P5 - PROJECT

Emil Thougaard Petersen
Frederik Østerby Hansen
Joachim T.H. Nielsen
Palle Thillemann
Ruben Henrik Mensink

Developing a reaction protocol for DDoS mitigation

Supervisor:
Hans Hüttel

Department of Computer Science
Aalborg University
Denmark
21/12-2017

This page was intentionally left blank

AALBORG UNIVERSITET
STUDENTERRAPPORT

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
9220 Aalborg East, DK
Telephone: +45 9940 9940
Telefax: +45 9940 9798
http://www.cs.aau.dk

Title:

Developing a reaction protocol for DDoS mitigation

Theme:

Massively Parallel Systems

Project Period:

6/9-2017 - 21/12-2017

Project Group:

D505e17

Authors:

Emil Thougaard Petersen
Frederik Østerby Hansen
Joachim T.H. Nielsen
Palle Thillemann
Ruben Henrik Mensink

Supervisor:

Hans Hüttel

Total Pages: 91 + frontpage
Appendix: 4
Completion Date: 21/12-2017

Abstract:

Botnets are often used for various malicious activities, one of these being distributed-denial-of-service (DDoS)-attacks. DDoS attacks are often very effective at taking down services, such as web hosts. In this report, we introduce an application layer reaction protocol that is designed to be implemented partly by an internet service provider and partly by clients in order to mitigate attacks. We demonstrate how Mininet can be used to emulate a network and we implement and test a proof-of-concept that shows the viability of our solution. This solution is expected to reduce the effectiveness of DDoS attacks and help clients stay online through the attacks.

Preface

This report concerns the design and development of a reaction protocol for mitigation of DDoS attacks. In this report we will document our work and thought process throughout the analysis of the problem at hand, as well as during the design, implementation, and test of the developed protocol.

Throughout the analysis we explored problems caused by botnets, as well as possible methods to tackle the problems at hand. We chose to focus on a reaction approach with the goal of being able to throttle DDoS traffic aimed at a client. The reaction approach consisted of the design of a reaction protocol and implementation of a proof-of-concept that was tested for its ability to reach the desired throttling of traffic. The design and implementation proved successful in throttling the traffic going toward clients targeted by DDoS attacks. However, the design did not account for the differentiation between regular traffic and DDoS traffic which resulted in the implementation throttling all traffic towards the targeted clients. Additionally, the implemented throttling of traffic is not effective against all types of DDoS attacks.

We would like to express our appreciation to our supervisor for this project, Hans Hüttel, who has given constructive criticism to our work throughout the entirety of the project period. As well as provided us with helpful tools to aid in the design of the developed reaction protocol.

Table of Contents

Chapter 1

Introduction

This chapter will serve as an introduction to the project. Throughout the chapter we will conduct an analysis of the chosen problem, which is botnets, with the goal of stating a problem that we will attempt to solve. This will be done by first presenting the motivation behind the project in section 1.1. Following we will be analyzing the motivating factor of being in control of botnets and how botnets spread through infection in sections 1.2 and 1.3. We will then take a look at the structure and life cycle of a botnet in section 1.4 through a taxonomy. In section 1.5 we will list the areas of interest obtained from the taxonomy. A taxonomy for prevention and detection in sections 1.6 and 1.7 will then be presented. With the insight from these sections we will begin to limit the scope of the project in section 1.8 followed by a problem statement and requirements in sections 1.9 and 1.10, which we will try to answer and fulfill throughout the rest of the project.

1.1 Motivation

There are some distributed systems, that most people are not fond of. These are called *botnets*, which consist of networks of compromised hosts (named bots) used by a *botmaster* for their own gain at the expense of others. Bots execute nefarious code, which has been placed by an attacker, through a myriad of methods. Botnets are frequently used to perform tasks such as distributed-denial-of-service (DDoS)-attacks.

The initiating problems for this project include the following questions.

- What is the motivating factor of being in control of a botnet?
- How can we avoid a host in becoming part of a botnet?
- How can we determine if a host is part of a botnet?

