3. Finally, the values of "rhoUp" and "rhoLow" represent the drift in between the Client and the Server. If we were to continue with the simulation, we will notice that the range "ρLow ≤ ρ ≤ ρUp" will eventually become closer to 2 which would mean that the Clock from the Client is "ρ = 2" times faster than the Clock from the Server.

We believe that at this point, the reader should have a better understanding of exactly how the algorithm works and also a clearer idea of how all the information, from the previous chapters, fall into place in the simulation. In the next sections we will, extend this scenario as part of a full simulation and endorse, with more concrete results, that the Hoperaa Implementation performs as described.

5

Analysis and Evaluation

" Curiosity begins as an act of tearing to pieces or analysis."

- Samuel Alexander

In chapter 4, we gave an overview about how the HOPERAA was implemented in the simulator; in this chapter, we focus our efforts on studying how the implementation will behave under specific conditions, in order to understand the framework and how is it affected when certain changes are applied. Throughout this chapter we demonstrate, with more concrete results, that the Hoperaa Implementation performs as described in [5] and also, we will present and analyze the results obtained from simulating different scenarios in ns. By using this approach, we will fulfill two important purposes, the first one is to show that the simulation is consistent with the expected behavior from the algorithm and, the second one is to make evident how each one of the elements involved relate and affect each other depending on their different values and particular circumstances; results obtained from each case scenario will be presented, and analyzed, all throughout the following sections.

5.1 Study Case 1: Single Client/Server Scenario

5.1.1 Experiment Specification

As previously mentioned, this first Study Case will be used to endorse with more concrete results that the Hoperaa Implementation performs as described in [5]. In this Scenario we study a first approach of the HOPERAA algorithm by running a simulation of the framework on a single Client/Server scenario. For this we will use a similar configuration as the one presented in the "Example 1".

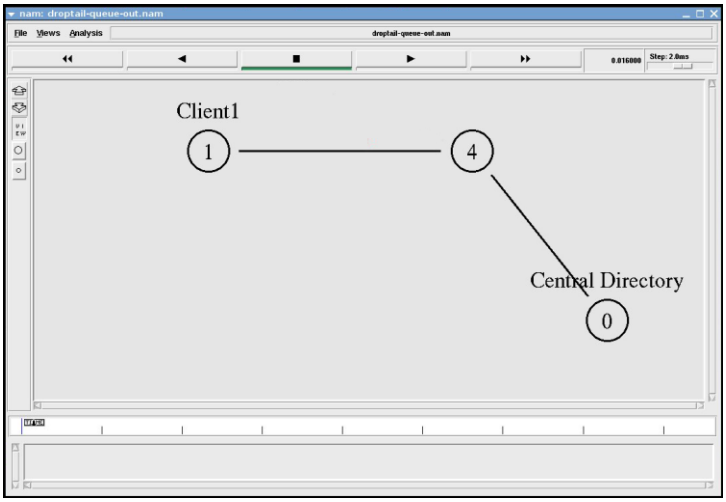


Figure 14: Shows a Single Client/Server topology as presented by NAM using Ns.

Figure 14 presents the topology being used for this case and, as we can see from comparing figure 14 and figure 10, the topology differs a little from the one used previously. For this case, we are using a three node topology in which the nodes labeled "Client 1" and "Central Directory" are the ones communicating with each other using HOPERAA; node 4, the node in between, works only as an intermediary in the communication and no "Hoperaa agent" has been attached to it hence, takes no part in the algorithm; we developed this topology to demonstrate that, due to the natural attributes from NS, we can place as many nodes in between as necessary and still, node "Client 1" will be able to successfully communicate with the "Central Directory".

The function of a node when it receives a packet is to examine the packet's fields, usually its destination address, and on occasion, its source address so, when "Client 1" sends a packet for "Central Directory", the node(s) in between will check the header in the packet received and, will process it accordingly; in this case, such action is to forward it so that it reaches the proper node, "Node 0". Figure 14 differs from the figures used so far because, it shows exactly how the topology looks in NAM when using the results obtained from ns to produce a graphic simulation. From now on we will use this kind of images to present how the scenarios look when implemented in ns.

For this study case, we will use the following specifications:

1. **μ** = 0.1
   Since "μ" is the packet "round-trip maximum delivery latency" this means that the packet takes 0.05 milliseconds to go from point A to point B (in this case "Client 1" to "Central Directory") so, for simulation purposes, the packets takes 0.025 milliseconds to go from Node 1 to Node 4 and then 0.025 milliseconds to go from Node 4 to Node 0.
2. ***L*** = 0.3
   The Server opens Worker Ports every 0.3 milliseconds.
3. $\boldsymbol{\tau} = 1$
   The Server changes "guard ports" every 1 second.
4. A single sequence of 6000 ports, divided into 6 different intervals.

The elements above presented are general for each case but, in order to identify specific behaviors, we will use different study scenarios:

- **Scenario 5.1.1:** Identify how the framework, and the elements derived from its implementation, behaves when the Client is faster than the Server. For this we will run different simulations on which we will vary the value of "$\boldsymbol{\rho}$" as follows:

  - $\boldsymbol{\rho} = 2 \mid 3, \Delta = 0.1$
    When "$\boldsymbol{\rho} = 2$" it means that the Clock from the Client is 2 times faster than the server's; intuitively, if "$\boldsymbol{\rho} = 3$" then the Client is 3 times faster than the Server.

- **Scenario 5.1.2:** This scenario will be somewhat similar to Scenario 5.1.1 but in this case, we will point out how the framework behaves when the Client is slower than the Server. For this we will use:

  - $\boldsymbol{\rho}$ = 0.33 | 0.5, Δ = 0.1
    When "$\boldsymbol{\rho} < 1$" it means that the Client is Slower than the Server hence, "$\boldsymbol{\rho}$ = 0.33" and "$\boldsymbol{\rho}$ = 0.5" represent the cases in which the Client is 2 and 3 times slower than the Server, respectively.

- **Scenario 5.1.3:** In this final approach, we will change the value of delta "Δ" from each simulation. First we will consider the case in which the Client is faster than the Server and latter, another one in which the Client is slower; the results obtained from this scenario will indicate how "Δ"affects the overall process.

  - $\boldsymbol{\rho}$ = 2, Δ= 0.05 | 0.1 | 0.2
  - $\boldsymbol{\rho}$ = 0.5, Δ= 0.05 | 0.1 | 0.2

For Scenario 5.1.3, it is important to mention lemma 5 from [5]:

> *"Using the HOPERAA algorithm, consider that client starts sending data messages to port "p" at time "t" (according to the Server's Clock) and changes the destination port at t' (according to the Server's Clock). Then "t" will not be Δ time units smaller than the corresponding opening time of port "p" by the Server, and t' will not be Δ time units greater than the corresponding closing time of port "p" by the Server".*

The previous statement guarantees that the Client's hopping times will not drift "Δ" time units away from the server's. This statement was presented in [5] as part of their study; however in this case (Scenario 5.3.3), we will focus on understanding how this changes also affects other elements derived from the implementation such as "rhoUp", "rhoLow", "Pc" or the "Hoperaa Execution Intervals" growth.

5.1.2 Results' Analysis: Scenario 5.1.1

For this first case, we will run two different simulations; both simulation will use a "Δ = 0.1"and they will set a common background on how the algorithm behaves under the assumption that the Client's Clock Drift "ρ" is constant and greater than the Server's.
Next, we present table 1 and table 2 with the results obtained from simulating a scenario in which "ρ = 2" and "ρ = 3", correspondingly.

Hoperaa Times	HopExecInt	rhoLow	rhoUp	Pc
1	0	0	0	0.3
2	0.325	1.44467	2.6007	0.4334
3	0.625	1.66682	2.2730	0.5000
4	6.325	1.96126	2.0240	0.5884
5	15.325	1.98383	2.0098	0.5951
6	24.325	1.98978	2.0062	0.5969
7	37.225	1.99331	2.0040	0.5980
8	61.225	1.99593	2.0025	0.5988
9	94.525	1.99736	2.0016	0.5992
10	144.325	1.99827	2.0010	0.5995

Table 1: Shows the results obtained from allowing the simulation to run HOPERAA 9 times, using ρ = 2 and Δ = 0.1.

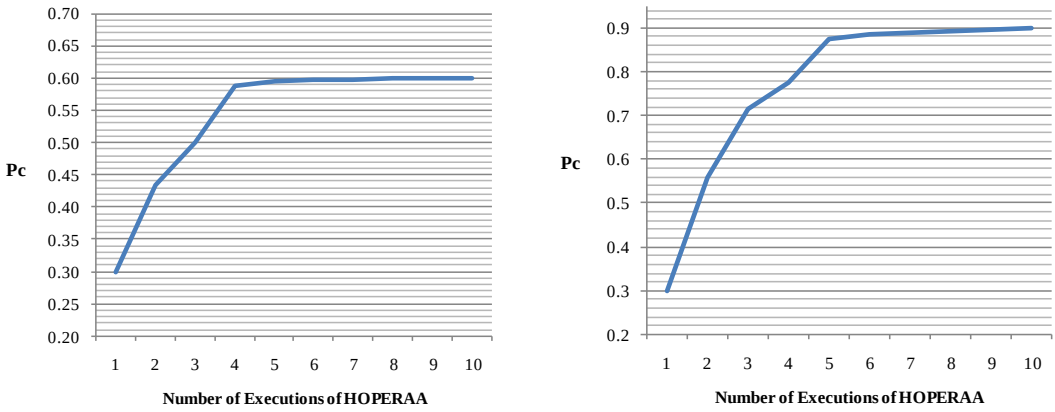
Hoperaa Times	HopExecInt	rhoLow	rhoUp	Pc
1	0	0	0	0.3
2	0.325	1.8573	4.3340	0.5572
3	0.775	2.3847	3.4446	0.7154
4	1.225	2.5790	3.2668	0.7737
5	6.625	2.9121	3.0460	0.8736
6	10.675	2.9448	3.0284	0.8835
7	18.325	2.9676	3.0165	0.8903
8	27.775	2.9786	3.0108	0.8936
9	45.325	2.9868	3.0066	0.8960
10	78.175	2.9924	3.0038	0.8977

Table 2: Shows the results obtained from allowing the simulation to run HOPERAA 9 times, using ρ = 3 and Δ = 0.1.

The tables in this and the remaining sections, they all will be organized in the same manner; the first column represent the number of times HOPERAA was executed, the second one illustrates the value of the "Hoperaa Execution Intervals" derived from each time the Client re-synchronizes with the Server, the third and forth correspond to the values "rhoLow" and "rhoUp" from the range "ρLow ≤ **ρ** ≤ ρUp"; finally, the fifth one shows the value of "Pc" which, as presented earlier, is used when sending application data from the Client to the Server and, should be in between the range "L ≥ **Pc** ≤ ρL" being "L" its original value and "ρL" its ideal. (For more information on what this variables are look at Chapter 3)

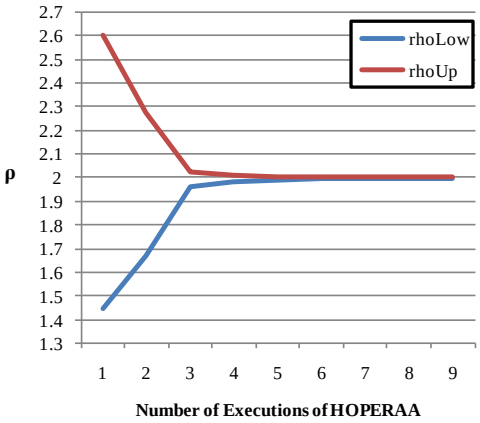
As expected from the algorithms natural definition, we can notice the following:

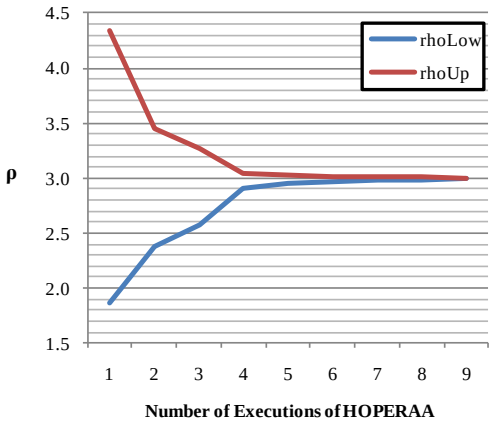
1. The value of "Pc", depending on the case, is slowly being adjusted to its "ideal value" 0.6 or 0.9. We can denote this behavior on Graph 1 and 2, "ρ=2" and "ρ=3" respectively, in which we can see that the lines are growing dramatically throughout the first times HOPERAA is executed and, the increment becomes less obvious as they come closer to the values 0.6 and 0.9 correspondingly.



Graph 1 (at the Left) presents the "Pc" grow rate for a simulation with "ρ = 2"
Graph 2 (at the right) presents the same grow rate but for a simulation where "ρ = 3".

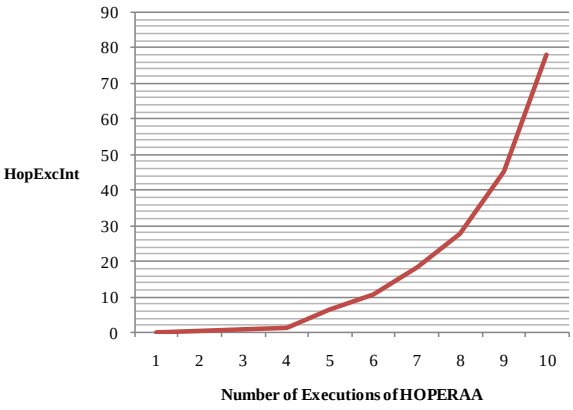
2. From tables 1 and 2 and, observing the lines plotted in graphs 3 and 4, we can conclude that "rhoLow" and "rhoUp" both remain in between the range "ρLow ≤ **ρ** ≤ ρUp" and, effectively approximating to the values defined for each simulation; in Graph 3 we observe that the range slowly closes around 2 and, in Graph 4 we note the same behavior although the range approximates to 3.

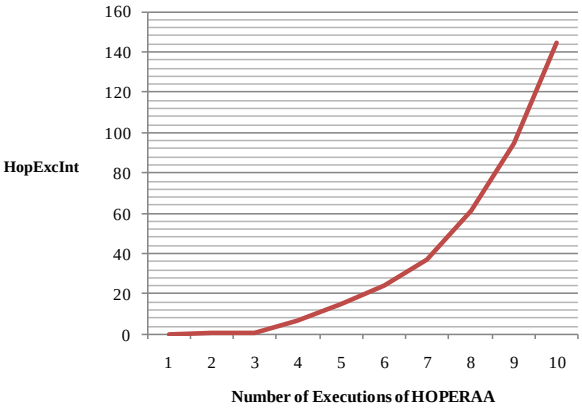




Graph 3 (at the Left) presents the values of "rhoLow" and "rhoUp" as the simulation progresses, when "ρ = 2"
Graph 4 (at the right) presents the same values of "rhoLow" and "rhoUp" but for a simulation where "ρ = 3".

3. Regardless of the value of "ρ", by looking at Graph 5 and 6 we can observe that the "Hoperaa Execution Intervals" increase exponentially, as defined in [5], every time the resynchronization phase is executed.





Graph 5 (at the Left) presents the "Hoperaa Execution Interval" grow rate for a simulation with "ρ = 3"
Graph 6 (at the right) presents the same grow rate but for a simulation where "ρ = 2".

5.1.3 Results' Analysis: Scenario 5.1.2

In this instance, we use a similar configuration as the one used in the previous scenario but now, we consider that the Client's Clock Drift "ρ" is constant and smaller than the Server's. For this, we present table 3 and table 4 with the results obtained from the simulation:

Hoperaa Times	HopExecInt	rhoLow	rhoUp	Pc
1	0	0	0	0.3
2	0.100	0.4445	0.8001	0.2400
3	0.250	0.4762	0.5882	0.1765
4	3.475	0.4982	0.5055	0.1516
5	10.300	0.4994	0.5018	0.1505
6	16.300	0.4996	0.5012	0.1503
7	29.800	0.4998	0.5006	0.1502
8	53.500	0.4999	0.5004	0.1501
9	81.550	0.4999	0.5002	0.1501
10	127.150	0.5000	0.5001	0.1500

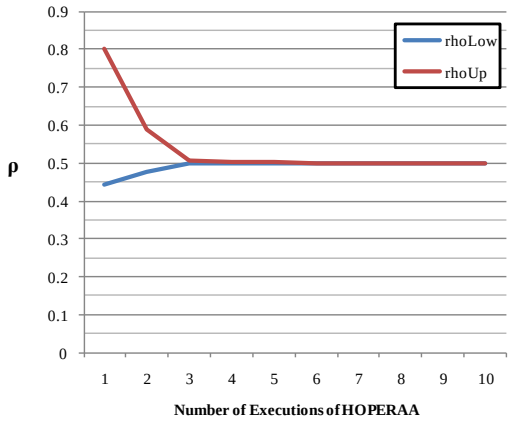
Table 3: Shows the results obtained from a topology in which the Client is 2 times slower than the Server, ρ = 0.5 and Δ = 0.1.

Hoperaa Times	HopExecInt	rhoLow	rhoUp	Pc
0	0	0	0	0.3
1	0.075	0.3334	0.6001	0.3
2	0.175	0.3334	0.4118	0.1800
3	2.825	0.3333	0.3373	0.1235
4	14.275	0.3333	0.3341	0.1012
5	26.975	0.3333	0.3337	0.1002
6	42.775	0.3333	0.3336	0.1001
7	69.225	0.3333	0.3335	0.1001
8	105.875	0.3333	0.3334	0.1000
9	159.875	0.3333	0.3334	0.1000

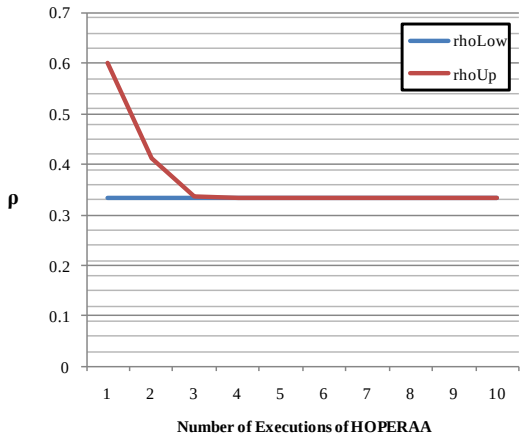
Table 4: Shows the results obtained from a topology in which the Client is 3 times slower than the Server, ρ = 0.33 and Δ = 0.1.

From table 3 and table 4, we notice the following:

1. In the previous case, we observed that when the Client was faster than the Server, "rhoLow" and "rhoUp" were greater than 1 and slowly being adjusted to "the number of times the Client was faster than the Server". In this case, we can see that "rhoLow" and "rhoUp" are smaller than 1 and, are slowly approaching to the value "1/number of times the Client is slower than the Server". The latter is evident when looking at graphs 7 and 8; if the Client is 2 times slower than the Server then "rhoLow ≤ **0.5** ≤ rhoUp" since "1/2 = 0.5"and, "rhoLow ≤ **0.33** ≤ rhoUp", when the Client is 3 times slower than the Server, since "1/3 = 0.33". We also notice that when the Client is slower, the value of "rhoLow", in comparison to the value of "rhoUp", requires little adjustment.

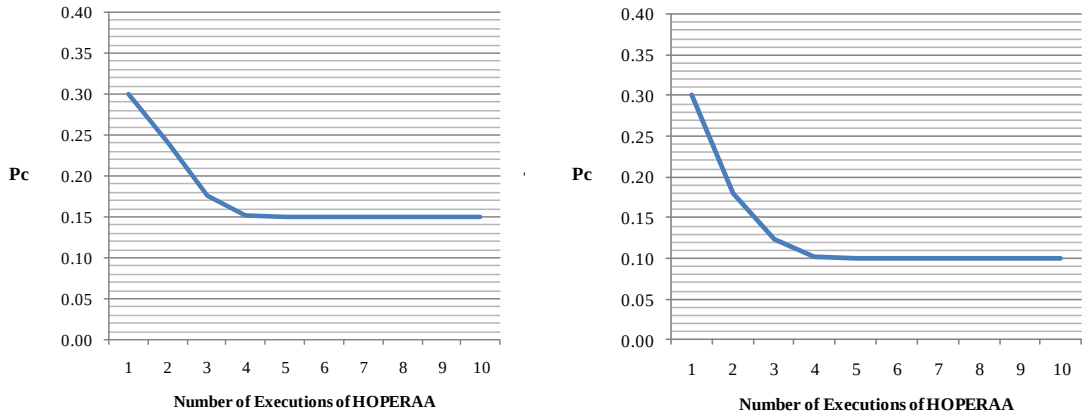


Graph 7 presents the values of "rhoLow" and "rhoUp" as the simulation progresses, when the Client is 2 times slower than the Server.



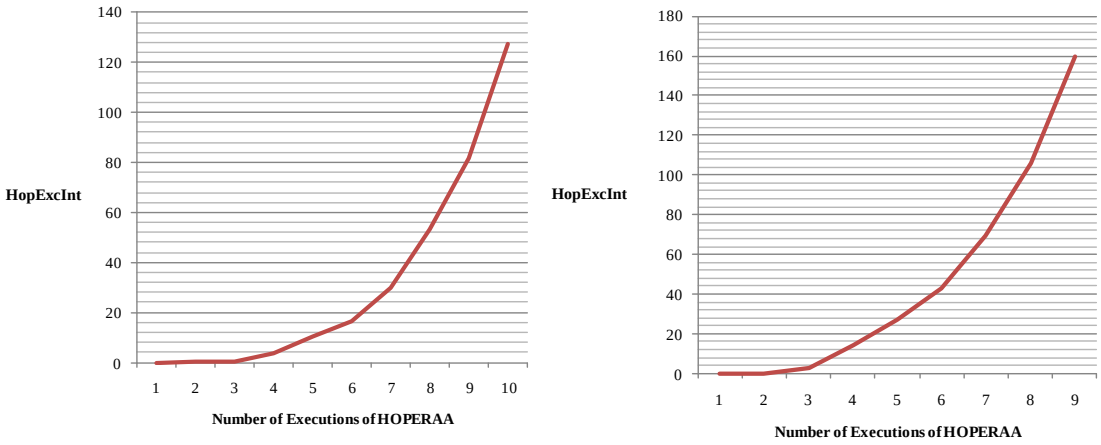
Graph 8 presents the values of "rhoLow" and "rhoUp" as the simulation progresses, when the Client is 3 times slower than the Server.

2. Using tables 3 and 4, we know what the ideal value of "**ρ**" should be so, by looking at graph 9 and 10 we can conclude that the value of "Pc", depending on the case, is slowly being adjusted to its "ideal value = ρL"; when the Client is 2 times slower Pc should approximate to "0.5(0.3) = 0.15" and, when the Client is 3 times slower then Pc should slowly be adjusted to "$0.\overline{33}$ (0.3) = 0.1".