1.2 Profiting from botnets

Botnets are typically large distributed systems with great computational power which can be abused in different ways. Our first look into the world of botnets will be with the purpose of finding out what botnets are generally used (or abused) for and how they can be profitable. We have listed some of the most common uses of botnets. This list is in no particular order.

Distributed Denial of Service (DDoS) attacks A DDoS attack is an attack against a computer system or a network. The computer or network is rendered useless, by not being able to service other hosts, because the DDoS consumes the available resources of the computer, typically in terms of bandwidth. To perform a DDoS attack it is necessary to have a lot of devices, and then have these devices use their bandwidth to overload the bandwidth of a potential victim, which is exactly what a *botmaster*, a person in control of the botnet, has available [1]. There are multiple ways botmasters can make money through DDoS attacks.

- Disrupting business websites and/or networks, where the botmaster will demand money in exchange for stopping the attack [2].
- The botmaster can rent the botnet to a business so that the business can use it to cause problems to a competitor [2].
- Owners of botnets have been known to post adverts of various online forums where any person, for a modest fee, can pay to use the botnet for a DDoS attack, of their own choosing. The costs of renting a botnet for a DDoS attack range from $50 to several thousand dollars [2].

Spamming Botnets can be used to send spam e-mails. It is estimated that 80% of e-mail traffic is spam and that botnets are responsible for the majority of these 80% [2]. The problem with this is that network bandwidth is a finite resource, which causes a burden to other, purposeful and non-harmful, traffic [3]. The way a botmaster makes money off of a botnet is similar to that of a DDoS attack, which is to offer the resources of the botnet to send spam, for someone that wants to pay for that service. It is estimated that in 2009 spammers made $780,000,000 sending e-mail spam [2].

Phishing e-mails and websites Both phishing e-mails and websites have the purpose of tricking the user into giving up personal information, which can be used to gain access to bank accounts, or it can be sold to other cybercriminals [1]. To prevent the phishing website IP address from getting blocked, the botmasters use their botnets as a hosting service. A particular useful property, to botmasters, of these botnets is that they can implement *fast flux*, which allows a phishing website to have the same domain name, but an IP address that changes every few minutes.

Because the IP address changes often, it is difficult to detect the website and make it inaccessible [2].

Stealing personal information Botnets have multiple ways of obtaining personal information from an infected host. It is possible to sniff the traffic of the infected system for usernames, passwords or creditcard numbers. However, if the traffic is encrypted then the botnet malware can employ a keylogger, which records all the keys that a user types, including the aforementioned data [1]. The way a botmaster can make money from this is fairly straightforward. They can either sell the information or use it. In 2007, Brazillian cybercriminals used stolen bank information to withdraw $ 4.74 million from various bank accounts [2].

Google Adsense - Advertising Google Adsense provides a way for website owners to monetize their website traffic, by letting website owners implement advertisements for Google products on their site. Money is earned when visiting users click on the ads. Botmasters are able to abuse this system by using their bots to click on advertisements, which generates money for the website owner. This is also known as *click fraud* [1].

Mining crypto currency Crypto currency has created a new opportunity for botmasters to use the resources of their bots. One of the things a crypto currency, such as bitcoin, requires is that a transaction is verified, which is done through a computation. This process is referred to as *mining*. As a reward for helping with verifying a transaction a user is awarded with bitcoins, at the cost of computational resources. A botmaster can naturally abuse this, since he has a lot of computational resources available in the form of bots. With today's demand for bitcoins, having a large botnet available for mining bitcoins can be quite profitable.

1.3 Infection methods

Botnets are created, spreading and grow larger through infection with malware. Depending on the design of its software, a botnet may infect computers running Windows, MacOS or Linux [1], as well as mobile devices [4] or Internet-of-Things (IoT) devices [5]. IoT devices, such as printers, TVs and smart-fridges have an embedded computer capable of accessing and receiving commands from a network and therefore a valid target for a botnet. While there are many forks and variations of botnets[1], in cases where the botnet is not using a unique solution, the infection methods can be generalized into two approaches.

Worms and viruses

Worms are malicious programs that infect a host by exploiting flaws in legitimate software, which the computer is running [6]. Unlike viruses, which execute by taking control of a host program and spread by sharing of infected files, worms are standalone programs that spread from host to host via the network, without any user interaction. Worms and viruses can be used to deliver other programs to the host, which makes them a useful tool for botnets.

Social engineering

A user may be tricked into downloading and executing the malicious software, via social engineering[7]. The users could be conned into granting control of their computer to a botnet in various ways. From humans pretending to be the help desk of real companies, using their own fake websites, to malicious programs sending mail from a trusted but infected source. If the users are tricked, their actions would bypass any network security measure, rendering otherwise secure systems vulnerable.

1.4 A taxonomy for botnets

In this section we take a general approach to botnets, specifically what kinds of communication topologies botnets use and how botnets operate, which we refer to as their *life cycle*.

1.4.1 Legitimate software topologies

Botnets are a kind of software, and therefore it is not surprising to find that botnets use well known software design patterns, such as *client-server* and *peer-to-peer*.

The client-server topology is a well known pattern, in which a *client* would request a service or a resource from a *server*. A client is defined by the fact that it initiates a connection to the server. The server is defined by its passive state of availability, being ready to service the client and waits for the client to initiate communication. There are many examples of applications that use the client-server topology. An example would be a credit card system, consisting of a card terminal (client) and a server. Multiple client terminals would be able to establish connections to the server, requesting the server to authorize and process transactions.

An example of an application that uses P2P is Skype. It uses the P2P topology to provide communication between its hosts, without the need for a centralized server to establish a connection. P2P is also used extensively for file sharing, since it can very effectively distribute files to a whole network, in that every host, or *peer*, once having a file available can immediately establish a connection with another peer that needs that file.

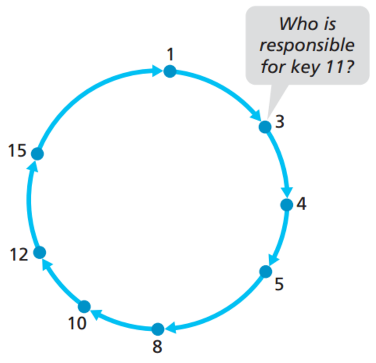
Overlay networks

When it comes to distributed P2P applications, it is necessary to implement a virtual layer on top of another network, which is referred to as an *overlay* or an *overlay network*, so that communication between peers can be organized. There are multiple approaches to overlay networks [3]:

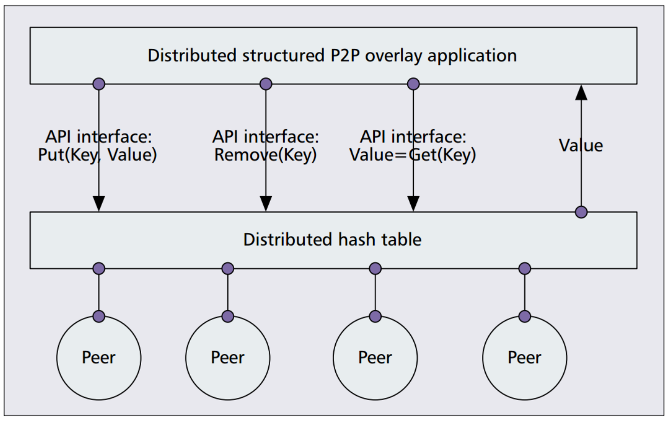
- Unstructured overlay
- Structured overlay
- Superpeer overlay

The peers form *logical* connections to each other, ignoring the actual physical links in the *underlay* network below.

Structured overlay networks are typically implemented with distributed hash tables. Figure 1.1a shows a circular distributed hash table, where each peer is responsible for a certain subset of data, which is accessible through a key, just as with regular hash tables. Since an overlay only provides logical connections, such as peer with key 3 being only connected with peer 1 and 4, we still need IP addresses to acquire a physical connection. Therefore peer 3 of figure 1.1a would need to have the IP address of peer 1 and peer 4 stored locally. These peers would be the neighbours of peer 3. If peer 3 wants to perform a look-up of key 11, it would in the example of figure 1.1a, send the request clockwise to peer 4 and peer 4 to peer 5 until a peer responsible for key 11 would be found. Such message calls would be done through an interface as seen on figure 1.1b. That which makes a structured overlay *structured*, is the fact that the key look-up process is deterministic [8].