Graph 9 (at the Left) presents the "Pc" growth rate for a simulation with "ρ = 0.5"
Graph 10 (at the right) presents the same grow rate but for a simulation where "ρ = 0.33"

3. Just as observed in the previous scenario, regardless of the value of "ρ" the "Hoperaa Execution Intervals" increase exponentially every time the resynchronization phase is executed.



Graph 11 (at the Left) presents the "Hoperaa Execution Interval" growth rate for a simulation with "ρ = 0.5"
Graph 12 (at the right) presents the same grow rate but for a simulation where "ρ = 0.33".

5.1.4 Results' Analysis: Scenario 5.1.3

For this Scenario we varied the value of delta "Δ" from each simulation, in order to indicate how this change affects the overall process. Next, we present the results obtained from running different simulations, each one of them using a different value of delta ("Δ"). First we present the values obtained when Client is 2 times faster than Server.

Hoperaa Times	HopExecInt	rhoLow	rhoUp	Pc
0	0	0	0	0.3
1	0.163	1.4450	2.6017	0.4335
2	0.313	1.6670	2.2734	0.5001
3	0.463	1.7622	2.1769	0.5287
4	1.663	1.9276	2.0463	0.5783
5	3.163	1.9613	2.0241	0.5884
6	4.213	1.9708	2.0180	0.5912
7	5.563	1.9778	2.0136	0.5933
8	7.063	1.9825	2.0107	0.5947
9	9.163	1.9865	2.0082	0.5959
10	11.713	1.9894	2.0064	0.5968

Table 5: Shows the results obtained from allowing the simulation to run HOPERAA 10 times, using ρ = 2 and Δ = 0.05.

Hoperaa Times	HopExecInt	rhoLow	rhoUp	Pc
0	0	0	0	0.3
1	0.325	1.4447	2.6007	0.4334
2	0.625	1.6668	2.2730	0.5001
3	1.225	1.8149	2.1306	0.5445
4	2.425	1.9020	2.0639	0.5706
5	3.925	1.9383	2.0390	0.5815
6	6.025	1.9594	2.0252	0.5878
7	10.225	1.9759	2.0148	0.5928
8	15.625	1.9841	2.0097	0.5952
9	24.325	1.9898	2.0062	0.5969
10	36.625	1.9932	2.0041	0.5980

Table 6: Shows the results obtained from allowing the simulation to run HOPERAA 10 times, using ρ = 2 and Δ = 0.1.

Hoperaa Times	HopExecInt	rhoLow	rhoUp	Pc
0	0	0	0	0.3
1	0.450	1.2859	3.0011	0.3858
2	1.050	1.6155	2.3336	0.4847
3	2.850	1.8388	2.1112	0.5516
4	6.450	1.9254	2.0477	0.5776
5	13.650	1.9641	2.0222	0.5892
6	28.050	1.9823	2.0108	0.5947
7	56.850	1.9913	2.0053	0.5974
8	114.450	1.9956	2.0026	0.5987
9	229.650	1.9978	2.0013	0.5994
10	469.050	1.9989	2.0006	0.5997

Table 7: Shows the results obtained from allowing the simulation to run HOPERAA 10 times, using ρ = 2 and Δ = 0.2.

Now, table 8 to 10 present the values obtained in a simulation where the Client is 2 times slower than Server.

Hoperaa Times	HopExecInt	rhoLow	rhoUp	Pc
0	0	0	0	0.3
1	0.050	0.4446	0.8005	0.2401
2	0.538	0.4943	0.5181	0.1554
3	1.213	0.4974	0.5079	0.1524
4	1.663	0.4981	0.5057	0.1517
5	2.750	0.4989	0.5034	0.1510
6	4.475	0.4993	0.5021	0.1506
7	6.275	0.4995	0.5015	0.1505
8	8.150	0.4996	0.5012	0.1503
9	11.113	0.4997	0.5008	0.1503
10	14.188	0.4998	0.5007	0.1502

Table 8: Shows the results obtained from allowing the simulation to run HOPERAA 10 times, using ρ = 0.5 and Δ = 0.05.

Hoperaa Times	HopExecInt	rhoLow	rhoUp	Pc
0	0	0	0	0.3
1	0.075	0.4287	1.0005	0.3000
2	0.225	0.4737	0.6001	0.1800
3	2.175	0.4971	0.5088	0.1526
4	3.375	0.4982	0.5056	0.1517
5	5.625	0.4989	0.5034	0.1510
6	9.750	0.4994	0.5019	0.1506
7	16.725	0.4996	0.5011	0.1503
8	28.200	0.4998	0.5007	0.1502
9	42.375	0.4999	0.5004	0.1501
10	64.650	0.4999	0.5003	0.1501

Table 9: Shows the results obtained from allowing the simulation to run HOPERAA 10 times, using $\rho = 0.5$ and $\Delta = 0.1$.

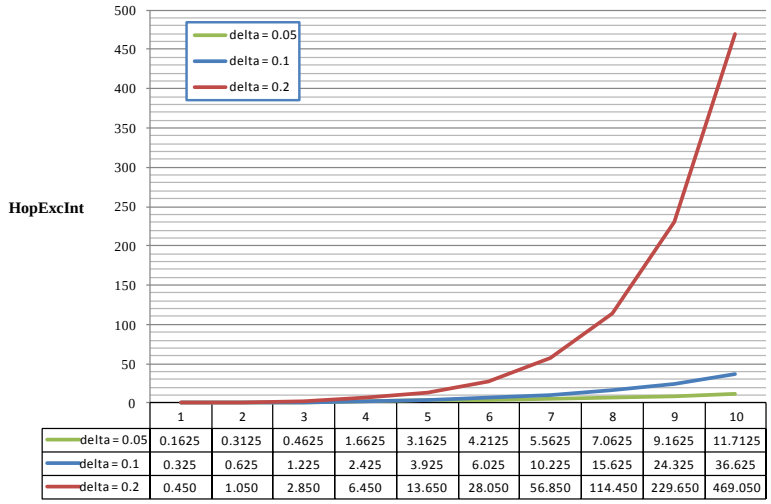
Hoperaa Times	HopExecInt	rhoLow	rhoUp	Pc
0	0	0	0	0.3
1	0.200	0.4446	0.8005	0.2402
2	0.500	0.4763	0.5884	0.1765
3	1.100	0.4889	0.5366	0.1610
4	3.350	0.4963	0.5115	0.1534
5	8.300	0.4985	0.5046	0.1514
6	34.250	0.4996	0.5011	0.1503
7	71.150	0.4998	0.5005	0.1502
8	164.900	0.4999	0.5002	0.1501
9	341.000	0.5000	0.5001	0.1500
10	692.150	0.5000	0.5001	0.1500

Table 10: Shows the results obtained from allowing the simulation to run HOPERAA 10 times, using $\rho = 0.5$ and $\Delta = 0.2$.

It is important to mention that, based on the results previously presented and the ones from these tables, we notice that at the beginning every adjustment results on a severe change on the value from each variable and, as the simulation progresses, the adjustment becomes less radical; this is because, the algorithm is designed to improve with each execution; therefore, in the early stages the adjustment is very notorious since the original value is far from the ideal but, as the simulation progresses, the values from "rhoUp", "rhoLow" and "Pc" they all approach closer to the ideal value making the changes less apparent.

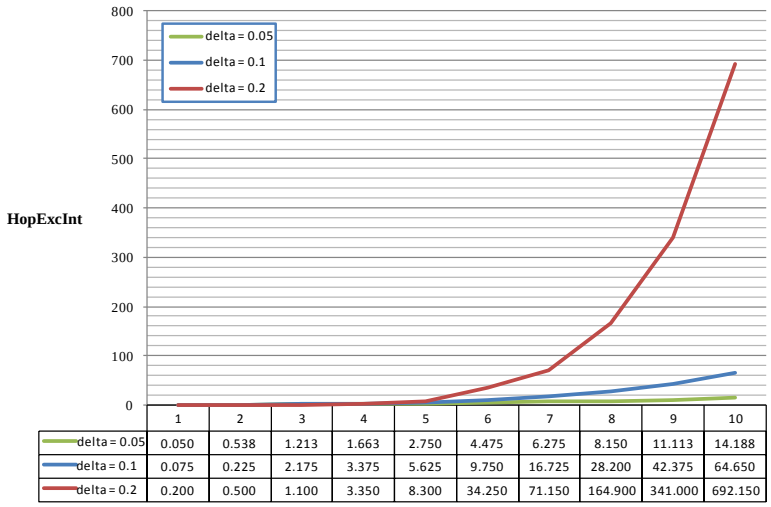
Now, using Graph 13 and 14 we can observe that, independently of the value of "**ρ**":

1. The "Hoperaa Execution Intervals" increase exponentially every time the resynchronization phase is executed; which allows the Client to send more data before it has to stop to resynchronize with the Server.
2. From the formula used to derive the "Hoperaa Execution interval" (see Chapter 3), the value of the latter is intrinsically related to that of Delta ("Δ") thus; the smaller Delta ("Δ") is, the slower the "Hoperaa Execution Intervals" grow.



	1	2	3	4	5	6	7	8	9	10
delta = 0.05	0.1625	0.3125	0.4625	1.6625	3.1625	4.2125	5.5625	7.0625	9.1625	11.7125
delta = 0.1	0.325	0.625	1.225	2.425	3.925	6.025	10.225	15.625	24.325	36.625
delta = 0.2	0.450	1.050	2.850	6.450	13.650	28.050	56.850	114.450	229.650	469.050

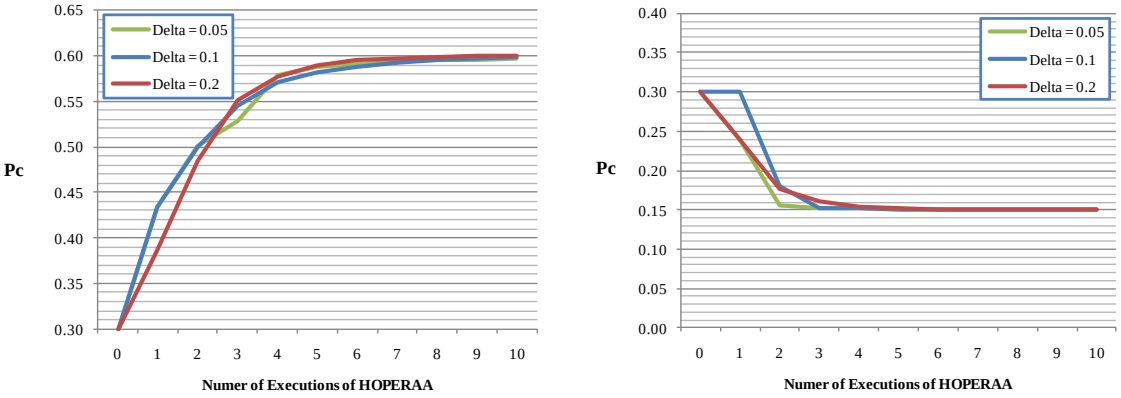
Graph 13 presents a comparison of the "Hoperaa Execution Interval" values from each simulation when ρ = 2 and, Δ is equal to 0.05, 0.1 and 0.2



	1	2	3	4	5	6	7	8	9	10
delta = 0.05	0.050	0.538	1.213	1.663	2.750	4.475	6.275	8.150	11.113	14.188
delta = 0.1	0.075	0.225	2.175	3.375	5.625	9.750	16.725	28.200	42.375	64.650
delta = 0.2	0.200	0.500	1.100	3.350	8.300	34.250	71.150	164.900	341.000	692.150

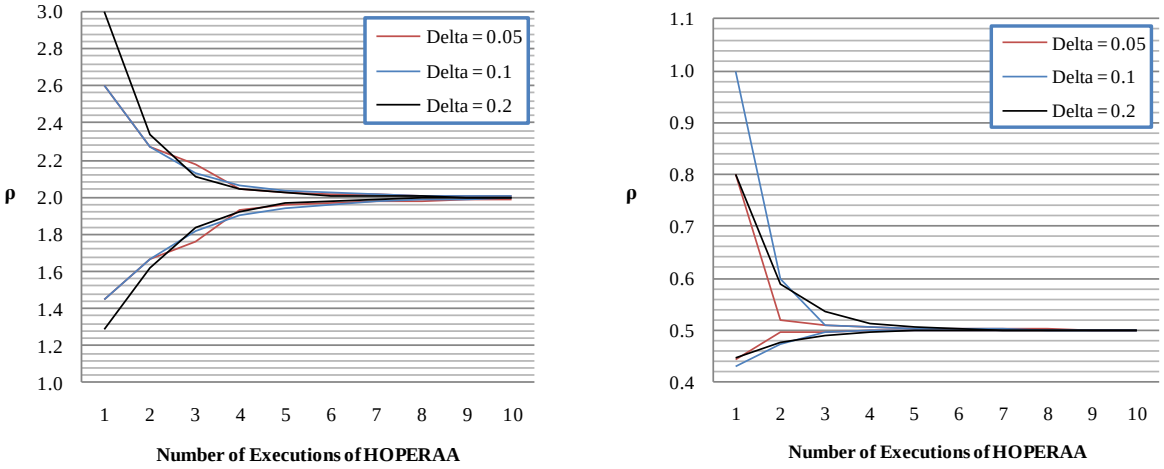
Graph 14 presents a comparison of the "Hoperaa Execution Interval" values from each simulation when ρ = 0.5 and, Δ is equal to 0.05, 0.1 and 0.2

One of the advantages that we have by simulating the algorithm, rather than implementing it in the "real world", is that we know the exact value of "ρ", which represents the clock's drift in between the Client and the Sever. Using graph 15 and 16 as a reference, we can conclude that in each case, the value of Pc continuously becomes closer to its ideal value "0.6" and "0.15" depending on the case and regardless of the value of "Δ".



Graph 15 (at the Left) presents a comparison of how "Pc" is adjusted, depending on the value of "Δ", when ρ = 2.
Graph 16 (at the Right) presents a comparison of how "Pc" is adjusted, depending on the value of "Δ", when ρ = 0.5.

Using Graph 17 and 18, we can show that the range "ρLow ≤ ρ ≤ ρUp" is eventually enclosing around its ideal value "2" and "0.5" regardless of the value of "Δ". In the mentioned graphs, the values from "rhoUp" and "rhoLow" are represented by the lines above and below 2.0 and 0.5, depending on the graph and the value of "ρ".



Graph 17 (at the Left) and 18 (at the Right) present a comparison of how "rhoUp" and "rhoLow" are adjusted, depending on the value of "Δ", when ρ = 2 and ρ = 0.5, correspondingly.

Thanks to the results presented in this section, we can corroborate:

1. That our implementation, behaves as described in [5].
2. That the description given at the end of the previous chapter (section "Example 1") holds as predicted.
3. The HOPERAA algorithm's behavior and, how the values of "rhoUp", "rhoLow" and "Pc" (derived from the framework's execution) are adjusted throughout the simulation when the Client is faster than the server and, vice versa.
4. How different values of "Δ" affect the overall behavior of the simulation in the different stages of the algorithm.
5. From the results obtained in Scenario 5.1.3, and looking at graphs 13 and 14; we can conclude that, out of all the variables derived from implementing the framework, the "Hoperaa Execution Interval" is the only one notoriously affected by the value of Delta ("Δ").

5.2 Study Case 2: Variable Clock Drifts

5.2.1 Experiment Specification

So far we have assumed that Clock Drift "ρ" in between the Client and the Server remains constant throughout the whole communication process however; although the following was never considered or assumed in the original model [5], in this work we wanted to study how the algorithm would behave when the clock drift from the Client changes unexpectedly at some point during the transmission hence, it is no longer constant.
For the latter, we will use a similar scenario to the one used in the "Study Case 1" nevertheless, in this case the Client will start a communication with the Server by having a particular Clock Drift "ρ" and, after HOPERAA has been executed a specific number of times, the drift will change to "ρ + 1" for the rest of the communication. For example, we can say that for the first 5 times HOPERAA is executed the Clock Drift will be "2" and, from the 6^{th} time on, the drift will increase to "3" and will remain like that until the simulation has finished. The latter will allow us to denote and study how the algorithm behaves under these circumstances and what kinds of changes are observed in the different variables involved such as "rhoLow", "rhoUp", "Pc" and the "Hoperaa Execution Intervals". First we will consider the case in which the drift "ρ" changes from 2 to 3; then we will study the inverse scenario, in which the drift "ρ" is originally 3 and then changes to 2; to finally consider the extreme case in which the drift "ρ" changes from 2 to 3 and then from 3 to 4 to remain as 4 throughout the whole communication process. As previously discussed, we will use a topology as the one shown in figure 14 and using the following specifications for the simulation:

1. $\boldsymbol{\rho} = 2 \mid 3 \mid 4$
   It means that the existent "Clock Drift" in between the Client and Server will vary among these values. We are using the number of times HOPERAA is executed, in order to determine when and which value of "ρ" will be used for the rest of the communication process.
2. $\boldsymbol{\mu} = 0.1$
3. $\boldsymbol{L} = 0.3$
4. $\boldsymbol{\tau} = 1$
5. $\Delta = 0.1$
6. A single sequence of 1800 ports, divided into 9 different intervals.

5.2.2 Results' Analysis

As stipulated previously, with this Study Case we want to study how the algorithms performs when the assumed Clock Drift "ρ", in between the Client and the Server, does not remains constant throughout the whole communication process.
Thanks to the results obtained in "Study Case 1" we have established a common baseline of what kind of behavior is expected from each situation, considering "ρ" as a constant variable now, let's observe what happens when such condition is not longer imposed and "ρ" changes sometime along the simulation. In this section we will study the following simulations:

1. The Client and Server have an original Clock Drift "ρ = 2" during the first 6 times HOPERAA is executed and then, the Clock Drift increases to "ρ = 3" for the rest of the simulation.
2. This Simulation is similar to the previous but in this case, the original Clock Drift is "ρ = 3" and, after the 6th time HOPERAA is executed, it decreases to "ρ = 2".
3. For the final case, we will consider a scenario in which the original Clock Drift is "ρ = 2", after the 5th time HOPERAA is executed it changes to "ρ = 3" and finally, after the 10th time HOPERAA is executed, it changes to "ρ = 4".

Tables 11, 12 and 13 present the values obtained for each one of the scenarios above described.

Hoperaa Times	HopExecInt	rhoLow	rhoUp	Pc
1	0.150	1.0008	3.0003	0.3002
2	0.325	1.4451	2.6020	0.4335
3	0.625	1.6671	2.2735	0.5001
4	7.332	1.9665	2.0207	0.5900
5	18.132	1.9863	2.0083	0.5960
6	50.585	1.9951	2.0030	0.5985
7	127.939	3.1551	3.1629	0.9465
8	208.495	3.0946	3.0992	0.9284
9	346.018	3.0552	3.0579	0.9166
10	512.241	3.0356	3.0374	0.9107
11	831.023	3.0229	3.0240	0.9069
12	1298.905	3.0150	3.0157	0.9045

Table 11: Shows the results obtained from allowing the simulation to run HOPERAA 6 times using ρ = 2, another 6 with ρ = 3, and Δ= 0.1.

Hoperaa Times	HopExecInt	rhoLow	rhoUp	Pc
1	0.207	1.4009	4.3340	0.4203
2	0.475	2.1119	3.8026	0.6336
3	0.925	2.4672	3.3647	0.7402
4	1.525	2.6526	3.2111	0.7958
5	29.410	2.9798	3.0103	0.8940
6	48.549	2.9877	3.0062	0.8963
7	49.827	1.8641	1.8711	0.5592
8	85.976	1.9206	1.9249	0.5762
9	153.061	1.9549	1.9574	0.5865
10	243.049	1.9712	1.9728	0.5914
11	392.674	1.9811	1.9821	0.5943
12	658.823	1.9879	1.9885	0.5964

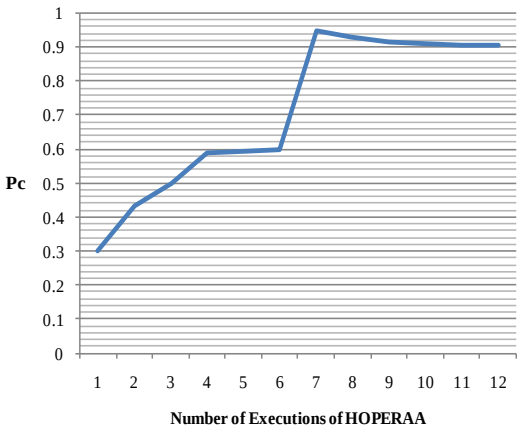
Table 12: Shows the results obtained from allowing the simulation to run HOPERAA 6 times using ρ = 3, another 6 with ρ = 2, and Δ= 0.1.

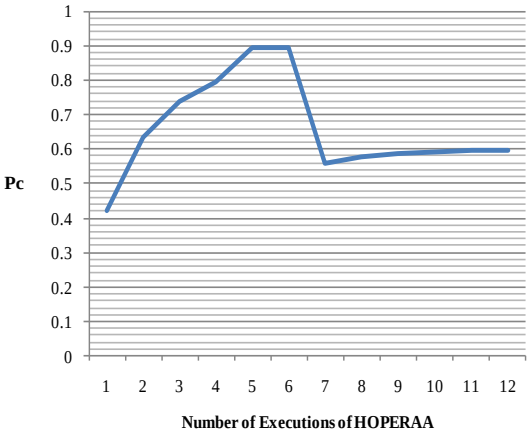
Hoperaa Times	HopExecInt	rhoLow	rhoUp	Pc
1	0.325	1.4446	2.6003	0.4334
2	0.625	1.6667	2.2729	0.5
3	1.225	1.8149	2.1305	0.5445
4	2.425	1.902	2.0639	0.5706
5	3.925	1.9383	2.039	0.5815
6	14.255	3.4106	3.4942	1.1732
7	21.945	3.3103	3.361	1.0231
8	40.846	3.2209	3.2465	0.9663
9	61.292	3.143	3.1592	0.9429
10	93.225	3.0936	3.1039	0.9281
11	199.878	4.1176	4.1261	1.2353
12	308.161	4.0766	4.082	1.223
13	468.886	4.0493	4.0528	1.2148
14	707.511	4.0328	4.0351	1.2098
15	1078.727	4.0218	4.0233	1.2065

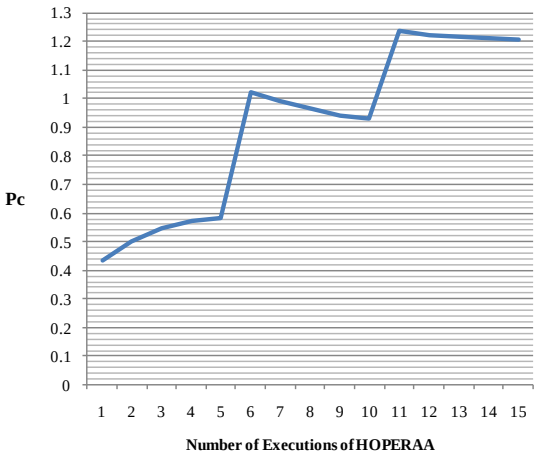
Table 13: Shows the results obtained from allowing the simulation to run HOPERAA 5 times using ρ = 2, 5 times more with ρ = 3 and 5 last times with ρ = 4; using, Δ= 0.1.

As we can see from the tables 11, 12 and 13, it is very easy to denote the interval in which the Clock Drift is changing by looking at how the values of "rhoUp", "rhoLow" and "Pc" increment and get adjusted; although it might seems that the algorithm is behaving as expected, we can conclude that there are some slight differences in comparison to the cases in which "ρ" remained constant; these differences are the following:

1. Despite of the number given to "ρ", we can conclude that "Pc" will continue approximating to its ideal value in any case; this is evident, when looking at the values in each table. In the graphs 19, 20 and 21, we can see that every time "ρ" changes, the value of "Pc" slowly starts to adjust into becoming 0.6, 0.9 or 1.2, based on the value of "ρ". Using the graphs below we can notice how "Pc" at the beginning is being adjusted to a particular value but, as the simulation progresses and the value of "ρ" is modified, it changes the adjustment from one value to the other.





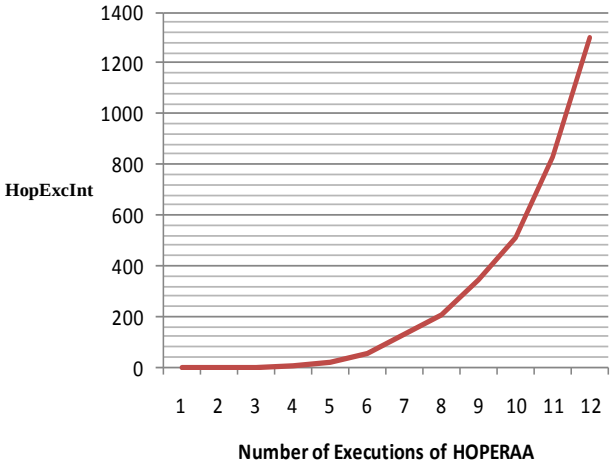


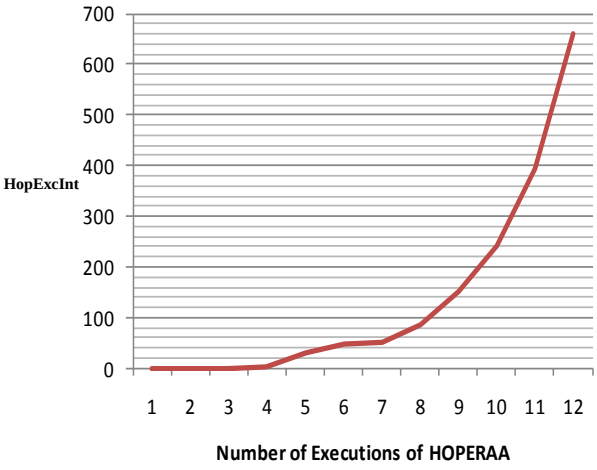
Graph 19 (Top Left): Represents the values of Pc when allowing the Client to change from ρ = 2 to ρ = 3 after HOPERAA was executed for 6 times.

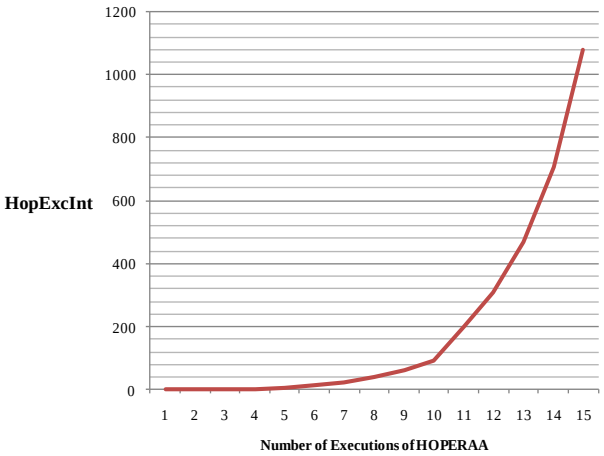
Graph 20 (Top Right): Represents the values of Pc when allowing the Client to change from ρ = 3 to ρ = 2 after HOPERAA was executed for 6 times.

Graph 21 (Bottom Left): Represents the values of Pc when allowing the Client to change from ρ = 2 to ρ = 3 after HOPERAA was executed for 5 times, and then to ρ = 4 after HOPERAA was executed for 10 times.

2. We can perceive that the change in "ρ" affected the rate at which the "Hoperaa Execution Interval" incremented. In every case it is very noticeable that whenever the Clock Drift changed to a higher number, the "Hoperaa Execution Interval" incremented dramatically (look at Table 11 and 13) in comparison to the case when "ρ" decreased from 3 to 2 (Look at table 12) where, although there was also an increment, it was more subtle. Remembering Chapter 3, we know that in this case, because of the value of "ρ", the HOPERAA execution interval is set to "(ρUp)(ρLow)(Δ) / ρUp – ρLow". By looking at the formula, it is easier to understand that sudden change of "rhoUp" and "rhoLow", is the reason for the behavior observed every time "ρ" was changed to a higher or lower value.





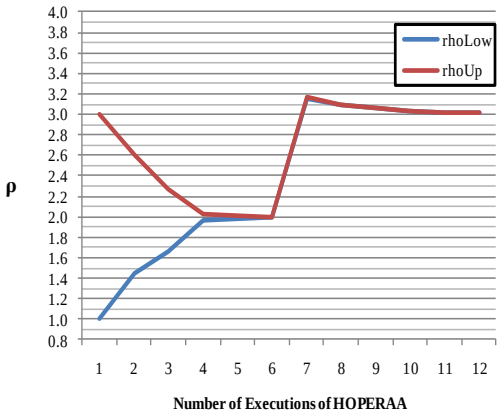


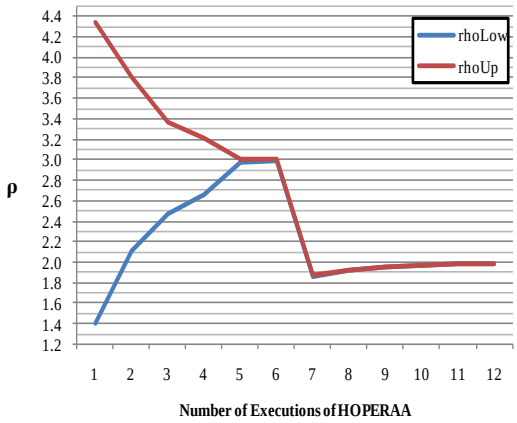
Graph 22 (Top Left): Represents the values of the "Hoperaa Execution Intervals" when allowing the Client to change from ρ = 2 to ρ = 3 after HOPERAA was executed for 6 times.

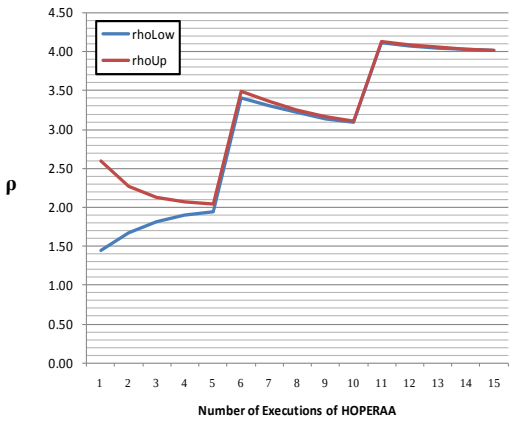
Graph 23 (Top Right): Represents the values of the "Hoperaa Execution Intervals" when allowing the Client to change from ρ = 3 to ρ = 2 after HOPERAA was executed for 6 times.

Graph 24 (Bottom Left): Represents the values of the "Hoperaa Execution Intervals" when allowing the Client to change from ρ = 2 to ρ = 3 after HOPERAA was executed for 5 times, and then to ρ = 4 after HOPERAA was executed for 10 times.

3. This is probably the most essential change that we noticed; by looking at the graphs presented below, we can see that throughout the first HOPERAA executions the values of "rhoUp" and "rhoLow" were in between the range of "ρLow ≤ **ρ** ≤ ρUp" as expected; however when the Clock Drift changed, the values of "rhoUp" and "rhoLow" were in between the range of "**ρ** ≤ ρLow ≤ ρUp" hence breaking the condition, stipulated in [5], for these variables. This behavior is evident in every case no matter what sort of change "ρ" goes through, whether is decreasing or increasing. In spite of this, we can see that eventually the values of "rhoUp" and "rhoLow" slowly are getting adjusted to their ideal values as the simulation progresses. According to the description of the framework, we can speculate that apart from the fact that "rhoUp" and "rhoLow" are no longer in between the range of "ρLow ≤ **ρ** ≤ ρUp", if we were to continue with the simulation we will also notice that the restriction given by "Δ" will no longer hold thus, making the implementation to behave outside of the acceptance margin.







Graph 25 (Top Left): Represents the values of "rhoUp" and "rhoLow" when allowing the Client to change from ρ = 2 to ρ = 3 after HOPERAA was executed for 6 times.

Graph 26 (Top Right): Represents the values of "rhoUp" and "rhoLow" when allowing the Client to change from ρ = 3 to ρ = 2 after HOPERAA was executed for 6 times.

Graph 27 (Bottom Left): Represents the values of "rhoUp" and "rhoLow" when allowing the Client to change from ρ = 2 to ρ = 3 after HOPERAA was executed for 5 times, and then to ρ = 4 after HOPERAA was executed for 10 times.

5.3 Study Case 3: Variable Clock Drifts (2)

5.3.1 Experiment Specification

For this study case we will use a similar scenario as the one used last in the previous section which means that, the Clock Drift "ρ" will change from 2 to 3 and then from 3 to 4 to remain as 4 throughout the whole communication process. Once again, we will study how the algorithm performs when we have the presence of variable Clock Drifts but, unlike the previous time, now we want to show that the reason why the algorithm behave the way it did, whenever we change the value of "ρ", is related to the values of:

1. **"t1"**, time at which the Server received the first contact-initiation message from the Client.
2. **"h1"**, timestamp at which the Client sent the first contact-initiation message.

Taking in consideration what we know from the previous case, let's assume that at the beginning, when "ρ = 2" the values of h1 is "6.0" and the one from t1 is "3.05", under these circumstances we can see that the correlation from both values is "2" since "6.0" is equivalent to "3.0" on the Server's Clock; however, when the Clock Drift changed in the previous studied case, the correlation in between "t1" and "h1" was still "2" regardless of the new value of "ρ", making the algorithm to behave inappropriately.
It is because we are using the same values of "h1", "t1" (regardless of the new value of "ρ") and, because the influence of the delivery latency ("μ") is quite small, which allows the values of "rhoUp" and "rhoLow" to be close to each other and in between the range of "**ρ** ≤ ρLow ≤ ρUp", instead of "ρLow ≤ **ρ** ≤ ρUp"; the latter is pretty obvious when looking at graphs 25 to 27, where we can see that the values of "rhoLow" and "rhoUp" almost overlap each other after the first change of "ρ".
For this study case, we will assume that whenever the value of "rhoLow" gets closer to the value of "rhoUp", as we saw in the graphs, we allow the Client to take for granted that it's Clock Rate has changed and, let the Server know that it has to record new values of "h1" and "t1" in order to compensate for such change. The results derived from the latter change in the algorithm's behavior, are presented in the following subsections.

5.3.2 Results' Analysis

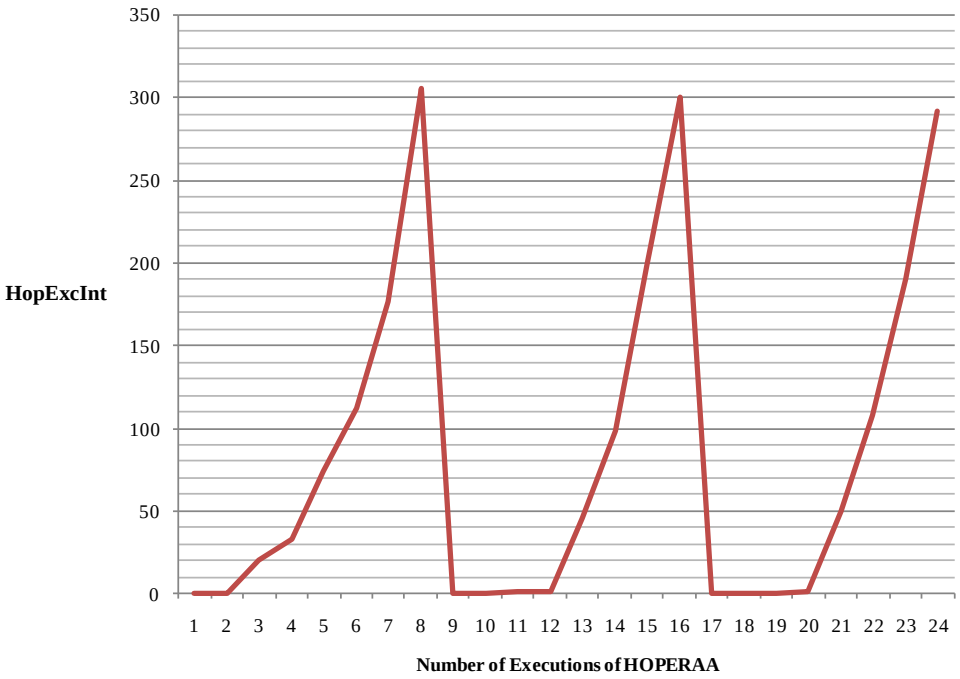
As stipulated previously, we want to study how the algorithms performs when the assumed Clock Drift "ρ", in between the Client and the Server, does not remain constant throughout the whole communication process but also, under the assumption that the Server can record new values of "t1" and "h1" every time "ρ" changes; for this, we will use a similar simulation as the one presented at the end of "Study Case 2". The results obtained in the latter will be used to set a baseline on how the algorithm behaved back then and how does it behaves when this change is applied.

Hoperaa Times	HopExecInt	rhoLow	rhoUp	Pc
1	0.1250	1.0010	5.0233	0.300
2	0.3250	1.4452	2.6024	0.434
3	20.1250	1.9877	2.0075	0.596
4	33.3250	1.9925	2.0045	0.598
5	74.1250	1.9966	2.0020	0.599
6	112.3250	1.9978	2.0013	0.599
7	176.1250	1.9986	2.0009	0.600
8	305.3250	1.9992	2.0005	0.600
9	0.1937	1.4770	6.2213	0.443
10	0.4937	2.1359	3.7648	0.641
11	0.9437	2.4760	3.3567	0.743
12	1.5437	2.6563	3.2084	0.797
13	45.1937	2.9868	3.0067	0.896
14	98.7437	2.9939	3.0031	0.898
15	198.3440	2.9970	3.0015	0.899
16	300.1940	2.9980	3.0010	0.899
18	0.2250	1.8005	9.0116	0.540
19	0.4250	2.4291	5.6692	0.729
20	0.6250	2.7782	5.0013	0.833
21	1.2250	3.2670	4.4551	0.980
22	50.2250	3.9782	4.0100	1.193
23	108.2250	3.9899	4.0046	1.197
24	190.0250	3.9942	4.0026	1.198
25	292.2250	3.9962	4.0017	1.199

Table 14: Shows the results obtained from allowing the simulation to run HOPERAA 8 times using ρ = 2, 8 times more with ρ = 3 and 8 last times using ρ = 4; using, Δ= 0.1.

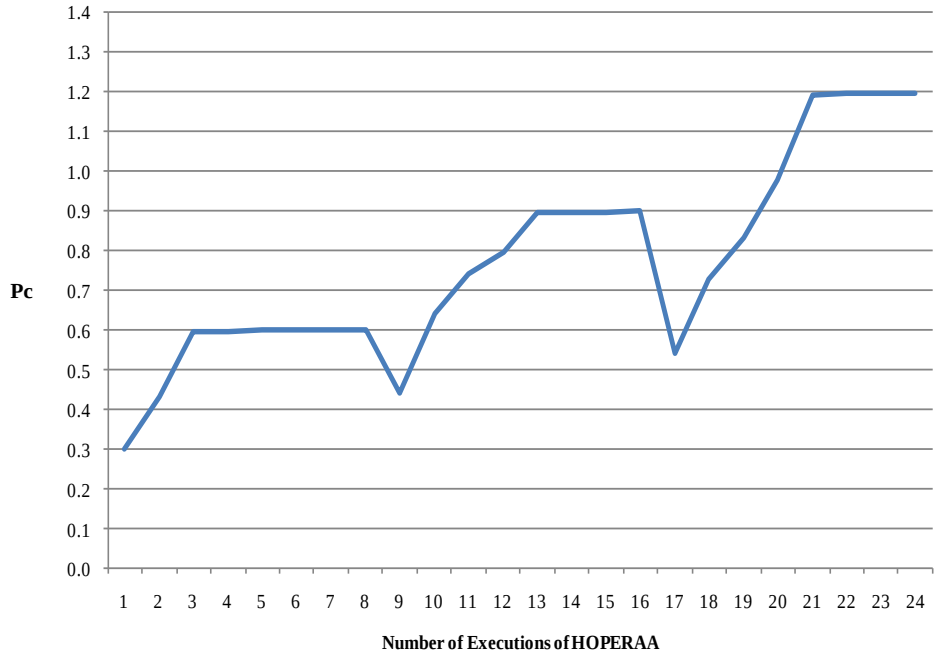
Based on the results presented in Table 14, we can conclude the following:

1. The number of times Hoperaa is executed increases although we are using less amount of simulation time than before. The reason for this is because now, whenever "ρ" changes and the Server records new values of "h1" and "t1", the behavior observed from the algorithm is similar to the one it had when we were using constant values for "ρ". The previous assertion is very noticeable by looking at graph 28 where we can see that when "ρ = 2" the "Hoperaa Execution Interval" grows from 0 to as much as possible and, when "ρ = 3 or 4" it decreases back to 0 and follows the same pattern; unlike the previous case, when we observed that the Intervals grew constantly no matter what value of "ρ" we had.



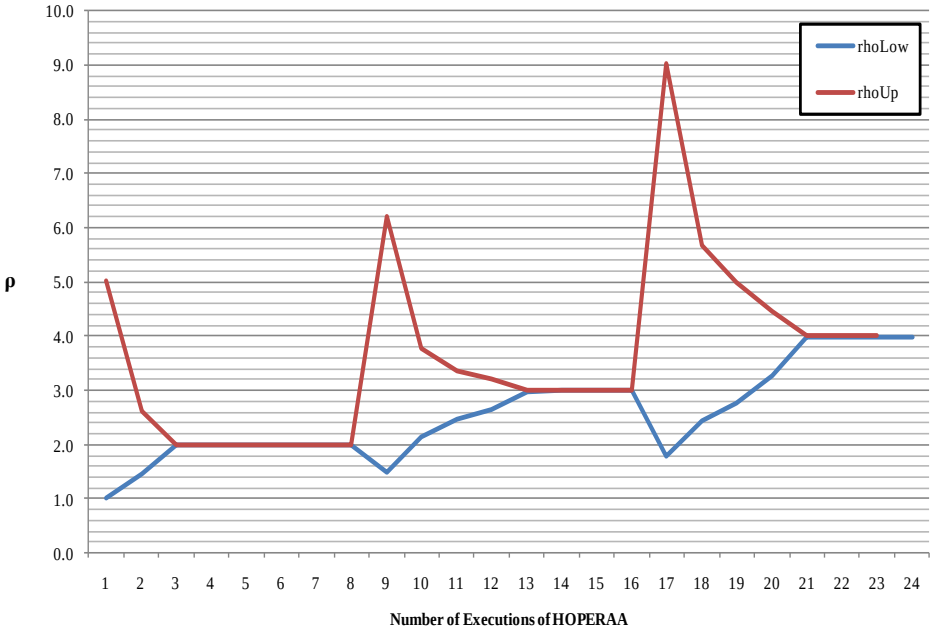
Graph 28: Represents the values of the "Hoperaa Execution Intervals" when allowing the Client's clock drift to change from 2 to 4. (Values obtained from table 14)

2. As we can see from graph 29, "Pc" behaves as expected; its value remains in between the range "$L \geq Pc \leq \rho L$" and its successfully being adjusted to its ideal value "ρL".



Graph 29: Represents the different values of "Pc" when allowing the Client's clock drift to change from 2 to 4. (Values obtained from table 14)

3. Lastly, different to the previous case, we can observe that now no matter how "ρ" increments, the values of "rhoUp" and "rhoLow" will remain in between the range of "ρLow ≤ ρ ≤ ρUp" as defined by the algorithm's definition; this behavior is evident in every case no matter what sort of change "ρ" goes through. (See Graph 30)



Graph 30: Represents the values of "rhoLow" and "rhoUp" when allowing the Client's clock drift to change from 2 to 4. (Values obtained from table 14)

From “Study Case 2” we were able to denote that when “ρ” changes, the framework break two of the conditions established by the algorithm’s definition in [5], as shown previously. Now, thanks to the results presented in this section, we can conclude that when we allow the Server to record new values of “h1” and “t1”, whenever the Clock Drift changes, the algorithm’s behavior will be inside of the acceptance margin and will perform as if it had no record that there were ever a drift change to begin with; the latter allows the variables “rhoLow”, ”rhoUp”, “Pc” and “Hoperaa Execution Interval” to act as defined by the algorithm and, their respective values, to be more comparable to the ones obtained when “ρ” was constant.

5.4 Study Case 4: Framework's overhead

5.4.1 Experiment Specification

For this case we will study a scenario as the one presented in figure 15. The topology in this case is formed by a single Server Node (labeled "Server") and, three different Clients. Although every client has its own link, the traffic from all 3 concentrates on a gateway node which then, forwards it to the Server. This gateway node works only as an intermediary in the communication and takes no part in the algorithm; as previously mentioned, we included this gateway node in order to confirm that, due to the natural attributes from NS, we can place as many nodes in between as necessary and still, the "Client" nodes will be able to successfully communicate with the "Server".