(a) An example of a structured network overlay, by using a distributed hash table. The peer with key 3 needs information from key 11, which either peer 10 or 12 could be in possession of, depending on how the overlay would be implemented [9].


Distributed structured P2P overlay application
API interface: Put(Key, Value)
API interface: Remove(Key)
API interface: Value=Get(Key)
Value
Distributed hash table
Peer
Peer
Peer
Peer

(b) Each peer is responsible for a subset of the key space of a distributed hash table. Obtaining data is done through typical function calls, as done with regular hash tables [8].

Figure 1.1

An unstructured approach would therefore not be deterministic in how it sends queries. Peers in an unstructured overlay are typically organized into a graph and the peers form logical connections just as in structured overlays. There are multiple methods of how to forward requests to other peers in the graph, two of which are the following:

- *Flooding*
- *Random walk*

Given a graph and the flooding method, then from the starting point we would send a request to all neighbouring peers, which in turn would send a request to all their neighbouring peers, except to the peer from which the request came [9]. If the graph is connected we would eventually reach the peer that we want to contact.

A random walk of a graph requires a starting point. The starting peer p_1 randomly selects a neighbouring peer p_2. p_2 then selects a random neighbour p_3. This selection of peers (nodes) constitutes a random walk [10].

In a superpeer overlay an individual peer can be selected to be a superpeer, where the total number of superpeers can vary, if that peer satisfies some criteria, such as having a lot of available bandwidth, low latency and high availability. A superpeer takes on the role as a server, which the network of normal peers can use for searching. A peer sends a query to a superpeer, which then processes that query with the help of other superpeers [8].

1.4.2 Botnet implementation of the client-server topology

In the client-server topology the botmaster has access to one or more *command and control* (C&C) servers depending on what specific structure is used. These servers often use the *Internet Relay Chat* (IRC) protocol to communicate, which is an application layer protocol [11]. HTTP or SMTP would be other examples of application layer protocols. The botmaster can, along with the bots, connect to these servers, which allows the botmaster to send commands to the bots via the IRC servers. The bots receive commands from the IRC server, which they will execute locally.

The botmaster relies on servers to communicate with bots. Removing the C&C server would render the bots useless, since they cannot be commanded to do anything. This means that those servers are of interest, when trying to dismantle a botnet using this topology. This is also the reason why it is beneficial to use multiple servers, since this means that if one server were to go down, then the bots connected to it can just connect to another server. A method often used to disrupt these botnets is to target all the C&C servers at the same time.

Within the client-server topology botnets have implemented these topologies with slight variations.

The star topology

The first of the three is called *star*. This is the most simple, where only one C&C server is used, which is the server that all the bots are connected to. Figure 1.2 shows an example of how it is structured.

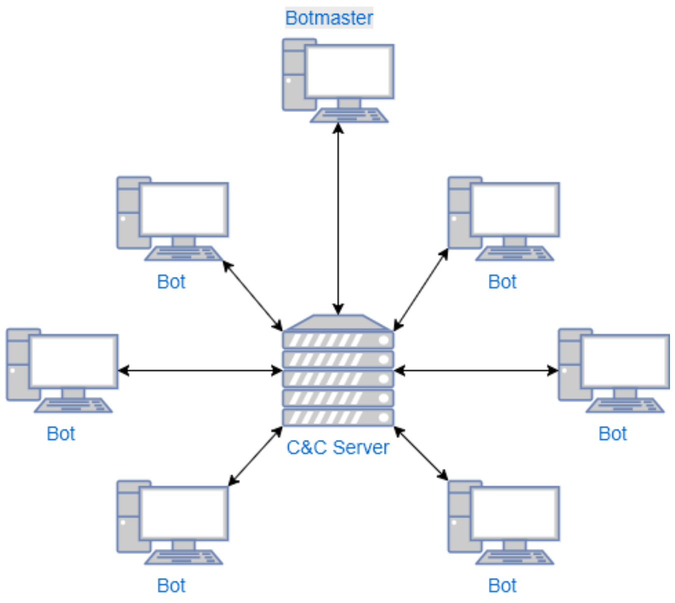


Figure 1.2: A botnet structured according to a Star topology

The multi-server topology

The next topology is *multi-server*, and as the name suggests, uses multiple C&C servers, where the bots are split up between them and can swap to other servers if they lose connection to their current server. Figure 1.3 shows an example of a multi-server setup.