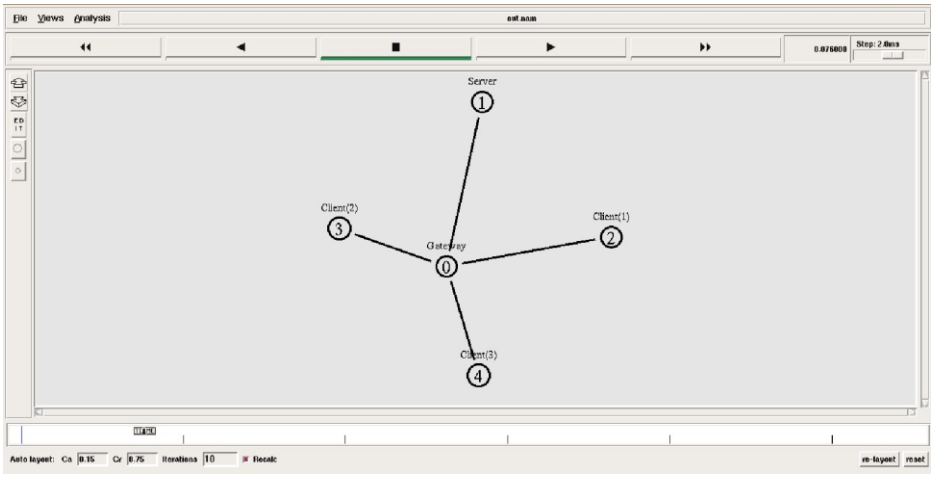


Figure 15: Shows a topology in which 3 Clients communicate with a single server.

Similar to the study cases presented so far, in this instance we will use the following specifications for the simulation:

1. $\boldsymbol{\rho_1} = 2$
   This means that the Clock from the "Client 1" is 2 times faster than the server's.
2. $\boldsymbol{\rho_2} = 0.5$
   This means that the Clock from the "Client 2" is 2 times slower than the server's.
3. $\boldsymbol{\rho_3} = 0.\overline{33}$
   This means that the Clock from the "Client 3" is 3 times slower than the server's.

4. $\boldsymbol{\mu} = 0.1$
5. $\boldsymbol{L} = 0.3$
6. $\boldsymbol{\tau} = 1$
7. $\Delta = 0.1$

In order to study this case, we will use a different approach to the one used so far; for this, we need to define specific and differentiable packet's sizes for each packet being transferred during the simulation:

8. Data Packet Size: 1040
9. HOPERAA Packet Size: 60
10. TCP Handshake Packet Size: 40

Unlike the prior study case, in this occasion we will use different scenarios in order to study the framework's behavior under specific circumstances:

- **Scenario 5.4.1:** Due to the defense framework implementation, data transmission is interrupted to resynchronize with the Server; the latter, makes the client takes longer when sending a specific amount of data. In this scenario, using a fixed time period, we will study the differences (in terms of overall transfer performance) between a topology using the defense framework and another one which doesn't.

- **Scenario 5.4.2:** When implementing the defense framework, data transmission takes longer due in part to the fact that the Client has to resynchronize with the Sever but also, because of the time the Client has to wait, after sending contact-initialization messages, before getting a reply from the Server. In this Scenario, we will study the relation in between the time Clients have to wait for a reply and, the number of sequences being hosted by the Server.

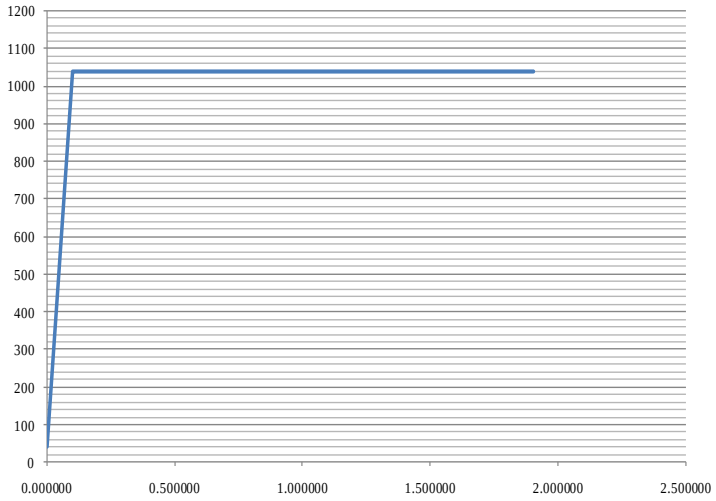
- **Scenario 5.4.3:** As we know by now, when using HOPERAA, data transmission is interrupted so that the Client resynchronizes with the Server thus, affecting the number of data packets sent in every interval. By using this scenario we will study how, as the simulation progresses, the frequency at which the Client has to resynchronize with the Server decreases hence, allowing the Client to send more data before having to stop to resynchronize again.

- **Scenario 5.4.4:** Every time the Client has to resynchronize with the Server, it has to perform the same actions as in the "Contact-Initialization phase" therefore; it means that it has to send "contact-initialization messages" to all the ports from a randomly chosen interval thus, increasing the amount of packets being exchange by the Server and the Client. By studying this scenario, we want to observe the relation in between the number of intervals "k" and, the network's packet overhead (the total amount of packets travelling through the network).

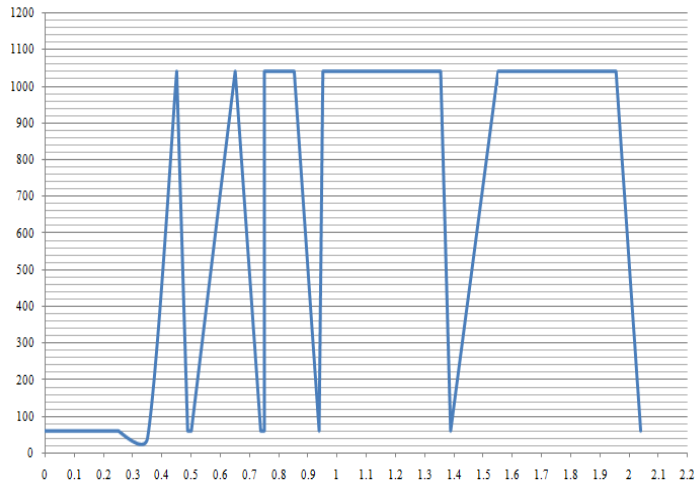
The information gathered from the points above described, will be compared to the one obtained from the baseline case study. In order to establish a proper baseline, first we will create a scenario with a similar composition but, using neither HOPERAA nor BIG WHEEL. Out of all the specification already defined, for this scenario we will use only $\boldsymbol{\mu} = 0.1$ since it is the only feature applicable from one Scenario to the other. The methodology followed in these cases, was to study the information logged by the simulator into the "trace" file and, use Microsoft Excel to create graphs and analyze how every node in the network behaves.

5.4.2 Results' Analysis: Scenario 5.4.1

For this case, first we will focus our attention on studying how a regular network (not using the defense framework) performs. As can be seen in graph 30a, the client starts by sending a 40 bytes packet in order to simulate the handshake phase from TCP. After it has received a reply from the Sever, a session is started and the Client slowly starts sending data packets. As we can see, once the Client receives the reply and starts sending data packets, it will not stop until the transmission has been completed reason why, in the graph we can notice a horizontal line in 1040 which represents an uninterrupted "data packets" transmission.



a)



b)

Graph 31: Represents the relation in between time (X-axis) and, the size of the packets (Y-axis) being transferred.

Now we will use a similar approach but, we will gather information from the network in which the HOPERAA and BIGWHEEL algorithms have been implemented. Graph 31b shows the same information as graph 31a but now, we want to point out how the performance is affected when implementing the defense framework. In this case we are using a network in which the Server is hosting only one single sequence for all three clients. Looking at the graph carefully, we can see that from 0 to 0.3 there is a horizontal line at 60, which represents the size of a "hoperaa packet"; this is because throughout that time, the Client (which is 2 times faster than the Server) is sending "contact-initiation" messages to the Server. At time 0.35 (simulation time), the Client will receive the reply and it will send a 40 bytes packet (in order to establish a session with the Server) and once it receives the reply, it will send data packets (with size 1040 bytes) until the next hoperaa execution interval. This behavior can be seen in graph 31b by looking at the range from 0.0 to 0.5 (X axis).
Every time the Client resynchronizes with the Server, we can see a decrease in the graph from 1040 to 60 which mean that at such point, the data transmission has been stopped and the Client is waiting for a reply from the Server. This behavior is evident throughout the whole simulation but, as described by the algorithm in [5], we can see that after every resynchronization the hoperaa execution intervals increase and the Client is capable of sending more data thus, reducing the overhead on the network's infrastructure.

5.4.3 Results' Analysis: Scenario 5.4.2

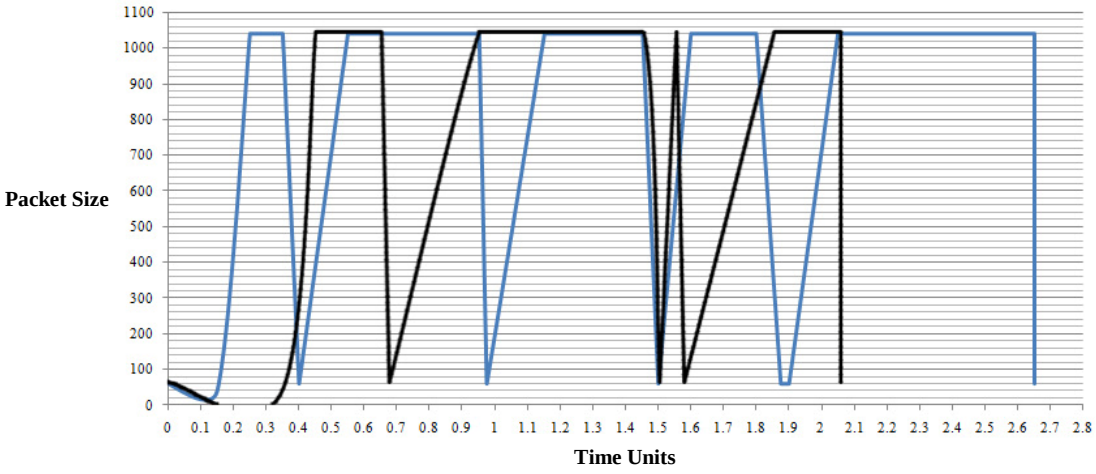
In this case, we want to show that there is a close relationship in between the framework's time overhead and the number of sequences being hosted by the Server. Graph 32 shows the graphical representation of the packet transmission rate for a client in two different situations. The blue line represents the node "client 3" which belongs to a network where the server only has one sequence available while; the other one, represents the same situation but in this instance the Server hosts 3 different sequences instead of just one. By looking closely at both graphs, we can observe that the data transmission is faster, since the client has to wait less time before getting a reply from the Server hence it reaches the "data transmission" phase faster and the overall transmission time is decreased.

Based on the previous, we can conclude in the following:

- Depending on the number of sequences available, as explained in section 4 from [5], the client's waiting time decreases and the maximum waiting time is bounded to the following:

$$\text{Client's Maximum waiting time} = 2\mu + \frac{L}{m}$$

- Based on the data presented in graph 31, we can conclude that the overhead derived from the framework, can be affected by the number of intervals hosted at the server; therefore, the "time overhead" (the total amount of time it takes to complete the transmission) can be reduced if the waiting time bounded to the client is also reduced.



Graph 32: Represents the amount of data packets transmitted from a Client, 3 times slower than the server. Black shows the situation in which the Server hosts 3 sequences and; blue shows the situation in which only one sequence is being hosted.

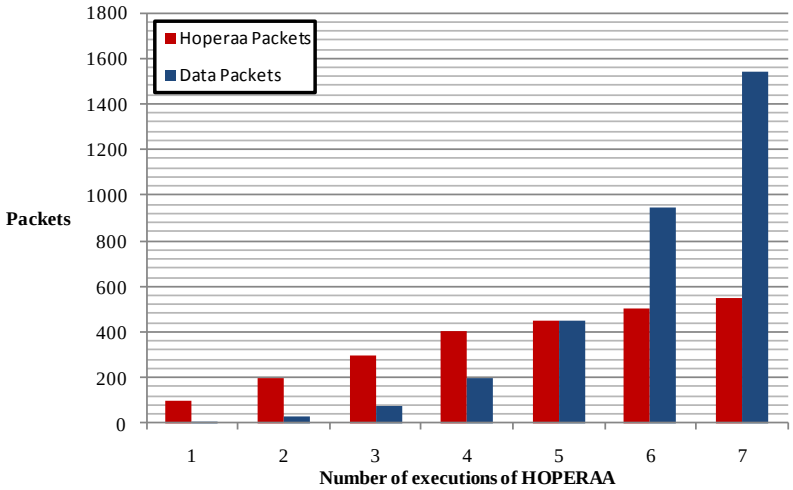
5.4.4 Results' Analysis: Scenario 5.4.3

In this scenario, we will observe more in detail how the data transmission is affected from implementing HOPERAA and BIG WHEEL. For this case, once again we use the behavior, observed in a network where the defense framework was not implemented, as baseline and in order to point out the differences in between both architectures. For this section, we ran several simulations in which we varied the number of times the Client resynchronized with the Server so for example, in graph 33c the first interval represents the number of packets that the a Client sent on a simulation in which HOPERAA was executed only for one time; the second interval, represents the results obtained from another simulation, in which the Client executed HOPERAA for 2 times and so on, up to 7 times. From the previous simulations, we took in consideration the amount of time each scenario took and, to gather results for the case in which the defense framework has not been implemented, we ran different simulations in which, the time periods considered, were the duration of time that it took for the network, with defense mechanism implemented, from the beginning of simulation to the "n^{th}" HOPERAA execution.
In each of the following graphs, the blue bars represent the amount of data packets sent throughout that simulation; while the red ones, represent the amount of "contact-initiation packets".

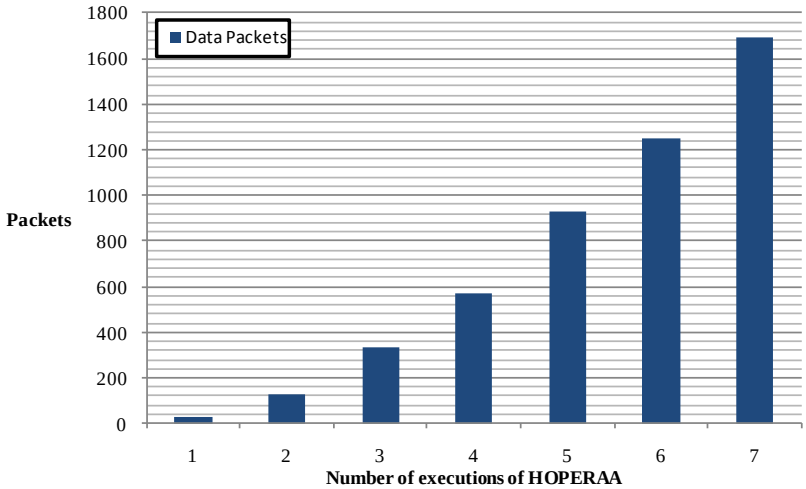
In the network with HOPERAA and BIG WHEEL implemented, we are using a simple scenario in which the Server has 60 ports divided into 3 different intervals "k" with 20 ports each. By looking carefully at the graphs, we can denote the following:

1. In graph 33a we can see that, as the time goes by, the number of data packets being sent increases due to the fact that the HOPERAA Execution Intervals increase after every resynchronization with the Server. Looking at the graph, we can conclude that as the transmission progresses, the overhead derived from the algorithm's implementation is reduced and the amount of data packets being sent overcome the amount of "contact-initialization" packets required; we can also perceive how overhead is reduced after some time so, if the amount of data packets to send is too small then, the cost due to the defense framework, would be relatively larger.

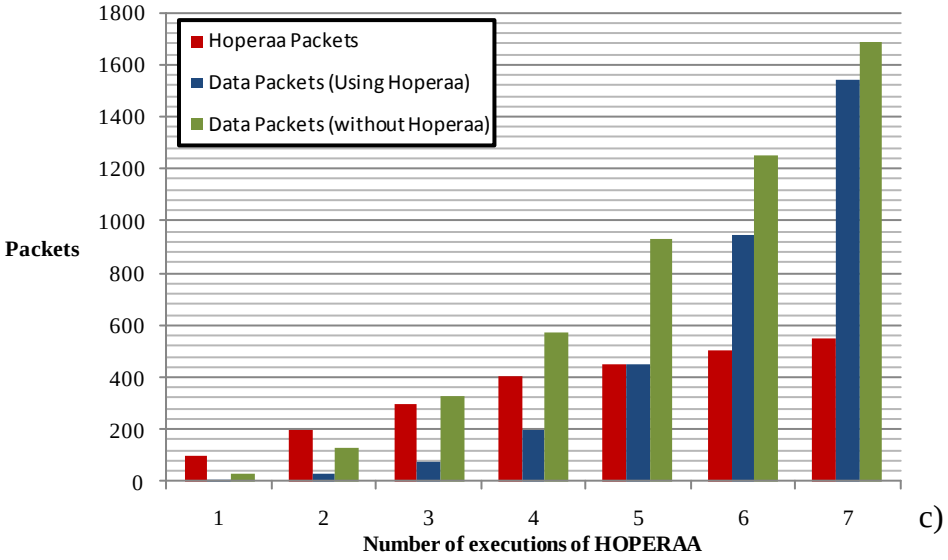
2. Comparing graph 33a with graph 33b, the first evident difference is that the network, where the defense framework was not implemented, is capable of sending more data packets in the same amount of time that it took the other network to execute HOPERAA a specific number of times. This can be seen more clearly in graph 33c where we are using the same *y* axis scale to present the results from graphs 33a and 33b. In graph 33c, the green bars represent the amount of data packets sent when the defense framework was not implemented while; blue bars represent the transmitted data packets when the framework was present, the red bars are related to the "contact-initialization" packets sent throughout each simulation. As mentioned before each of these simulations are related to different number of HOPERAA executions and this number is growing in each simulation from 1 to 7.



a)



b)



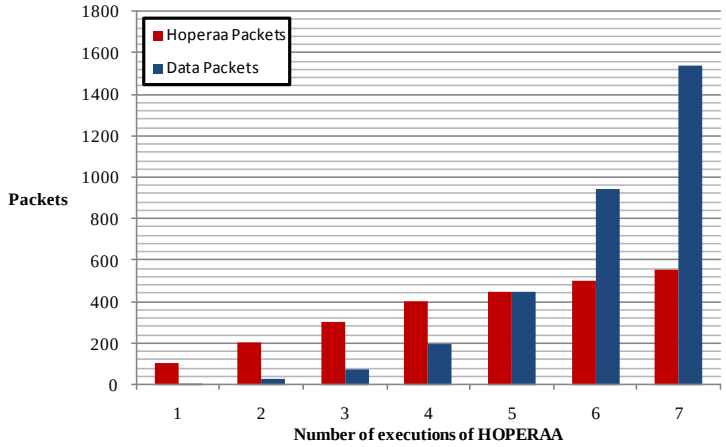
Graph 33: Represents the number of data packets and control packets during n^{th} execution of HOPERAA algorithm.

5.4.5 Results' Analysis: Scenario 5.4.4

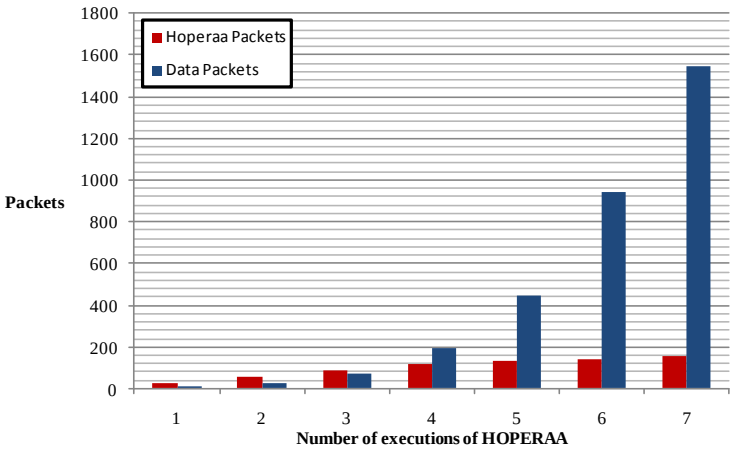
When a client has to resynchronize with the server or, contact it for the first time; it has to execute the "contact initiation phase" which implies sending "contact-initialization" messages to all the ports from a randomly chosen interval and, wait "2μ+L" time units before retrying. Derived from that behavior and comparing graph 34a and 34b, we can observe the overhead induced by the HOPERAA algorithm, as discussed in [5] section 5, can be modeled by the formula:

$$\frac{N}{k} \cdot \frac{1}{1 - \left(\frac{1}{e}\right)^{F}}$$

In our cases we have assumed the same number of ports which means "N" in both cases is 150; however, the value of "k" in 34a and 34b is 3 and 10 respectively. Taking this in consideration, 34a shows the performance of the network in which 150 ports and 3 intervals has been assumed, this means that each interval contains 50 ports; 34b on the other hand, shows the same network but this time with 150 ports and 10 intervals, which means that each interval includes 15 ports. Here, each combination of red and blue bars shows independent simulations with "n^{th}" execution of HOPERAA. So basically each of these combinations shows the whole amount of data packet and contact initiation packets from the beginning of simulation until the "n^{th}" execution of HOPERAA algorithm.



Graph 34a: Represents the amount of data and control packets, during n^{th} execution of HOPERAA, using 150 ports divided into k=3 intervals.



Graph 34b: Represents the amount of data and control packets, during n^{th} execution of HOPERAA, using 150 ports divided into k=10 intervals.

If we look at graph 34a and 34b, we would notice that the amount of contact initialization messages, represented by the red bars, is different for each case; now, if we also take in consideration the formula: $\frac{N}{k} \cdot \frac{1}{1-\left(\frac{1}{e}\right)^F}$

we can deduce that, out of all the variables involved, the overheard is inherently related to the value of "k". By looking closely to the variables considered by this formula, if we can assume that "N" and "$\frac{1}{1-\left(\frac{1}{e}\right)^F}$" are constant in both cases, then we can also say that the overheard depends merely on the value of "k" so, the smaller "k" is, the greater the overhead will be. This can be seen clearly by comparing the height of the red bars from the two graphs previously discussed. Using the results obtained in this section, we have observed that having more intervals will decrease the message overhead but, will also increase the number of guard ports available which, could not always be a good thing since, it means that the number of ports vulnerable for an attack will also increase.

5.5 Study Case 5: Defense Framework and the DDOS problem.

5.5.1 Experiment Specification

In this section, we want to put things in perspective by giving a brief insight and comparison on how the defense framework, suggested in [5] and studied so far, performs compared to a network which is under a DDOS attack.
Let's assume, for the sake of this study case, that we have a Gnutella P2P Network which looks like follows:

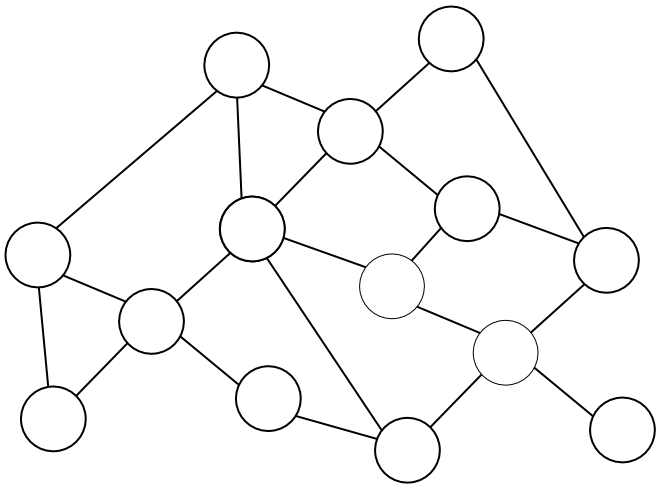


Figure 16: Simple P2P network formed by several nodes of equal roles and capabilities which exchange information and services directly with each other.

From chapter 3, we know that Peer-to-peer (P2P) systems are distributed systems in which nodes of equal roles and capabilities exchange information and services directly with each other and, the network presented above is no different in this sense. Based on what we have studied so far from the Defense Framework, and the results from "Study Case 4", we know that the algorithm's overhead is counterbalanced by the amount of data to be transmitted so; in this case, we will focus our efforts on the data transmission and we will assume that control process, prior to the file transfer, has already been implemented and taken care of.

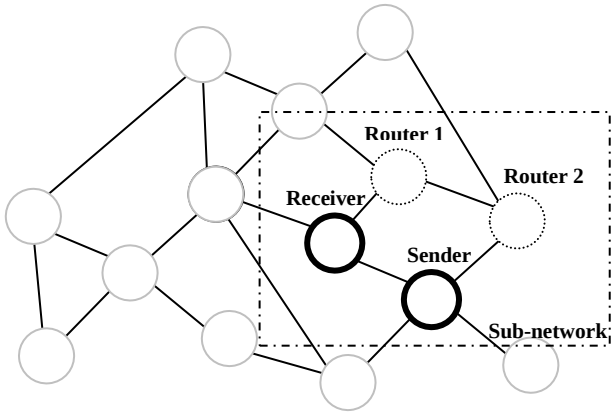


Figure 17: Sub-network derived from the P2P network in figure 16; used to gather results for the scenarios defined in this study case.

In order to gather results for this study case, we will pay particular attention to only one section of the network presented in figure 16 (look at figure 17); in the sub-network that we are going to study, data packets are being transferred from the "sender" to the "receiver" node throughout "router 1" and "router 2" however, depending on the scenario, the condition of the network may change affecting the data transfer in between the nodes. Since for this case, we want to denote how the defense network can be used to mitigate, or reduce, the damage caused by a DDOS attack on a network, the sub-network presented in figure 17 was implemented in ns-2 as follows:

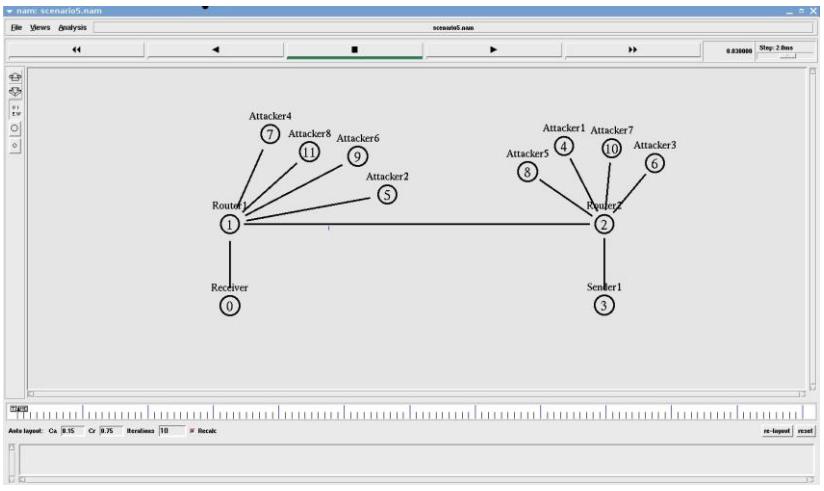


Figure 18: Shows how the Sub-network, presented in figure 17, looks like when simulated in Ns-2

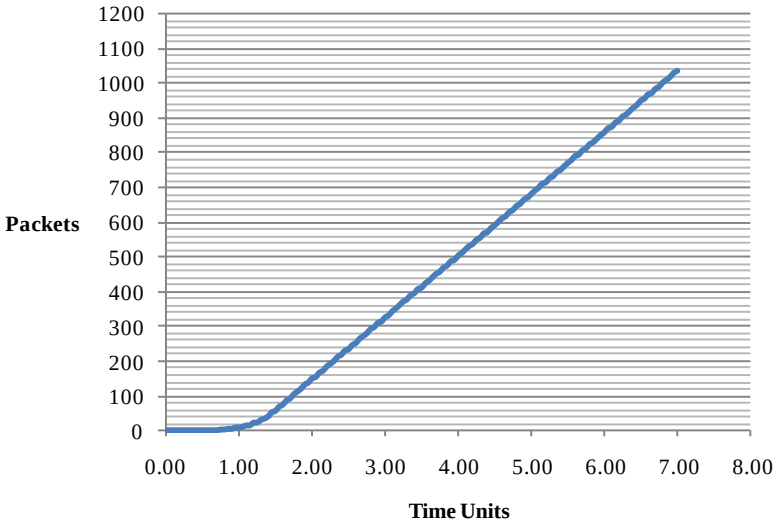
The network above presented, was used to simulate and gather results for the study of the following scenarios:

- Scenario 5.5.1: We will use this case to gather information about how the network behaves under normal circumstances; for this, we will use ns-2 to run a simulation in which the "receiver" and the "sender" interchange data packets with each other for 7 seconds without the intervention from an attacker.
- Scenario 5.5.2: Using the results obtained in "Scenario 5.5.1" as a baseline, we will study how the data transmission (in between "sender" and "receiver") gets affected when an attacker starts sending as many packets as possible to the "receiver". For this, we will consider 2 possible cases:
    - Case (1): The attacker's goal is not to completely disable the "receiver" but rather, diminish the quality of the service by decreasing the amount of legitimate data packets the "receiver" gets.
    - Case (2): The attacker's goal is to completely disable the "receiver" so that the communication with the "sender" gets completely interrupted.
- Scenario 5.5.3: In this last case, we will study the main differences in between the result obtained in the previous 2 scenarios and, the results obtained from a network which implements the defense network suggested in [5] and studied throughout this dissertation.

5.5.2 Results' Analysis: Scenario 5.5.1

For this case, first we will focus our attention on studying how the network behaves under normal circumstances; the results obtained from the latter, will help us to define a common ground so later on, we can compare the network's performance throughout the remaining defined scenarios. With the aim of gathering data results for this section, we used ns-2 in which we defined a network as the one presented in the "Experiment Specification"; we let the simulation run for 7 seconds, with no intervention from an attacker whatsoever, and we centered our attention on the amount of packets being transferred in between the parties.

To achieve the latter, we monitored the simulation and logged, every 0.05 sec, the amount of acknowledgements received at the "sender", from the "receiver" end (Since the receiver send a reply for every packet received, we can assume that every acknowledgment (referred to as "ack" from now on) received means that the packet effectively reached the receiver thus, the packet's transfer was successful).



Graph 35: Shows the amount of acknowledgements received by the "sender", from the "receiver", in a period of 7 seconds and, when the communication is not affected by a third-party (attacker).

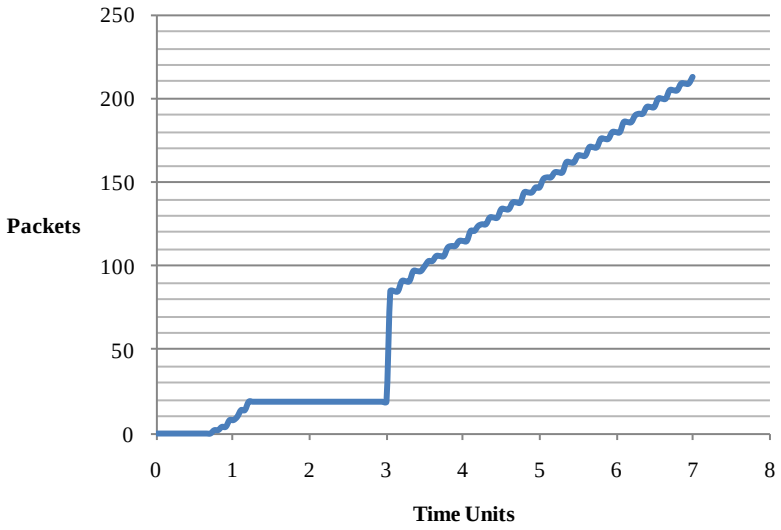
For this study case, the Roundtrip Maximum Delivery Latency ("μ") was defined as "μ = 1.0"; that is the reason why, in graph 35 throughout the first seconds, the line remains on "0" since the "ack" from the packets sent at time 0, and received at 0.5 in the receiver's end, has not reached the "sender" yet. We can see that after the first "ack" is received (around 1.0 in graph 35), the line grows at a constant and steady rate throughout the rest of the simulation; the latter means, that the communication is never interrupted thus the sender is able to successfully transmit (By "transmit", we mean that the Client sent a packet, and received the "ack" from that packet eventually in the time span of the simulation) a total of 1038 packets, at a constant rate.
From the results obtained in this scenario, we can conclude that in 7.0 seconds (simulation time) the Sender was able to successfully transmit 1038 packets uninterruptedly and, at a constant rate.

5.5.3 Results' Analysis: Scenario 5.5.2

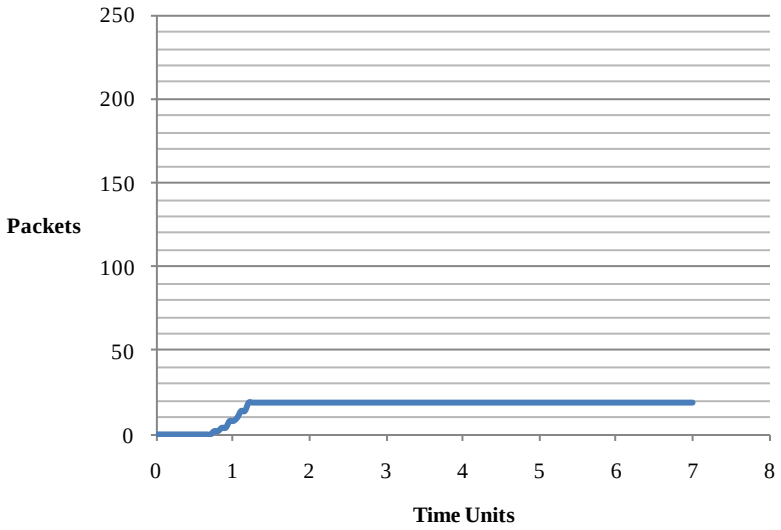
Now that we understand how the network behaves when there is no intervention from a third party, is time to analyze what will happen when an attacker starts sending as many packets as possible to the Receiver, so that the quality of the service, which is providing to the Sender, gets diminished (1) or; the receiver is completely disabled (2).
In order to study the behavior on the network in these two cases (1) and (2), we used a similar approach as the one used in "Scenario 5.5.1" but now, we will take in consideration the following changes:

1. All traffic generated by the Sender is being directed to the default Gnutella port (6346).
2. We included several other nodes (attackers) which create as much traffic as possible and direct it also to the port 6346. The function of the attackers, defined for this and the remaining sections, is only centered on creating as much traffic as possible and, directing it to a specific port.
3. The attack begins at simulation time 1.0.
4. The results gathered for this section represent the ones of a simple network with no defense mechanism implemented whatsoever.

In order to study how the data transmission is affected, we use a similar methodology as the one used before; which means, that once again we used ns-2 to define a network and let the simulation run for 7 seconds, centering our attention on the amount of packets being transferred in between the parties by monitoring the simulation and logging, every 0.05 sec, the amount of acknowledgements received at the "sender's" side. To begin with, we will study how the transmission gets affected just by looking at the graphs obtained from the acknowledgements received at the sender's side; and then, we will compare these graphs with the one obtained in Scenario 5.5.1, when there is no attack present. Graph 36 and 37 present the amount of packet acknowledged when, the attacker diminishes the quality of the communication (1) and, when it completely disables the Receiver (2), respectively.



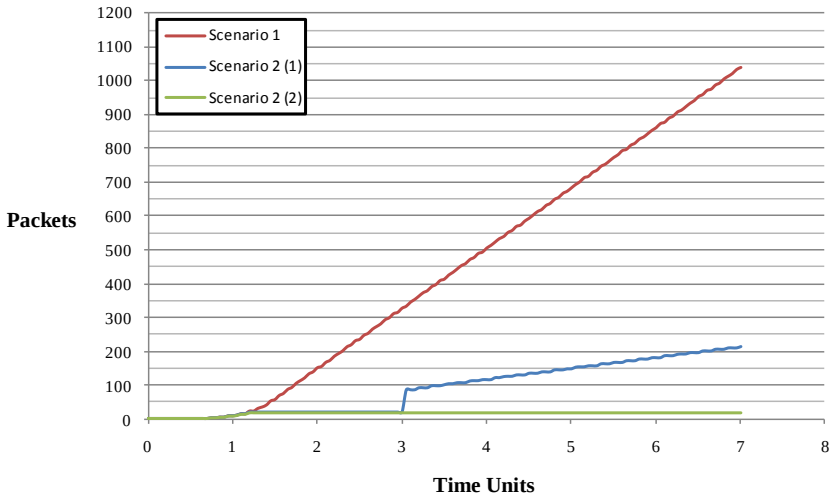
Graph 36: Shows the amount of acknowledgements received by the "sender", from the "receiver", in a period of 7 seconds and, when the communication's quality is diminished by a third-party (attacker).



Graph 37: Shows the amount of acknowledgements received by the "sender", from the "receiver", in a period of 7 seconds and, when the communication is completely disabled by a third-party (attacker).

The first big difference that we can notice is that in "Scenario 5.5.1", the sender was able to successfully transmit 1038 packets uninterruptedly and, at a constant rate. Now, looking at the graph 36 from the first case (1) of this scenario, we can denote that the sender was able to send only 213 packets in the same amount of time; also, after the attack started (at time 1.0) the sender was not able to get an "ack" for about 2 seconds and even after that, the rate at which it was receiving them was not constant.

The latter is obvious when looking at graph 36 ; if we look closely, after time 3.0 we can see that the even though the amount of packets increases, the graph has a “step like” form; the reason for this, is because out of all the packets the sender sent some of them were acknowledged and some other no (due to the fact that the attacker’s packets were consuming the resources at the receiving end therefore, the latter was not able to receive all the packets from the sender) so, the ones that were not, had to be sent again and that is why the graph looks the way it does. Graph 37 represents what happens when the attacker is capable of completely disable the receiver (2), in this case is very obvious that after the attack started the sender was not able to receive any more acknowledgments, the reason for this is because the receiver was too “busy” dealing with the attacker’s packets hence not being able to process any of the packets from the sender; that is why in graph 37, we can see that the sender successfully transmitted 20 packets but, after the attack started, it did not receive any other “ack”; the moment at which the receiver is unable to communicate with the sender, is represented in the graph by the horizontal line starting at 1.2 (X-axis).



Graph 38: Presents a graphical comparison of all the results obtained so far; scenario 5.5.1 when there is no attack; scenario 5.5.2 (1), when the quality of the communication gets weakened; and, scenario 5.5.2 (2) when the communication is completely disabled.

Although we have already discussed the main differences in between the results obtained in this and Scenario 5.5.1; graph 38 presents an accurate, and probably more understandable, comparison on how different these results are from a more graphical point of view. We also want to point out that, as we can see in graph 38, from 0 to 1.0 all three cases behave in the same way and the reason for this is because the attack started only at 1.0 so, it’s from that moment on that the behavior changes depending on whether it is case (1) or case (2).

5.5.4 Results' Analysis: Scenario 5.5.3

In this last case, we will study the main differences in between the result obtained in the previous 2 scenarios and, the results obtained from a network which implements the defense network suggested in [5] and studied throughout this dissertation. In order to study the latter and keep the results as comparable as possible, we used a similar approach as the one used so far but now, due to the definition of the HOPERAA and BIG WHEEL algorithm, we will take in consideration the following aspects for the sake of the results involved in this study:

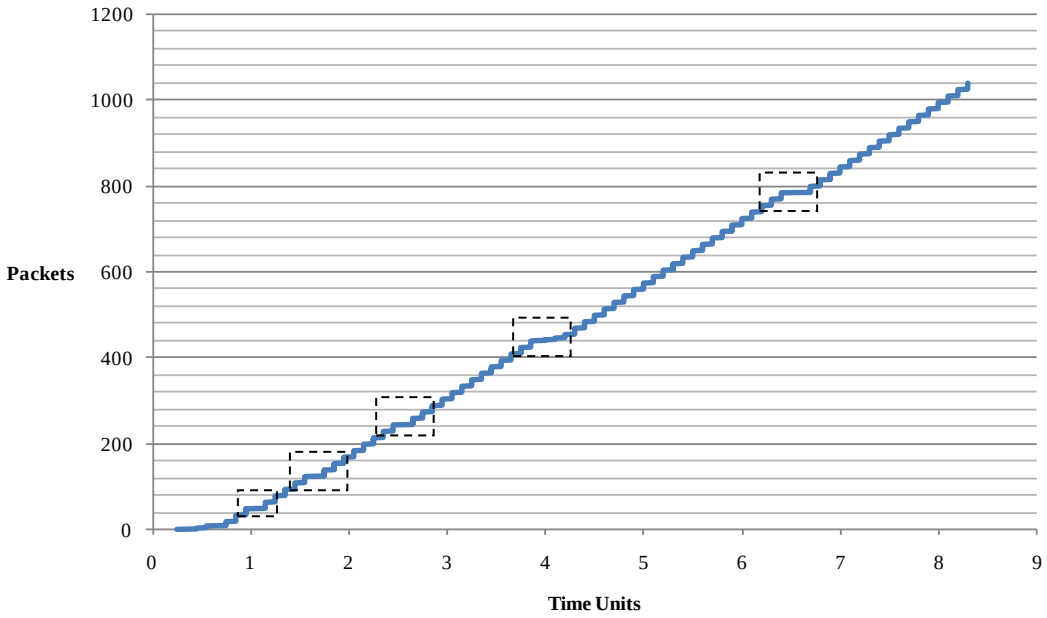
1. The receiver hosts only one sequence with 6000 ports available for data transmission; ports are divided into "k = 6" hence, each interval has 1000 ports evenly.
2. We are considering the "ideal scenario" in which "contact-initialization" packets are never lost and always reach the receiver thus, every time the sender has to resynchronize with the receiver, the latter sends the reply as soon as possible.
3. All traffic generated by the Sender is sent to the ports calculated by the Algorithm thus; they are no longer directed to a single port.
4. As for the network's attributes, we are using:

    - $\mu = 0.1$
    - $L = 0.3$
    - $\tau = 1$
    - $\Delta = 0.1$
    - $\rho = 2$

From "Scenario 5.5.1", we know that the "sender" is able to send a total of 1038 packets in 7 seconds (simulation time) therefore; in this case, we will study how long does it take for this network, using the defense framework, to send the same amount of packets. Graph 39 shows how the network behaves when the defense network is implemented and, as we can see, in this case it took 8.3 seconds to send the 1038 packets; just as we did before, graph 39 represents the amount of acknowledgements received by the "sender", from the "receiver".

By looking at the graph we can notice the following:

1. The squares in the graph represent the moments at which no "ack" was received since the "sender" stop data transmission in order to resynchronize with the "receiver".
2. By looking at the distance from each square in the graph, we can conclude that effectively the "Hoperaa Execution Intervals" are increasing hence; the "sender" is capable of transmitting more and more packets after every time the resynchronization phase is reached.
3. We can see a constant and steady growth in the way packets are being acknowledged by the "receiver".



Graph 39: Shows the moments at which the "sender" is re-synchronizing with the "receiver" and, the amount of acknowledgements received; when considering a network in which the defense framework has been implemented.

Now that we know how long it takes, for the network with the defense framework implemented, to send the same amount of packets as the network from "Scenario 5.5.1", we can affirmatively say that even though the growth presented in this case is still not as good as the one seen in "Scenario 5.5.1" (Look at graph 35), it is definitely better to the ones seen in "Scenario 5.5.2". (Look at graphs 36 and 37)

From the results obtained from "Scenario 5.5.1", "Scenario 5.5.2" and the ones from this, we can conclude:

1. As previously discussed, the network with the defense framework implemented, doesn't perform as well as the one presented in "Scenario 5.5.1" but still performs better than any of the ones presented in "Scenario 5.5.2". The following table presents the results obtained in the previous scenarios compared to the one obtained in this:

	Simulation Time	Packets Transmitted
Scenario 5.5.1	*7.0 sec*	*1038*
Scenario 5.5.2 (1)	*7.0 sec*	*213*
Scenario 5.5.2 (2)	*7.0 sec*	*20*
Scenario 5.5.3	*7.0 sec*	*828*

Table 15: Shows the amount of packets transmitted in 7 seconds (simulation time), by each one of the networks used in "Scenario 5.5.1", "Scenario 5.5.2" and "Scenario 5.5.3".

2. For this case, we considered the "ideal scenario" in which "contact-initialization" packets are never lost and always reach the "receiver" however; when the latter holds no more and according to the results obtained in 5.4, the time it takes to transmit the same amount of packets could increase due to the fact that the "Sender" has to wait until it gets a reply from the "receiver" after every resynchronization. If the latter were to happen, we will notice that the lines marked (by the squares) in graph 39 will increase in longitude hence increasing the overall transmission time.
3. Also based on the results obtained in 5.4, we know that increasing the number of sequences being hosted at the "receiver" can affect the waiting time of the "sender". In this section we used only one sequence but, if we were to increase the number of these, we will notice that the lines marked (by the squares) in graph 39 will decrease in longitude hence reducing the overall transmission time.
4. For this particular case, even if the attacker were to "find out" one of the ports being used during the transmission, the damage caused by the attack would not be so critical, since the overall transmission time will not be severely affected due to the fact that the ports are continuously changing.

6

FUTURE WORK

"A generation which ignores history has no past and no future."

- Robert Heinlein

In this chapter, based on the knowledge and experience gathered throughout the development of this dissertation, we focus on pointing out what we consider as areas of improvement for future study, on the subject discussed in this work.

6.1 Thoroughly investigate algorithm's performance in different network architectures

According to [5], we know that the algorithm was designed with two very important features on mind, the first one is that each Client is able to interact with the Server on an individual basis; the second one is that this is an application-level solution. These two features combined, make the framework highly suitable for implementation on different network architectures, such as P2P Networks. Ever since we started implementing this solution in ns, we had in mind the possibilities of the algorithm so, when designing the new agents, we tried to create them so that theoretically, this extension would be possible by using most of the code developed in this dissertation.