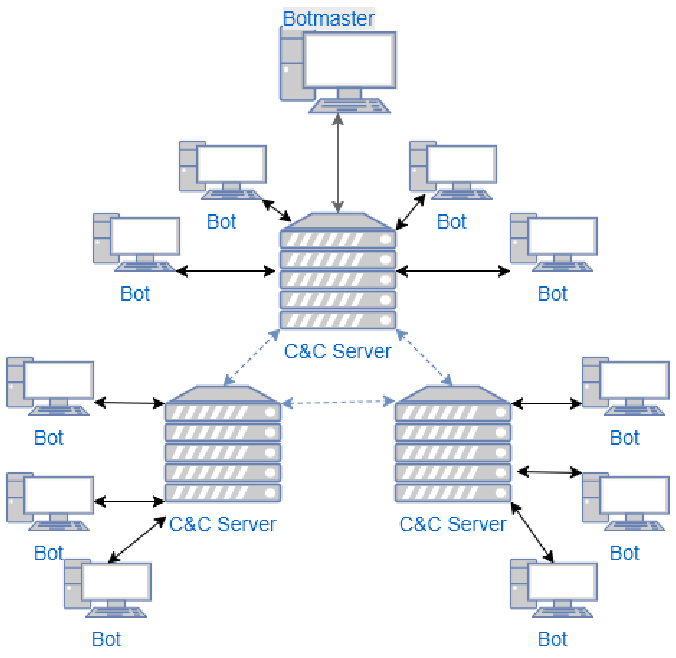


Figure 1.3: A botnet using a topology with multiple C&C servers.

The hierarchical topology

The last variation is *hierarchical*, where each bot can pass on a command sent by the botmaster. This has the effect that bots can connect to each other, which also means that it can take longer for the commands to reach all the bots. A hierarchical structure has the advantage of making it harder to trace through the botnet because all bots are not connected to the C&C servers. Figure 1.4 shows an example of a hierarchy.

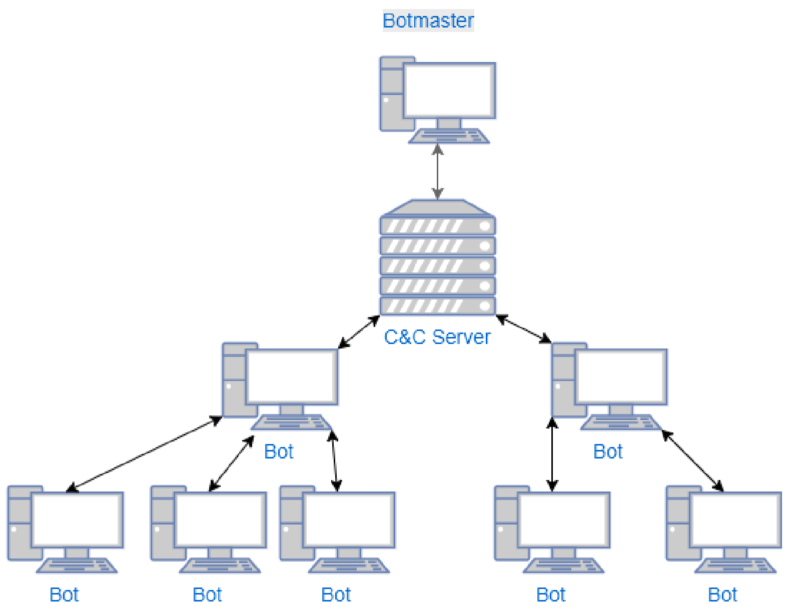


Figure 1.4: A botnet with a hierarchical structure.