From Chapter 4, we know that in the simulation the "HOPERAA agents" are the ones responsible of transmitting all the information required by Hoperaa, also the ones who calculate the "Hoperaa execution interval" and, to schedule the next time at which the client must re-synchronize with the server.

For the Client-Server model studied in Chapter 4, we know that there is an agent of this kind attached to the Server and, to each Client in the simulation; however, no matter where the "HOPERAA Agents" are located, they all have the same capabilities; the only difference is that the "HOPERAA agent" in the Server is calling different methods to the ones being called by the agent at the Client's side. The previous statement would theoretically make possible for the simulation to be extended, without major changes, into a P2P network in which all nodes have equal roles and capabilities to exchange information and services directly with each other.
Based on this, further analysis of the framework in different architectures is probably an interesting prolongation of the work started in [5] and continued in this dissertation.

6.2 Extend the Framework's Defense Capabilities

As discussed in Chapter 1 and 2, in the past few years, Peer-to-peer networks have become immensely important as one of the most popular Content-Delivery systems [1] but also, unsecure networks had provided fertile ground for attackers to create "zombies" and use them to deploy more powerful DDoS attacks. Taking the previous in consideration, we suggest strengthening the Framework, by grouping several servers as one logical entity. (See figure 19)

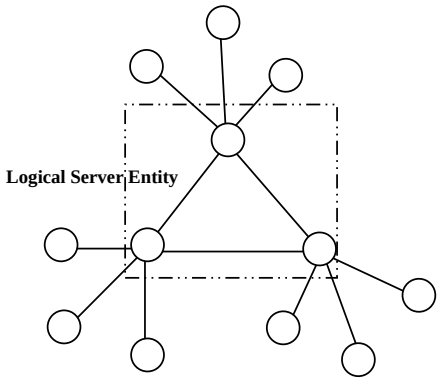


Figure 19: Shows how the Defense Framework can be strengthened by logically grouping several servers as one entity.

In this example the "Server" is a logical unit internally formed by 3 different nodes; unlike the case studied earlier, each one of these nodes will host one sequence of ports. Previously, all the sequence were hosted in the same Server and the Client, depending on the time from its "contact-initiation" message, would hop from one sequence to the other.

In this case, each node will be hosting one sequence and, every time the Client hops from sequence, it will change also the node with who it was communicating. In the original approach, if for any reason the server was disabled, the entire framework would be compromised and none of the Clients would be able to continue with their transmission; this approach has the main advantage that even if one of the internal nodes is disabled, the Clients still have two more sequences (and also the nodes hosting them) available to continue with the transmission.
Nevertheless, this alternative also raises the following problems:

- Depending on the number of Clients involved, the bandwidth required, in the links from internal nodes of the logical unit, will increase thus, creating a greater overhead on the overall framework.
- Previously, the Client only stored information, such as address, number of sequences, ports in each sequence, etc, from a single Server; but now, depending on the methodology followed, the Clients may have to store more information at its end, depending on which node they are communicating with.
- This approach also would increase the number of session being established. Before, only one was necessary since the communication was only with one node; but in this case, because of the sequence being used, the number of sessions established could increase depending on the node hosting such sequence.
- Since Clients will be hopping from node to node depending on the time from their "contact-initiation" messages, it is necessary to design a way for each node (inside the server unit) to be able to access the information from a Client that is contacting it for the first time but, started the communication process with a prior node.
- The most natural solution, for the previous case, would be to have a central repository in which each "server node" would log data such as h1 and t1 (recorded during the first contact with the Client) or the status of the TCP Session; however, this approach would be more disadvantageous than beneficial since, the attacker could target this central repository instead hence, disabling the whole architecture.

Based on the previous observations, and many other that probably we didn't even think of, we believe that the study of this case would be of high importance in determining whether is feasible or not to improve the framework, suggested in [5], by using this approach.

7
CONCLUSIONS

"A conclusion is the place where you get tired of thinking."

- Arthur Bloch

As we discussed throughout this dissertation, Distributed Denial of Service attack is a grave problem which has no easy solution; as defined in the beginning of this work, DDOS attacks are deployed in order to prevent legitimate users of a service, or network resource, from accessing that service or resource. This thesis, among other things, has presented a detailed study and evaluation of HOPERAA and BIG WHEEL, two algorithms derived from the work in [5], used to mitigate this kind of attacks by using a "Port-Hopping" approach while allowing multiparty communication in the presence of clock drifts; from the study of these two algorithms and the results obtained from the study cases considered in this work, we can conclude that:

1. Derived from the analysis of the results obtained in the study cases, we can conclude that the implementation of the Algorithms' in the simulator ns-2, behaves as expected and the results are consistent with the description given in the paper used as reference. Thanks to the prior, this implementation can be used as a reference for further study of the defense framework under circumstances different to the ones considered in this work.
2. The algorithm's definition clearly specifies that it will behave as described whenever the clock drift in between the parties remain constant throughout the whole communication process. From "Study Case 2" we were able to denote that, when "ρ" changes, two of the conditions established by the algorithm's definition are broken, since the values of "rhoUp" and "rhoLow" go from being in between the range of "ρLow ≤ **ρ** ≤ ρUp", as expected; to the range of "**ρ** ≤ ρLow ≤ ρUp".

This behavior is evident in every case, no matter whether "ρ" is increased or decreased from its original value. In spite of the latter, we could observe that the values of "rhoUp" and "rhoLow" were eventually adjusted to their ideal values as the simulation progressed however, according to the framework's description, if we were to continue with the simulation we will also notice that the restriction given by "Δ" will no longer hold thus, making the implementation to behave outside of the acceptance margin.

3. From the results obtained from the "Study Case 3", we were able to conclude that whenever the clock drift in between the parties does not remain constant throughout the whole communication process but, we allow the Server to record new values of "h1" and "t1", the algorithm's behavior will remain inside of the acceptance margin and perform as if it had no record that there were ever a drift change to begin with; the latter, allows the variables "rhoLow", "rhoUp", "Pc" and "Hoperaa Execution Interval" to act as defined in [5] and, their respective values, to be more comparable to the ones obtained from when "ρ" was constant.
4. Based on the analysis of the results obtained in the "Study Case 4", we were able to verify that the overhead created by the defense framework can be affected by several aspects such as:
    a) Number of sequences hosted by the server
    b) Number of intervals in which ports are divided
    c) Number of attempts before the Client gets a reply for the server
5. Finally, from the results gathered in the "Study Case 5" we were able to show that even though a network with the defense framework implemented, doesn't perform as well as one with no defense mechanism whatsoever (in terms of overhead and overall data/time transmission), the algorithm's overhead will be counterbalanced by the amount of data to be transmitted and the reliability of the network whenever the latter is under a DDOS attack.

In this section, we wanted to give the reader a better idea about the analysis and the possible repercussion of the results obtained throughout the develop of this work; however, we strongly recommend the reader to study each subsection included in chapter 5, since in each one of them we give a more detailed explanation and insight about the results and their derived implications as far as the defense framework concerns.

Appendix

A.1 Individual contributions to this work

This paper gives a detailed approach to what Distributed Denial of Service attacks are and, the precarious problem they present for today most common internet-based services and resources. For the work of this thesis, we have included several sections in order to give the reader a complete picture of the problem and, then presented a detailed study, evaluation and explanation of how HOPERAA and BIG WHEEL, two algorithms derived from the work in [5], are suggested in order to mitigate DDOS attacks and also, how they were implemented in the simulator ns-2 and all the necessary changes in order to make such implementation works and behaves as the one presented in the paper "Mitigating Distributed Denial of Service Attacks in Multiparty Applications in the Presence of Clock Drifts" by Zhang Fu, Papatriantafilou Marina and Philippas Tsigas.

Some of the content of this thesis is based on the references specified in the next section; however, most of it was written up based on our understanding about the discussed topic. The following table presents a complete description of all the contributions made by each one of the members, whatever content included in each section (graphs, diagrams, tables, etc) was also structured, studied and created by the person in charge of the overall segment:

Thesis' Report Content	*Responsible (s)*
Abstract	Negin F.
Acknowledgements	Ricardo M. and Negin F.
Introduction	Ricardo M.
1.1 Motivation	Ricardo M.
1.2 Outline	Ricardo M.
Background	Ricardo M.
2.1 P2P Networks	Ricardo M.
2.2 Distributed Denial of Service Attacks (DDOS)	Ricardo M.
2.3 P2P & DDOS attacks	Ricardo M.
A port-hopping approach against DDOS	Ricardo M.
3.1 HOPPING PERIOD, ALIGN AND ADJUST ALGORITHM	Ricardo M.
3.2 BIG WHEEL Algorithm	Ricardo M.

Thesis' Report Content	*Responsible (s)*
Implementation in ns-2	Ricardo M.
4.1 The network Simulator	Ricardo M.
4.2 HOPERAA Implementation in ns-2	Ricardo M.
Analysis and Evaluation	-
5.1 Study Case 1: Single Client/Server Scenario	-
5.2 Study Case 2: Variable Clock Drifts	-
5.3 Study Case 3: Variable Clock Drifts (2)	-
5.4 Study Case 4: Frameworks Overhead	-
5.5 Study Case 5: Defense Framework and the DDOS Problem	-
Future Work	Ricardo M.
6.1 Thoroughly investigate the algorithms' performance in different architectures	Ricardo M. and Negin F.
6.2 Extend the Defense Framework's capabilities	Ricardo M.
Conclusions	Ricardo M.

As for the simulation of the algorithms in ns-2 and the scenarios used to analyze their behavior, Ricardo Moscoso was the one responsible for implementing their respective counterpart in C++ and OTcl; nevertheless, for each one of the study cases there were several actions to be followed and a person responsible for each one of them; reason why, this section was not included in the prior table but, it's included in the following:

Section's Name	*Scenario Design*	*Testing*	*Result's Analysis*
Analysis and Evaluation	Ricardo M.	Negin F.	Ricardo M
1.1 Study Case 1: Single Client/Server Scenario	Ricardo M.	Negin F.	Ricardo M.
1.2 Study Case 2: Variable Clock Drifts	Ricardo M.	Negin F.	Ricardo M.
1.3 Study Case 3: Variable Clock Drifts (2)	Ricardo M.	Negin F.	Ricardo M.
1.4 Study Case 4: Frameworks Overhead	Negin F.	Negin F.	Negin F.
1.5 Study Case 5: Defense Framework and the DDOS Problem	Ricardo M.	Negin F.	Ricardo M.

A.2 hoperaa.h

```
1
2  /*
3   * File: hoperaa.h
4   */
5
6  #ifndef ns_hoperaa_h
7  #define ns_hoperaa_h
8
9  #include "agent.h"
10 #include "tclcl.h"
11 #include "packet.h"
12 #include "address.h"
13 #include "ip.h"
14 #include <stdio.h>
15 #include <iostream>
16 #include <tcl.h>
17 #include <time.h>
18 #include <algorithm>
19 #include <string>
20 #include <fstream>
21
22 struct hdr_hoperaa{
23     char init_reply;     // Control variable (Used to know what kind of packet is being received by the hoperaa agent)
24     double time_stamp;       // This one is used for time record (time - Client / timestamp - Server)
25     int port_seed;       // This one is used for info transmission (p - Client / seed - Server)
26     double t1;               // t1: Time packet was received in the Server side
27     double h1;       // h1: Time from the client's packet
28     int lambda;      // Lambda value for the BIG WHEEL Algorithm
29
30     // Header access methods
31     static int offset_;  // required by PacketHeaderManager
32     inline static int& offset() { return offset_; }
33     inline static hdr_hoperaa* access(const Packet* p) {
34         return (hdr_hoperaa*) p->access(offset_);
35     }
36 };
37
38 class HoperaaAgent : public Agent {
39 public:
40     // Constructor method
41     HoperaaAgent();
42     // Method used to call C++ command procedures for the given shadow class from OTcl
43     virtual int command(int argc, const char*const* argv);
44     // Method called everytime a packet is received by any node
45     virtual void recv(Packet*, Handler*);
46     // Method used to achieve the contact Initialization phase from the algorithm
47     virtual void sendContact();
48     // Method used to perform the "time conversion" from Client to Client (depends on the Clock Drift)
49     virtual double getClock(int client,int option);
50 private:
51     // Variables used to perform the calculations needed by HOPERAA
52     double h1;
53     double t1;
54     double Tc;
55     // Variable used to keep track on the reply packets received from the Server
56     int reply;
57     double Pc;
58     int Pnew;
59     int Pold;
60     double nextHoperaa;
61     double HopExcInt;
62     int seed;
63     int lambda;
64     // Variable "i" from the "Sending Data" phase from the algorithm
65     int iAlg_;
66     // Variable used to count how many times HOPERAA has been executed
67     int hoperaaTimes;
68 };
69
70 #endif // ns_hoperaa_h
71
```

A.3 supportFunctions.h

```
1
2  /*
3   * Description: Include all the support functions used into the Simulation.
4   * Date: 28 July 2009
5   * Version 1.0
6   */
7
8  using namespace std;
9  // Used as an index for the Temporal buffer
10 int iTemporal = 0;
11 // Used as an index for the Permanent buffer
12 int iPermanent = 0;
13 // Used as a control variable (Tcl/C++)
14 int bufferTempFlag = 0;
15
16 /*
17  * This is the port space from the server
18  */
19 int port_space = 60;
20 /*
21  * Specifies the number of intervals in which the port space is divided
22  */
23 int intervals = 3;
24 /*
25  * Specifies the number of elements per interval
26  */
27 int no_elements_interval = port_space/intervals;
28 /*
29  * The intervals are mapped into an array (For easy access and manipulation)
30  */
31 int interval[3][20];
32
33 /*
34  * Structure implemented to simulate the relation in between ports and sequences in BigWheel Algorithm
35  */
36 struct sequence {
37     int lambda;
38     int seed[60];
39     int port[60];
40 };
41 /*
42  * Structure implemented to simulate BUFFER in BigWheel Algorithm
43  */
44 struct buffer {
45     buffer() : clientId(0), clientPort(0), h1(0), t1(0), session(false) {}
46
47     int clientId;
48     int clientPort;
49     double h1;
50     double t1;
51     bool session;
52 };
53 /*
54  * Structure implemented to keep all the relevant information from the Clients
55  */
56 struct clientProfile {
57     int clientId;
58     // 2 -faster ; 1 - Slower
59     int cClockRate;
60     int cDrift;
61 };
62
63 // **********************         BUFFERS USED BY BIGWHEEL          ********************
64 buffer temporal[5];
65 buffer permanent[5];
66 clientProfile clients[4];
67 // ********************************************************************************************
68
69 /*
70  * This procedure 'creates' the actual intervals by initializing the array with elements representing
71  * the Server's ports space ('N')
72  */
73 void createIntervals () {
74    for  (int x = 0; x < intervals; x++){
75       for (int j = 0; j < no_elements_interval; j++) {
76         int temp2 = (port_space/intervals) * (x);
77             interval[x][j] = j + temp2;
78        //printf("\ninterval[%d][%d] = %d",x,j,interval[x][j]);
79       }
80    }
81 }
82
```

```
82
83  /*
84   * Procedure which returns a random integer given a value of seed and lambda; such number
85   * is in between the range of 0 to port_space (total number of ports).
86   */
87  unsigned int randomFunctions (int seed, int lambda) {
88      if (lambda == 0){
89          srand(seed);
90          seed = rand();
91          return (unsigned int)(seed*rand()) % (port_space - 1);
92      }
93      if (lambda == 1){
94          srand(seed);
95          seed = (seed+rand()*lambda)+rand();
96          return (unsigned int)seed % (port_space - 1);
97
98      }
99      if (lambda == 2){
100         srand(seed);
101         seed = (seed+lambda*rand())*rand()+seed;
102         return (unsigned int)seed % (port_space - 1);
103     }
104 }
105 /*
106  * Inserts the Client's Information into the Temporal Buffer
107  */
108 void insertBufferTemporal(int Client,int port,double h1,double t1,bool session) {
109     int flag = 0;
110     for(int j = 0; j <= iTemporal; j++){
111         if (temporal[j].clientId == Client){
112             flag = 1;
113         }
114     }
115     if (flag == 0){
116         temporal[iTemporal].clientId = Client;
117         temporal[iTemporal].clientPort = port;
118         temporal[iTemporal].h1 = h1;
119         temporal[iTemporal].t1 = t1;
120         temporal[iTemporal].session = session;
121         printf("\nclientId(temp): %d\n",temporal[iTemporal].clientId);
122         printf("\nclientPort: %d\n",temporal[iTemporal].clientPort);
123         printf("\nh1: %f\n",temporal[iTemporal].h1);
124         printf("\nt1: %f\n",temporal[iTemporal].t1);
125         iTemporal++;
126         bufferTempFlag = 1;
127         Tcl& tcl = Tcl::instance();         // This is used to execute TCL commands in C++
128         tcl.evalf("set bufferTempFlag 1");
129         tcl.evalf("set sessionStatus_ 1");
130     }
131 }
132 /*
133  * Inserts the Client's Information into the Permanent Buffer
134  */
135 void insertBufferPermanent(int Client,int port,double h1,double t1,bool session) {
136     int flag = 0;
137     for(int j = 0; j <= iPermanent; j++){
138         if (permanent[j].clientId == Client){
139             flag = 1;
140         }
141     }
142     if (flag == 0){
143         permanent[iPermanent].clientId = Client;
144         permanent[iPermanent].clientPort = port;
145         permanent[iPermanent].h1 = h1;
146         permanent[iPermanent].t1 = t1;
147         permanent[iPermanent].session = session;
148         printf("\nclientId(perm): %d\n",permanent[iPermanent].clientId);
149         printf("\nclientPort: %d\n",permanent[iPermanent].clientPort);
150         printf("\nh1: %f\n",permanent[iPermanent].h1);
151         printf("\nt1: %f\n",permanent[iPermanent].t1);
152         iPermanent++;
153     }
154 }
155 /*
156  * Used to get the Information (stored in the permanent buffer) from a Client, based on the Client's Address
157  */
158 buffer getRecord(int Client){
159     for(int i = 0; i <= (sizeof(permanent) / sizeof(buffer)); i++){
160         if (permanent[i].clientId == Client){
161             return permanent[i];
162         }
163     }
164 }
165 /*
166  * Used to get the Information registered for an specific Client, based on the Client's Address
167  * (Information stored by "registerClient" in hoperaa.cc)
168  */
169 clientProfile getProfile(int Client){
170     for(int i = 0; i <= (sizeof(clients) / sizeof(clientProfile)); i++){
171         if (clients[i].clientId == Client){
172             return clients[i];
173         }
174     }
175 }
176 /*
177  * Used to erase the information stored in the temporal buffer (Called after the Server has replied to all
178  * Clients in the buffer)
179  */
180 void resetBufferTemporal(){
181     memset( &temporal, 0, sizeof(temporal) );
182     iTemporal = 0;
183     printf("\033[22;31m\n   ------------------------------------------------------ \n");
184     printf("      Cleaning the Temporal Buffer (After sending Reply)\n");
185     printf("   ------------------------------------------------------\033[22;30m");
186 }
187
```

A.4 hoperaa.cc

```
1
2  /*
3   * File: hoperaa.cc
4   */
5
6  #include "hoperaa.h"
7  #include "supportFunctions.h"
8
9  /*
10  * NULL is being used to represent 'UNDEF' from the algorithm
11  */
12
13 double periodL_;                        // Period ('L') for the sumulated scenario
14 double maxDeliveryLatency_;             // Maximun Delivery Latency ('Miu')
15 int noSequences_;                       // Total number of sequences in the Server (Mainly used for Big WHeel Alg)
16 double deltaTime_;                      // Delta time (Title: Adaptive Hopping Algorithm)
17 int totalPorts_;                       // Total number of ports in the Server
18 int noIntervals_;                      // Used to know in how many intervals the server's ports are divided
19 int indexProfileClient = 1;            // Used to keep track of how many Clients are involved in the Simulation
20 int seedServer = 2;                    // The seed value used by the Server to calculate the ports
21 /*
22  * Variable used in Case Study 3 to allow the Server to record new values when the Clock Drift is variable.
23  * int flag = 0;
24  */
25 Tcl& tcl = Tcl::instance();             // Instance used to execute tcl commands from C++
26
27 int hdr_hoperaa::offset_;
28 static class HoperaaHeaderClass : public PacketHeaderClass {
29 public:
30         HoperaaHeaderClass() : PacketHeaderClass("PacketHeader/hoperaa",
31                                                  sizeof(hdr_hoperaa)) {
32                 bind_offset(&hdr_hoperaa::offset_);
33         }
34 } class_hoperaahdr;
35
36
37 static class HoperaaClass : public TclClass {
38 public:
39         HoperaaClass() : TclClass("Agent/hoperaa") {}
40         TclObject* create(int, const char*const*) {
41                 return (new HoperaaAgent());
42         }
43 } class_hoperaa;
44
45
46
47 /* These are binded variables which link the C++ enviroment with TCL*/
48 HoperaaAgent::HoperaaAgent() : Agent(PT_HOPERAA)
49 {
50         bind("periodL_", &periodL_);
51         bind("maxDeliveryLatency_", &maxDeliveryLatency_);
52         bind("deltaTime_",&deltaTime_);
53         bind("totalPorts_",&totalPorts_);
54         bind("noSequences_",&noSequences_);
55         bind("packetSize_", &size_);
56         bind("noIntervals_", &noIntervals_);
57 }
58
59 /*
60  * This function is used to determine the Application's Clock from the Client
61  */
62 double HoperaaAgent::getClock(int client ,int option)
63 {
64         double t = 0;
65         if (option == 0) {
66                 if(clients[client].cClockRate == 2){
67                         t = (Scheduler::instance().clock() * clients[client].cDrift);
68                         t = ((t + ((2 * maxDeliveryLatency_) + periodL_))/clients[client].cDrift);
69                 }
70                 if(clients[client].cClockRate == 1){
71                         t = Scheduler::instance().clock();
72                         t = ((t + (((2 * maxDeliveryLatency_) + periodL_) * clients[client].cDrift)));
73                 }
74         }
75         if (option == 1) {
76                 if(clients[client].cClockRate == 2){
77                         t = Scheduler::instance().clock() * clients[client].cDrift;
78                 }
79                 if(clients[client].cClockRate == 1){
80                         t = Scheduler::instance().clock() / clients[client].cDrift;
81                 }
82         }
```

```
        /*
         * This is used to compensate for the value of miu
         */
        if (option == 2) {
                if (clients[client].cClockRate == 2){
                        t = ((Scheduler::instance().clock() - (maxDeliveryLatency_/2)) * clients[client].cDrift) + (maxDeliveryLatency_/2);
                }
                if (clients[client].cClockRate == 1) {
                        t = ((Scheduler::instance().clock() - (maxDeliveryLatency_/2)) / clients[client].cDrift) + (maxDeliveryLatency_/2);
                }
        }
        return t;
}

/*
 * This function sends Contact Initialization messages to all ports from a randomly-chosen interval
 */
void HoperaaAgent::sendContact()
{
/*
 * This was used in Study Case 3 to tell the Client when to change the "Clock's Drift"
 *              if (hoperaaTimes == 8){
 *                      tcl.evalf("$hoperaaC1 changeDrift 3");
 *                      flag = 1;
 *              }
 *              if (hoperaaTimes == 16){
 *                      tcl.evalf("$hoperaaC1 changeDrift 4");
 *                      flag = 1;
 *              }
 */
        // Creating the intervals to be used by the Client
        createIntervals();
        // Sets a Random Seed
        srand(time(NULL));
        // Gives a random number from 0 to noIntervals_
        int randomInterval = (unsigned int)(rand() % (noIntervals_ + 1));
        printf("\033[22;31m\n   ------------------------------------------------------------");
        printf("\n      TIME: %f - Client %d sends C.I. PKTS to interval %d   ",getClock(addr(),1),addr(),randomInterval);
        printf("\n   ------------------------------------------------------------\033[22;30m");
        // Start sending contac-initialization packets to all the ports from the randomly-chosen interval
        for  (int i = 0; i < no_elements_interval; i++){
                // temporal variable used for the port specification
                int temp = interval[randomInterval][i];
                // Create a new packet
                Packet* pkt = allocpkt();
                // Access the Ping header for the new packet:
                hdr_hoperaa* hdr = hdr_hoperaa::access(pkt);
                // Access the IP header for the new packet:
                hdr_ip* hdrip = hdr_ip::access(pkt);
                // Control variable (0 - Init / 1 - Reply)
                hdr->init_reply = 9;
                // Specifying the port range based on the algorithm
                hdr->port_seed = temp;
                // Sets the packet's source address
                hdrip->saddr() = addr();
                // Sets the packet's source port
                hdrip->sport() = port();
                // Especifies the destination address (the id_node of the receiver)
                hdrip->daddr() = daddr();
                // Specifies the destination port.
                hdrip->dport() = temp;
                // Timestamps the packet with the current Client's time (Depends on the Clock Drift)
                hdr->time_stamp = getClock(addr(),1);
                send(pkt, 0);
        }
        double t = getClock(addr(),0);
        tcl.evalf("$ns at %f { if {$replyTCL == 0} {$hoperaaC%d sendContact } }",t,addr());
}

int HoperaaAgent::command(int argc, const char*const* argv)
{
        if(argc == 5) {
                /*
                 * Used to register a Client
                 */
                if (strcmp(argv[1], "registerClient") == 0) {
                        clients[indexProfileClient].clientId = atoi(argv[2]);
                        clients[indexProfileClient].cClockRate = atoi(argv[3]);
                        clients[indexProfileClient].cDrift = atoi(argv[4]);
                        indexProfileClient++;
                        return (TCL_OK);
                }
        }
        if(argc == 4) {
                /*
                * Used by the Server to send a REPLY after being contacted by the Client
                */
                if (strcmp(argv[1], "sendReply") == 0) {
                        for(int j = 0; j < iTemporal; j++){
                                Packet* pktret = allocpkt();
                                // Access the header for the new packet:
                                hdr_hoperaa* hdrret = hdr_hoperaa::access(pktret);
                                // Access the IP header for the new packet:
                                hdr_ip* hdrip = hdr_ip::access(pktret);
                                // Sets the packet's source address
                                hdrip->saddr() = addr();
```

```
                        hdrip->daddr() = temporal[j].clientId;
                        // Sets the packet's source port
                        hdrip->sport() = port();
                        // Control variable (0 - Init / 1 - Reply)
                        hdrret->init_reply = 8;
                        // Sending the seed to the client
                        hdrret->port_seed = atoi(argv[2]);
                        // When the client sent the packet
                        hdrret->h1 = h1;
                        // Lambda value for the BIG WHEEL Alg
                        hdrret->lambda = atoi(argv[3]);
                        // When the server received the packet
                        hdrret->t1 = t1;
                        hdrip->dport() = temporal[j].clientPort;
                        // Timestamp from when the reply is sent back to the client
                        hdrret->time_stamp = Scheduler::instance().clock();
                        // Send the packet
                        send(pktret, 0);
                        printf("\033[22;31m\n   ------------------------------------------------------------ \n");
                        printf("      TIME: %f - Sending Reply to Client %d (seed %d)\n",Scheduler::instance().clock(),hdrip->daddr(),atoi(argv[2]));
                        printf("   ------------------------------------------------------------\033[22;30m");
                        tcl.evalf("$tcpC%d set ack_",hdrip->daddr());
                        const char* ni = tcl.result();
                        double next = Scheduler::instance().clock();
                        tcl.evalf("$ns at %f {if {[$tcpC%d set ack_] == %d } { $hoperaaCD closeClientSession } }",next + periodL_,hdrip->daddr(),atoi(ni));
                }
                // Cleans the Temporal Buffer when using BIG WHEEL
                resetBufferTemporal();
                return (TCL_OK);
        }
}

if(argc == 2) {
        /*
         * Prints the details about the simulation
         */
        if (strcmp(argv[1], "printDetails") == 0) {
                for (int i = 1; i <= indexProfileClient; i++){
                        if(clients[i].cClockRate == 2){
                                printf("\033[22;39m\n   ------------------------------------------------------------------------------------------ \n");
                                printf("      CLIENT(%d) %dX faster than Server (miu = %f, delta = %f, L = %f )\n",
                                clients[i].clientId,clients[i].cDrift,maxDeliveryLatency_,deltaTime_,periodL_);
                                printf("   ------------------------------------------------------------------------------------------\n\033[22;30m");
                        }
                        if(clients[i].cClockRate == 1){
                                printf("\033[22;39m\n   ------------------------------------------------------------------------------------------ \n");
                                printf("      CLIENT(%d) %dX slower than the server (miu = %f, delta = %f, L = %f )\n",
                                clients[i].clientId,clients[i].cDrift,maxDeliveryLatency_,deltaTime_,periodL_);
                                printf("   ------------------------------------------------------------------------------------------\n\033[22;30m");
                        }
                }
                return (TCL_OK);
        }
        /*
        * Closes a previously started session, when neccesary
        */
        if (strcmp(argv[1], "closeClientSession") == 0) {
                tcl.evalf("closeSession");
                h1 = 0;
                t1 = 0;
                Tc = NULL;
                reply = 0;
                Pc = periodL_;
                return (TCL_OK);
        }
        /*
         * This is used to Update the values of Pold and Pnew according to the Algorithm
         */
        if (strcmp(argv[1], "updatePorts") == 0) {
                // Used to stop the Client from sending any more packets to Pold and Pnew simultaneously
                tcl.evalf(" $tcpC%d set replicate_ -1 ",addr());
                Pold = Pnew;
                Pnew = randomFunctions ((seed + iAlg_ + 1), lambda);
                printf("\033[22;31m\n   ------------------------------------------------------------ \n");
                printf("      TIME : %f - Client %d Updating Pold: %d and Pnew: %d \n",getClock(addr(),1),addr(),Pold,Pnew);
                printf("   ------------------------------------------------------------\n\033[22;30m");
                iAlg_ ++;
                // recursive call to sendData
                tcl.evalf("$hoperaaC%d sendData",addr());
                return (TCL_OK);
        }
        /*
         * Used for the Client to Send Data to the Server
         */
        if (strcmp(argv[1], "sendData") == 0) {
                printf("\033[22;31m\n   ------------------------------------------------------------ \n");
                printf("      TIME: %f - Client %d Sending DATA to Pold: %d\n",getClock(addr(),2),addr(),Pold);
                printf("   ------------------------------------------------------------\n\033[22;30m");
                tcl.evalf("$tcpC%d set window_ 15 ; $tcpC%d set dst_port_ %d ; $ftpC%d start",addr(),addr(),Pold,addr());
                double lowRange = 0;
                double highRange = 0;
                // This is used for when the Client is faster than the Server
                if(clients[addr()].cClockRate == 2){
                        if (Tc == 0) {
                                Tc = getClock(addr(),2);
                        }
                        lowRange = (Tc + ((iAlg_ * Pc) - maxDeliveryLatency_))/clients[addr()].cDrift;
                        highRange = (Tc + ((iAlg_ * Pc)))/clients[addr()].cDrift;
                }
                // This is used for when the Client is Slower than the Server
                if (clients[addr()].cClockRate == 1){
                        if (Tc == 0) {
                                Tc = getClock(addr(),2);
                        }
                        lowRange = (Tc + ((iAlg_ * Pc) - maxDeliveryLatency_))*clients[addr()].cDrift;
                        highRange = (Tc + (iAlg_ * Pc))*clients[addr()].cDrift;
                }
                // Schedules the time at which the Client must send DATA to Pold and Pnew alike
                if ( lowRange <= nextHoperaa) {
                        tcl.evalf("$ns at %f {puts \" \n[$ns now]: Sending data also to Pnew \" ; $tcpC%d set replicate_ %d}",lowRange,addr(),Pnew);
                }
                // Schedules the time at which the Client must calculate new values of Pold and Pnew
                if (highRange <= nextHoperaa) {
                        tcl.evalf("$ns at %f { $hoperaaC%d updatePorts }",highRange,addr());
                }
```

```
294                     /*
295                      * This Section is used to print, in the command window, information related to the algorithm
296                      * It has no effect or use in the actual calculations from HOPERAA
297                      */
298                     double temp1 = 0;
299                     double temp2 = 0;
300                     double temp3 = 0;
301                     if (clients[addr()].cClockRate == 2){
302                             temp1 = nextHoperaa*clients[addr()].cDrift;
303                             temp2 = lowRange*clients[addr()].cDrift;
304                             temp3 = highRange*clients[addr()].cDrift;
305
306                     }
307                     if (clients[addr()].cClockRate == 1) {
308                             temp1 = nextHoperaa/clients[addr()].cDrift;
309                             temp2 = lowRange/clients[addr()].cDrift;
310                             temp3 = highRange/clients[addr()].cDrift;
311                     }
312                     printf("\033[22;34m\n   **********************  Client %d  *************************", addr());
313                     printf("\n      * Tc = %f",Tc);
314                     printf("\n      * nextHoperaa = %f (Clients:%f)",nextHoperaa,temp1);
315                     printf("\n      * (Tc + ((iAlg_ * Pc) - maxDeliveryLatency_)) = %f (Client's:%f)",lowRange,temp2);
316                     printf("\n      * (Tc + (iAlg_ * Pc)) = %f (Client's:%f)\n",highRange,temp3);
317                     printf("   ************************************************************\033[22;30m");
318                     /*
319                      ***************************************************************************************************
320                      */
321                     return (TCL_OK);
322             }
323             /*
324              * Called everytime the Client has to resynchronize with the Server
325              */
326             if (strcmp(argv[1], "startHoperaa") == 0) {
327                     printf("\033[22;31m\n   ----------------------------------------------");
328                     printf("\n      TIME: %f - Client %d Executing HOPERAA  ",getClock(addr(),1)),addr();
329                     printf("\n   ----------------------------------------------");
330                     printf("\033[22;30m");
331                     tcl.evalf("set replyTCL 0");
332                     reply = 0;
333                     sendContact();
334                     double t = getClock(addr(),0);
335                     tcl.evalf("$ns at %f { if {$replyTCL == 0 && $counter == 0} {$hoperaaC%d startHoperaa ; set counter 1 } }",t,addr());
336                     return (TCL_OK);
337             }
338             /*
339              * Used by the Server Node to Create the Port Sequences and Schedule the ports' open and close time
340              */
341             if (strcmp(argv[1], "startSequence") == 0) {
342                     double next = Scheduler::instance().clock();
343                     sequence seq[noSequences_];
344                     int counter = 0;
345                     for  (int j = 0; j < noSequences_; j++){
346                             seq[j].lambda = j;
347                             //seedServer = 0; if you wanna have the same ssed for each
348                             for  (int x = 0; x < totalPorts_; x++) {
349                                     seq[j].seed[x] = seedServer;
350                                     seq[j].port[x] = randomFunctions (seedServer,j);
351                                     seedServer = seedServer + 1;
352                             }
353                     }
354                     for  (int j = 0; j < noSequences_; j++){
355                     printf(" ------------ sequence %d ------------\n",seq[j].lambda);
356                             for  (int x = 0; x < totalPorts_; x++) {
357                                     printf("        port: %d (seed: %d) \n",seq[j].port[x],seq[j].seed[x]);
358                             }
359                             printf("---------------------------------------- \n");
360                     }
361                     while(counter < totalPorts_){
362                     int i = 0;
363                             while (i < noSequences_){
364                                     tcl.evalf("$ns at %f {if {$sessionStatus_ == 1} { openWorkerPort %d ; if { $bufferTempFlag == 1} { $hoperaaCD sendReply %d %d
365                                     i++;
366                             }
367                             next = next + periodL_;
368                             counter ++;
369                     }
370                     // return TCL_OK, so the calling function knows that the command has been processed
371                     return (TCL_OK);
372             }
373             /*
374              * Used by the Client Node in the "Contact-initiation" phase
375              */
376             if (strcmp(argv[1], "sendContact") == 0) {
377                     Tc = NULL;
378                     reply = 0;
379                     tcl.evalf("setReply 0");
380                     // Set the value of Pc = L (HOPERAA Algorithm's definition)
381                     Pc = periodL_;
382                     sendContact();
383                     // return TCL_OK, so the calling function knows that the command has been processed
384                     return (TCL_OK);
385             }
386         }
387
388   return (Agent::command(argc, argv));
389 }
390
391 void HoperaaAgent::recv(Packet* pkt, Handler*)
392 {
393   // Access the IP header for the received packet:
394   hdr_ip* hdrip = hdr_ip::access(pkt);
395
396   // Access the Ping header for the received packet:
397   hdr_hoperaa* hdr = hdr_hoperaa::access(pkt);
398
399   /*
400    *    Defines the Actions to be taken when the Client Node receives a reply from the Server
401    */
402   if (hdr->init_reply == 8) {
403
404         printf("\033[22;31m\n   ------------------------------------------------------------ \n");
405         printf("      TIME: %f - Client %d RECEIVE (reply,seed,time,h1,t1)\n",getClock(addr(),2),addr());
406         printf("   ------------------------------------------------------------\n\033[22;30m");
```

```
        if (reply == 0){
                reply = 1;
                // As defined by the algorithm Tc = 0
                Tc = 0;
                double hcT4 = 0;
                tcl.evalf("set replyTCL 1");
                hcT4 = getClock(addr(),2);
                // As defined by the algorithm i = 1
                iAlg_ = 1;
                // Obtaining the seed and lambda from the received packet
                seed = hdr->port_seed;
                lambda = hdr->lambda;
                /*****************************Execute HOPERAA for the first time**********************************/
                printf("\033[22;34m   ****************   [HOPERAA EXECUTION #%d]   ****************",hoperaaTimes);
                printf("\n      * SETTING ALL INVOLVED VARIABLES (Hc(t4), Hc(t1), t3,t2) \n");
                // hcT4 is updated at the beginning when PKT is received
                // deltaTime_ is set at the beginning of the simulation
                // Obtaining these values from the received packet
                double hcT1 = hdr->h1;
                double t3 = hdr->time_stamp;
                double t2 = hdr->t1;
                printf("         · t2 (t1) = %f \n         · t3 (ts) = %f \n         · hc(t1) (h1) = %f \n         · Hc(t4) (now) = %f",t2,t3,hcT1,hcT4);
                printf("\n      * CALCULATING VALUES OF rhoUp AND rhoLow \n");
                double rhoLow = ((hcT4 - hcT1)/(t3-t2+(2 * maxDeliveryLatency_)));
                double rhoUp = ((hcT4 - hcT1)/(t3-t2));
                printf("         · rhoUp = %f \n         · rhoLow = %f \n",rhoUp,rhoLow);
                printf("      * CALCULATING VALUE OF THE HOPERAA EXECUTION INTERVAL **\n");
                if((rhoUp < 1) && (rhoLow < 1)){
                        HopExcInt = ((rhoLow * deltaTime_)/(1 - rhoLow)) ;
                        printf("           (rhoUp and rhoLow smaller than 1)");
                        } else if((rhoUp > 1) && (rhoLow > 1)){
                                HopExcInt = ((rhoUp * deltaTime_)/(rhoUp - 1));
                                printf("           (rhoUp and rhoLow larger than 1)");
                        } else if((rhoUp > 1) && (rhoLow < 1)){
                                printf("           (rhoUp larger than 1 and rhoLow lower than 1)");
                                double temp1 = ((rhoLow * deltaTime_)/(1 - rhoLow));
                                double temp2 = ((rhoUp * deltaTime_)/(rhoUp - 1));
                                HopExcInt = min(temp1,temp2);
                        }
                printf("\n         · hopExeInt = %f\n",HopExcInt);
                printf("      * Pc ADJUSTMENT AND HOPERAA EXECUTION INTERVAL UPDATE \n");
                if((1 <= rhoLow) && (rhoLow <= rhoUp)){
                        Pc = periodL_ * rhoLow;
                        printf("           (Changing the value of Pc because 1 < rhoLow < rhoUp)");
                        HopExcInt = (((rhoUp * rhoLow)* deltaTime_)/(rhoUp - rhoLow));
                }else if((rhoLow <= rhoUp) && (rhoUp <= 1)){
                        Pc = periodL_ * rhoUp;
                        printf("           (Changing the value of Pc because rhoLow < rhoUp < 1)");
                        HopExcInt = (((rhoUp * rhoLow)* deltaTime_)/(rhoUp - rhoLow));
                } else {
                        printf(" (Do nothing)");
                }
                printf("\n         · hopExeInt = %f",HopExcInt);
                printf("\n         · Pc = %f\n",Pc);
                // Setting the value of Pold (First port to be used after receiving the REPLY) and Pnew (next Port)
                Pold = randomFunctions (seed, lambda);
                Pnew = randomFunctions ((seed + 1), lambda);
                printf("      * CALCULATING VALUE OF THE PORTS ");
                printf("\n         · seed: %d\n         · lambda: %d\n         · Pold: %d\n         · Pnew: %d\n",seed,lambda,Pold,Pnew);
                printf("   ************************************************************\033[22;30m");
                // Calculating when the next resynchronization will be executed
                Tc = getClock(addr(),2);
                if (clients[addr()].cClockRate == 2){
                        nextHoperaa = (Tc + HopExcInt)/clients[addr()].cDrift;
                }
                if (clients[addr()].cClockRate == 1) {
                        nextHoperaa = (Tc + HopExcInt)*clients[addr()].cDrift;
                }
                // After Executing Hoperaa Pnew and Pold are calculated and the next packet is sent to the Server right away
                tcl.evalf("$hoperaaC%d sendData",addr());
                // Scheduling the next resynchronization and Stopping the Traffic generator
                tcl.evalf("$ns at %f { $ftpC%d stop ; $tcpC%d set window_ 0 ; $hoperaaC%d startHoperaa }",nextHoperaa,addr(),addr(),addr());
                // The following is only used to print on a file the values derived from calculating HOPERAA
                if (addr() == 1) {
                        ofstream myfile ("hoperaaVariablesClient1",ios::app);
                        if (myfile.is_open())
                        {
                                myfile << "Hoperaa Execution Time:" << hoperaaTimes << "    ";
                                myfile << "t2:" << t2 << "    ";
                                myfile << "t3:" << t3 << "    ";
                                myfile << "Hc(t1):" << hcT1 << "    ";
                                myfile << "Hc(t4):" << hcT4 << "    ";
                                myfile << "Pc:" << Pc << "    ";
                                myfile << "HopExcInt:" << HopExcInt << "    ";
                                myfile << "rhoLow:" << rhoLow << "    ";
                                myfile << "rhoUp:" << rhoUp << "    \n";
                                myfile.close();
                                hoperaaTimes++;
                        }
                        else cout << "Unable to open file";
                }
                if (addr() == 2) {
                        ofstream myfile ("hoperaaVariablesClient2",ios::app);
                        if (myfile.is_open())
                        {
                                myfile << "Hoperaa Execution Time:" << hoperaaTimes << "    ";
                                myfile << "t2:" << t2 << "    ";
                                myfile << "t3:" << t3 << "    ";
                                myfile << "Hc(t1):" << hcT1 << "    ";
                                myfile << "Hc(t4):" << hcT4 << "    ";
                                myfile << "Pc:" << Pc << "    ";
                                myfile << "HopExcInt:" << HopExcInt << "    ";
                                myfile << "rhoLow:" << rhoLow << "    ";
                                myfile << "rhoUp:" << rhoUp << "    \n";
                                myfile.close();
                                hoperaaTimes++;
                        }
                        else cout << "Unable to open file";
                }
```

```
516                 if (addr() == 3) {
517                         ofstream myfile ("hoperaaVariablesClient3",ios::app);
518                         if (myfile.is_open())
519                         {
520                                 myfile << "Hoperaa Execution Time:" << hoperaaTimes << "    ";
521                                 myfile << "t2:" << t2 << "    ";
522                                 myfile << "t3:" << t3 << "    ";
523                                 myfile << "Hc(t1):" << hcT1 << "    ";
524                                 myfile << "Hc(t4):" << hcT4 << "    ";
525                                 myfile << "Pc:" << Pc << "    ";
526                                 myfile << "HopExcInt:" << HopExcInt << "    ";
527                                 myfile << "rhoLow:" << rhoLow << "    ";
528                                 myfile << "rhoUp:" << rhoUp << "    \n";
529                                 myfile.close();
530                                 hoperaaTimes++;
531                         }
532                         else cout << "Unable to open file";
533                 }
534         }
535   }
536
537   /*
538    *  Defines the actions taken when the Server gets a Contact-initiation message
539    */
540   if (hdr->init_reply == 9) {
541         int sClient = hdrip->saddr();
542         int sPort = hdrip->sport();
543         tcl.evalf("set counter 0");
544         // Stores when the packet was received at the server's side
545         t1 = Scheduler::instance().clock();
546         // Storing the time from the client's packet
547         h1 = hdr->time_stamp;
548         printf("\033[22;31m\n   ---------------------------------------------------------------- \n");
549         printf("      TIME: %f - RECEIVE (init,time,p) port: %d (Client %d) \n",Scheduler::instance().clock(),hdrip->dport(),sClient);
550         printf("   ----------------------------------------------------------------\033[22;30m");
551         // Inserting the contact information into the temporal Buffer (Defined by the BIG WHEEL ALGORITHM)
552         insertBufferTemporal(sClient,sPort,h1,t1,true);
553 /*
554  *      Used in Case Study 3 to overwrite the Permanent record and store new values of h1 and t1 when the Clock's Frift has changed
555  *      if (isClientinPermanent(sClient) == false ){
556  *      if (flag == 1){
557  *              permanent[iPermanent].clientId = sClient;
558  *              permanent[iPermanent].clientPort = sPort;
559  *              permanent[iPermanent].h1 = h1;
560  *              permanent[iPermanent].t1 = t1;
561  *              permanent[iPermanent].session = true;
562  *              flag = 0;
563  *      }
564  */
565         // Inserting the contact information into the permanent Buffer (Defined by the BIG WHEEL ALGORITHM)
566         if(getRecord(sClient).session  == false){
567                 getRecord(sClient).session  = true;
568                 insertBufferPermanent(sClient,sPort,h1,t1,true);
569         } else {
570         // If the Client has contacted the Server before, h1 and t1 will be obtained from the permanent buffer
571                 t1 = getRecord(sClient).t1;
572                 h1 = getRecord(sClient).h1;
573         }
574         // Discarding the received packet
575         Packet::free(pkt);
576   }
577 }
578
579
580
```

A.5 closedPort.cc

```
1
2  /*
3   * File: closedPort.cc
4   */
5
6  #include <stdio.h>
7  #include <string.h>
8  #include "agent.h"
9  #include "tclcl.h"
10 #include "packet.h"
11 #include "address.h"
12 #include "ip.h"
13 #include <time.h>
14 #include <algorithm>
15 #include <string>
16 #include <iostream>
17 #include <fstream>
18 #include "flags.h"
19 #include "ip.h"
20 #include "tcp-sink.h"
21 #include "hdr_qs.h"
22 using namespace std;
23
24 class closedPort : public Agent {
25 public:
26         closedPort();
27 protected:
28         virtual void recv(Packet*, Handler*);
29         virtual int command(int argc, const char*const* argv);
30 };
31
32 closedPort::closedPort() : Agent(PT_UDP) {
33
34 }
35
36 static class closedPortClass : public TclClass {
37 public:
38         closedPortClass() : TclClass("Agent/closedPort") {}
39         TclObject* create(int, const char*const*) {
40                 return(new closedPort());
41         }
42 } class_closed_port;
43
44
45 int closedPort::command(int argc, const char*const* argv) {
46       if(argc == 2) {
47           return(TCL_OK);
48       }
49       return(Agent::command(argc, argv));
50 }
51
52 void closedPort::recv(Packet* pkt, Handler*)
53 {
54         // Accessing the packet Headers
55         hdr_tcp *th = hdr_tcp::access(pkt);
56         hdr_ip *iph = hdr_ip::access(pkt);
57 /*
58  * Used to print on a file the information of a packet received on a "Closed Port"
59  *      double t = Scheduler::instance().clock();
60  *
61  *              ofstream myfile ("lostPackets",ios::app);
62  *              if (myfile.is_open())
63  *              {
64  *                      myfile << "Simulation Time:" << t << "    ";
65  *                      myfile << "Packet No:" << th->seqno() << "    ";
66  *                      myfile << "Sent to port:" << iph->dport() << "    ";
67  *                      myfile << "From:" << iph->saddr() << "    ";
68  *                      myfile << "at time:" << th->ts() << "    \n";
69  *                      myfile.close();
70  *              }
71  *              else cout << "Unable to open file";
72  */
73     // Here the received packet is destroyed
74     // HINT: Is similar to the null agent in Ns whose function is to destroy any received packet.
75     Packet::free(pkt);
76 }
77
78
```

A.6 test1.tcl

```
1
2  # Simulation Scenario used to show that when 2 sinks are attached to the same destination port
3  # the server will only reply to one of them (look at section 4.2.4 from the report)
4
5  #Create a simulator object
6  set ns [new Simulator]
7
8  #Open trace files
9  set f [open traceFileTest1.tr w]
10 $ns trace-all $f
11
12 #Open the nam trace file
13 set nf [open traceFileTest1.nam w]
14 $ns namtrace-all $nf
15
16 #Node acts as a receiver.
17 set centralD [$ns node]
18
19 #Nodes act as sources.
20 set client1 [$ns node]
21 set client2 [$ns node]
22
23 #Node acts as a gateway.
24 set G [$ns node]
25
26 $centralD demux
27 $client1 demux
28 $client2 demux
29 $G demux
30
31 #Define different colors for dats flows
32 $ns color 1 red ;# the color of packets from s1
33 $ns color 2 SeaGreen ;# the color of packets from s2
34
35 #Create links between the nodes
36 $ns duplex-link $client1 $G 10Mb 25ms DropTail
37 $ns duplex-link $client2 $G 10Mb 25ms DropTail
38 $ns duplex-link $G $centralD 10Mb 25ms DropTail
39
40 #Define the queue size for the link between node G and r
41 $ns queue-limit $G $centralD 15
42
43 #Define the layout of the topology
44 $ns duplex-link-op $client1 $G orient right-up
45 $ns duplex-link-op $client2 $G orient right
46 $ns duplex-link-op $G $centralD orient right
47
48 #Monitor the queues for links
49 $ns duplex-link-op $client1 $G queuePos 0.5
50 $ns duplex-link-op $client2 $G queuePos 0.5
51 $ns duplex-link-op $G $centralD queuePos 0.5
52
53 #Create a TCP agent and attach it to node s1
54 set tcpC1 [new Agent/TCP]
55 $ns attach-agent $client1 $tcpC1
56 $tcpC1 set dst_addr_ 0
57 $tcpC1 set dst_port_ 5
58 #$tcpC1 set window_ 8
59 #$tcpC1 set fid_ 1
60
61 #Create a TCP agent and attach it to node s2
62 set tcpC2 [new Agent/TCP]
63 $ns attach-agent $client2 $tcpC2
64 $tcpC2 set dst_addr_ 0
65 $tcpC2 set dst_port_ 5
66 #$tcpC2 set window_ 8
67 #$tcpC2 set fid_ 2
68
69 #Create TCP sink agents and attach them to node r
70 set sink1 [new Agent/TCPSink]
71 set sink2 [new Agent/TCPSink]
72 $ns attach-agent $centralD $sink1
73 $centralD attach $sink1 5
74 $ns attach-agent $centralD $sink2
75 $centralD attach $sink2 5
76
77 #Connect the traffic sources with the traffic sinks
78 $ns connect $tcpC1 $sink1
79 $ns connect $tcpC2 $sink2
80
81 #Create FTP applications and attach them to agents
82 set ftpC1 [new Application/Traffic/CBR]
83 $ftpC1 attach-agent $tcpC1
84
85 set ftpC2 [new Application/Traffic/CBR]
86 $ftpC2 attach-agent $tcpC2
87
88 #Define a 'finish' procedure
89 proc finish {} {
90     global ns nf
91     $ns flush-trace
92     #close trace file
93     close $nf
94     #execute nam on the trace file
95     exec ./traceFileTest1.nam &
96     exit 0
97 }
98
99 $ns at 0.0 "$ftpC1 start"
100 $ns at 0.0 "$ftpC2 start"
101 $ns at 1.0 "finish"
102 $ns run
```

A.7 test2.tcl

```
1
2  # Simulation Scenario Using 3 Clients and 3 Sequences on the Server's side
3  #
4  # Client 1 = 2X faster than the Server
5  # Client 2 = 2X slower than the Server
6  # Client 3 = 3X slower than the Server
7  #
8  # In order to simulate case 1, 2, 3 from the report, change "noSequences_ 1" and
9  # comment the following lines at the end:
10 #
11 # $ns at 0.0 { checkTransmission $tcpC2 $ftpC2 100 ; $hoperaaC2 sendContact }
12 # $ns at 0.0 { checkTransmission $tcpC3 $ftpC3 100 ; $hoperaaC3 sendContact }
13
14 # ----------------------------------------------------------------------------------------
15
16 $ns at 0.0 { checkTransmission $tcpC1 $ftpC1 100 ; $hoperaaC1 sendContact }
17
18 #Create a simulator object
19 set ns [new Simulator]
20
21 #Open trace files
22 set f [open droptail-queue-out.tr w]
23 $ns trace-all $f
24
25 #Open the nam trace file
26 set nf [open droptail-queue-out.nam w]
27 $ns namtrace-all $nf
28
29
30 # ------------------------       PROCEDURES USED FOR HOPERAA       ------------------------
31
32 # Procedure which tells you in how many elements each interval will have
33 proc getElemperInterval {no_ports no_intervals} {
34     for { set i 1 } { $i <= ($no_ports/$no_intervals) } { incr i } {
35                 if {$no_ports % $i == 0} {
36                         set temp $i
37                 }
38     }
39     return $temp
40 }
41
42 # Procedure which creates the actual intervals (array-like stored)
43 proc createIntervals {inter no_elem total_ports} {
44 global interval
45    for { set x 1 } { $x <= $inter } { incr x } {
46         puts -nonewline "\n Interval($x):"
47        for { set j 0 } { $j < $no_elem } { incr j } {
48              set temp2 [expr $total_ports/$inter * [expr $x - 1] ]
49              set interval($x,$j) [expr $j + $temp2]
50              puts -nonewline "$interval($x,$j)."
51            }
52             puts "\n"
53    }
54 }
55
56 # Procedure which gives a random number, this is used to Open "Random Guard Ports"
57 proc createRand {no_elem} {
58    set rand_num 0
59    while {$rand_num == 0} {
60        set rand_num [expr {int(rand() * $no_elem + 1)}]
61       }
62    return $rand_num
63 }
64
65 # Procedure which opens One Random port per interval. The procedure calls itself after "tau" units of time, in order to
66 # close the old guardPorts and open new ones.
67 proc setGuardPorts {node agent inter no_elem_int} {
68         global interval closedPortS ns tau
69         set t [$ns now]
70         set t [expr $t + $tau]
71         puts -nonewline "\n -- TIME ([format %.6f [$ns now]]) -- \n\n GuardPort(s) no."
72         for { set x 1 } { $x <= $inter } { incr x } {
73                 set temp [createRand [expr $no_elem_int - 1]]
74                 setPort $node $agent $interval($x,$temp)
75                 $ns at $t "setPort $node $closedPortS $interval($x,$temp)"
76                 puts -nonewline "$interval($x,$temp) "
77         }
78         puts -nonewline " opened in the server. \n\n"
79         $ns at $t " setGuardPorts $node $agent $inter $no_elem_int "
80 }
81 # Support Function to set a guardPort on the Server
82 proc setPort {node agent guardPort} {
83         global ns
84         $node attach $agent $guardPort
85 }
86
```

```
# Closing all the ports in the Server
proc closePorts {node agent port_space} {
   puts "Closing all ports"
   for { set x 0 } { $x <= $port_space } { incr x } {
       $node attach $agent $x
   }
}

# Close single port is used for when the SERVER has to jump to P(i+1) and close P
proc closeSinglePort { node agent portNumber } {
        global ns
        $node attach $agent $portNumber
        puts "      TIME: [format %.6f [$ns now]] - WorkerPort #$portNumber Closed. "
}

# Used to open worker ports at the Server's side
proc openWorkerPort { port } {
        global ns period miu sink2 closedPortS centralD sink1
        puts "      TIME: [format %.6f [$ns now]] - WorkerPort #$port Opened. "
        #Create TCP sink agents and attach them to node r
        $centralD attach $sink1 $port
        $centralD attach $sink2 $port
        #Connect the traffic sources with the traffic sinks
        $ns at [expr [$ns now] + [expr $period + $miu]] "closeSinglePort $centralD $closedPortS $port"
}

# Set Reply value
proc setReply {temp} {
    global replyTCL
    set replyTCL $temp
}

# Opens the port so that the Client can communicate with the server
proc openSession { } {
        global sessionStatus_ ns
        set sessionStatus_ 1
        puts "      TIME: [format %.6f [$ns now]] - Session Opened for Client"
}

proc closeSession { } {
        global sessionStatus_ ns replyTCL
        set sessionStatus_ 0
        set replyTCL 0
        puts "      TIME: [format %.6f [$ns now]] - Session Closed for Client"
}

# Keeps the session running
proc keepSession { } {
    global keepSession
    set keepSession 1
}
# Used to stop the Node's Data transmission when it has sent a centain amount of packets
proc checkTransmission { Client trafficSource totalPackets } {
        global ns totalClients
        if {[$Client set maxseq_] >= [expr $totalPackets + 1]} {
                $trafficSource stop
        }
        if {[$Client set ack_] >= [expr $totalPackets + 1]} {
                puts "Client $Client: Transmission Completed"
                set totalClients [expr $totalClients - 1]
                if { $totalClients == 0 } {
                        finish
                }
        }
        $ns at [expr [$ns now]+0.001] "checkTransmission $Client $trafficSource $totalPackets"
}


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

#r acts as a receiver.
set centralD [$ns node]
$centralD label {Central Directory}
#s1, s2 and s3 act as sources.
set client1 [$ns node]
$client1 label Client1
set client2 [$ns node]
$client2 label Client2
set client3 [$ns node]
$client3 label Client3
#G acts as a gateway.
set G [$ns node]
$G label BottleNeck

$centralD demux
$client1 demux
$client2 demux
$client3 demux
$G demux

```

```
set hoperaaC1 [new Agent/hoperaa]
$ns attach-agent $client1 $hoperaaC1
$hoperaaC1 set dst_addr_ 0
set hoperaaC2 [new Agent/hoperaa]
$ns attach-agent $client2 $hoperaaC2
$hoperaaC2 set dst_addr_ 0
set hoperaaC3 [new Agent/hoperaa]
$ns attach-agent $client3 $hoperaaC3
$hoperaaC3 set dst_addr_ 0
set closedPortS [new Agent/closedPort]
set hoperaaCD [new Agent/hoperaa]
$ns attach-agent $centralD $hoperaaCD

# -------------------------       VARIABLES FOR THE HOPERAA       -------------------------



set total_ports 60
set no_intervals 3
set tau 1
set replyTCL 0
set sessionStatus_ 0
set period 0.3
set miu 0.1
set iAlg_ 1
set bufferTempFlag 0
set totalClients 2
set counter 0


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


#Define different colors for dats flows
$ns color 1 red ;# the color of packets from s1
$ns color 2 SeaGreen ;# the color of packets from s2
$ns color 3 blue ;# the color of packets from s3

#Create links between the nodes (take miu in consideration when setting the links' delay)
$ns duplex-link $client1 $G 100Mb 25ms DropTail
$ns duplex-link $client2 $G 100Mb 25ms DropTail
$ns duplex-link $client3 $G 100Mb 25ms DropTail
$ns duplex-link $G $centralD 100Mb 25ms DropTail

#Define the queue size for the link between node G and r
$ns queue-limit $G $centralD 10000

#Define the layout of the topology
$ns duplex-link-op $client1 $G orient right-up
$ns duplex-link-op $client2 $G orient right
$ns duplex-link-op $client3 $G orient right-down
$ns duplex-link-op $G $centralD orient right

#Create a TCP agent and attach it to node s1
set tcpC1 [new Agent/TCP]
$ns attach-agent $client1 $tcpC1
$tcpC1 set dst_addr_ 0
#$tcpC1 set window_ 8
$tcpC1 set fid_ 1

#Create a TCP agent and attach it to node s2
set tcpC2 [new Agent/TCP]
$ns attach-agent $client2 $tcpC2
$tcpC2 set dst_addr_ 0
#$tcpC2 set window_ 8
$tcpC2 set fid_ 2

#Create a TCP agent and attach it to node s3
set tcpC3 [new Agent/TCP]
$ns attach-agent $client3 $tcpC3
$tcpC3 set window_ 4
#$tcpC3 set dst_addr_ 0
#$tcpC3 set fid_ 3

#Create TCP sink agents and attach them to node r
set sink1 [new Agent/TCPSink]
set sink2 [new Agent/TCPSink]
set sink3 [new Agent/TCPSink]
$ns attach-agent $centralD $sink1
$ns attach-agent $centralD $sink2
$ns attach-agent $centralD $sink3

#Connect the traffic sources with the traffic sinks
$ns connect $tcpC1 $sink1
$ns connect $tcpC2 $sink2
$ns connect $tcpC3 $sink3

#Create FTP applications and attach them to agents
set ftpC1 [new Application/FTP]
$ftpC1 attach-agent $tcpC1

set ftpC2 [new Application/FTP]
$ftpC2 attach-agent $tcpC2

```

```
set ftpC3 [new Application/FTP]
$ftpC3 attach-agent $tcpC3

#Define a 'finish' procedure
proc finish {} {
    global ns nf
    $ns flush-trace
    #close trace file
    close $nf
    #execute nam on the trace file
    exec ./nam droptail-queue-out.nam &
    exit 0
}

# ----------------------------       SETTING GUARD PORTS       ----------------------------

set no_elem_interval [getElemperInterval $total_ports $no_intervals]
createIntervals $no_intervals $no_elem_interval $total_ports
closePorts $centralD $closedPortS $total_ports
$ns at 0.0 {
        # Registering the information from the Clients involved in the simulation
        # registerClient "Client's address" "2 - faster or 1 - Slower" "how many times faster or slower"
        $hoperaaCD registerClient 1 2 2
        $hoperaaCD registerClient 2 1 2
        $hoperaaCD registerClient 3 1 3
        setGuardPorts $centralD $hoperaaCD $no_intervals $no_elem_interval
        $hoperaaCD set periodL_ $period
        $hoperaaCD set maxDeliveryLatency_ $miu
        $hoperaaCD set deltaTime_ 0.1
        $hoperaaCD set totalPorts_ $total_ports
        $hoperaaCD set noSequences_ 3
        $hoperaaCD set noIntervals_ [ expr $no_intervals - 1]
        $hoperaaCD printDetails
        $hoperaaCD startSequence
}

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

$ns at 0.0 { checkTransmission $tcpC1 $ftpC1 100 ; $hoperaaC1 sendContact }
$ns at 0.0 { checkTransmission $tcpC2 $ftpC2 100 ; $hoperaaC2 sendContact }
$ns at 0.0 { checkTransmission $tcpC3 $ftpC3 100 ; $hoperaaC3 sendContact }
$ns at 600.0 "finish"
$ns run

```

Bibliography

[1] Ipoque. “Internet Study 2008/2009”
Retrieved September 24, 2009, from:
http://www.ipoque.com/resources/internet-studies/internet-study-2008_2009

[2] Jelena Mirkovic, Peter Reiher, “D-WARD: A Source-End Defense against Flooding Denial-of-Service Attacks,” IEEE Transactions on Dependable and Secure Computing, vol. 2, no. 3, pp. 216-232, July-Sept. 2005, doi:10.1109/TDSC.2005.35.

[3] Ashfrod, W. “Denial of service attacks on the increase, says C&W,” September 18, 2008.
Retrieved September 24, 2009, from ComputerWeekly:
http://www.computerweekly.com/Articles/2009/09/18/237771/denial-of-service-attacks-on-the-increase-says-cw.htm

[4] Mills, E. “Twitter, Facebook attack targeted one user,” August 06, 2009.
Retrieved September 24, 2009, from CNET:
http://news.cnet.com/8301-27080_3-10305200-245.html

[5] Zhang Fu, Marina Papatriantafilou, Philippas Tsigas, “Mitigating Distributed Denial of Service Attacks in Multiparty Applications in the Presence of Clock Drifts,” srds, pp.63-72, 2008 Symposium on Reliable Distributed Systems, 2008

[6] Pingdom. “The anatomy of a DDoS attack,” March 10, 2009.
Retrieved September 25, 2009, from Royal Pingdom:
http://royal.pingdom.com/2009/03/10/the-anatomy-of-a-ddos-attack/

[7] Benson, P. Greene, R. Jie-ae, S. “U.S. government sites among those hit by cyberattack,” July 8, 2009.
Retrieved September 25, 2009, from CNN:
http://edition.cnn.com/2009/TECH/07/08/government.hacking/index.html

[8] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, Zhichen Xu, "Peer-to-Peer Computing," HP Laboratories, Palo Alto, HPL-2002-57, 2003

[9] Beverly Yang, Hector Garcia-Molina, “Improving Search in Peer-to-Peer Networks,” icdcs, pp.5, 22nd IEEE International Conference on Distributed Computing Systems (ICDCS'02), 2002

[10] “Peer-to-Peer File Sharing: The Effects of File Sharing on a Service Provider's Network,” Sandvine Whitepaper, 2002

[11] Saad N. Ahmad “Business Models of P2P Companies: An outlook of P2P architecture usage in business today,” Humboldt University Berlin, Faculty of Economics and Management Sciences, Berlin, Germany, 2003

[12] William Acosta, Surendar Chandra, “Unstructured peer-to-peer networks - next generation of performance and reliability” in IEEE INFOCOM, March 2005

[13] Pankaj Kohli, Umadevi Ganugula “DDoS Attacks using P2P Networks,” April 2007

[14] Xin Sun, Ruben Torres, Sanjay Rao, "DDoS Attacks by Subverting Membership Management in P2P Systems," npsec, pp.1-6, 2007 3rd IEEE Workshop on Secure Network Protocols, 2007

[15] Giovanni Branca “A Distributed Denial-of-Service (DDoS) Attack using BitTorrent Peer-to Peer (P2P) Network,” presented in Internet Sicherheit (Seminar) at Technische Universität, Berlin, Germany, 2008

[16] Jelena Mirkovic , Peter Reiher, “A taxonomy of DDoS attack and DDoS defense mechanisms,” ACM SIGCOMM Computer Communication Review, v.34 n.2, April 2004, doi: 10.1145/997150.997156.

[17] Coleman, K. “Russia Now 3 and 0 in Cyber Warfare,” January 30, 2009. Retrieved September 29, 2009, from defensetech.org: http://www.defensetech.org/archives/004667.html

[18] Leyden J. “Techwatch weathers DDoS extortion attack,”January 30, 2009. Retrieved September 29, 2009, from The Register: http://www.theregister.co.uk/2009/01/30/techwatch_ddos/

[19] Starkie S. “Twitter is weapon of war on the net,” August 17, 2009. Retrieved September 29, 2009, from: http://www.iomtoday.co.im/business-columns/Twitter-is-weapon-of-war.5560459.jp

[20] InfoSec News, “Police arrest DDoS hackers in Bac Ninh,” August 2, 2006. Retrieved September 29, 2009, from: http://lists.jammed.com/ISN/2006/08/0007.html

[21] Teerawat Issariyakul, Ekram Hossain, “Introduction to Network Simulator NS2,” Springer Publishing Company, Incorporated, 2008

[22] The VINT Project, UC Berkeley, LBL, USC/ISI, and Xerox PARC. “The ns Manual,” Kevin Fall and Kannan Varadhan, Edition 2005

[23] Eitan Altman, Tania Jimenez, “Ns simulator for beginners,” Lecture notes, Univ. de Los Andes, Merida, Venezuela and ESSI, Sophia-Antipolis, France, 2003

[24] Greis M.; *Tutorial for the network simulator Ns*.
Retrieved September 30, 2009 from:
http://www.isi.edu/nsnam/ns/tutorial/

[25] 5th VINT/Ns Simulator Tutorial/Workshop
Retrieved October 14, 2009 from:
http://www.isi.edu/nsnam/ns/ns-tutorial/ucb-tutorial.html

[26] Chung J. Claypool M.; “Ns by Example”
Retrieved October 14, 2009 from:
http://nile.wpi.edu/NS/