1.4.3 Botnet implementation of the peer-to-peer topology

Due to the weaknesses of the client-server topology and the desire for a more robust botnet, botnet designers started to implement a P2P oriented topology. The P2P topology does not use a central C&C server, but has commands "passed along" by each individual bot, each serving as both a bot and a server. Subsequent disruption attempts would be more difficult, since there is no single point of failure [12].

The random topology

The random topology is a specialized class of P2P and as of 2010, no known botnet had implemented it [3]. In this structure, the bots do not actively try to contact the botmaster. The bots sit idle until they receive a command. For the botmaster to send a command he needs to scan around for bots and send the commands whenever he finds any. This is of course inefficient, but also safer since there is no connections between the botnet.

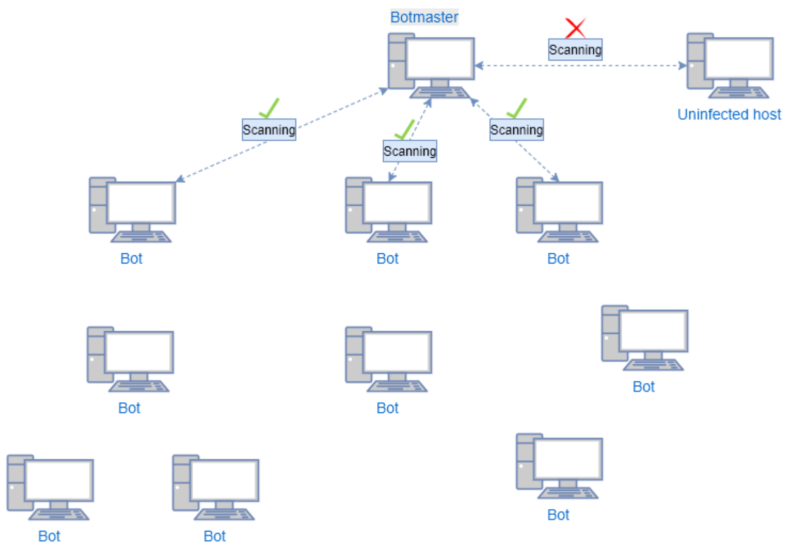


Figure 1.5: A botnet structured randomly.

Constructing P2P networks

The first step in constructing a P2P botnet is to find suitable peers to infect. Three methods are used to find such peers:

- *parasite*
- *leeching*
- *bot-only*

The parasite botnet targets existing P2P networks and tries to find vulnerable hosts within them in order to infect them. This method limits the potential size of the botnet to the size of the existing networks. Leeching looks for vulnerable targets to infect into an existing P2P network. The target can be from either outside or inside the existing network. Lastly, there is the bot-only where the bots build their own network from scratch [3].

In P2P networks when a host (file-sharing networks) or infected host (botnet) needs to become useful to the network, it must become integrated into that network. In the case of structured overlay networks, that would mean that a peer would be assigned a key space and a list of IP addresses of the neighbours of the peer. The following are ways this integration process can be achieved:

- The client software or infection malware contains a hardcoded list of IP addresses, which a peer/bot contact and use to establish who its neighbours are.
- The client software/infection malware contains an IP address not to other peers, but to a shared web cache, which provides the necessary information to update the newly joined host to update its neighbours (peers).

- If an overlay network employs superpeers, then the addresses of the superpeers are hardcoded into the software, and then the superpeers are contacted for the necessary information.

1.4.4 The hybrid topology

Another possible topology is a *hybrid* between P2P and client server. [13] looks into the possibilities of this topology by giving a proposed botnet. This is done to get some insight into this type of botnet before botmasters use it, allowing knowledge of how to battle them if they become used in actual botnets. The proposed botnet consists of two groups of bots. The first group is the bots with a static and non-private IP address which are accessible from the global internet. The bots in the first group are called *servant bots* as they behave as both clients and servers. The other group consists of the remaining bots, these being bots with dynamically allocated IP addresses, bots with private IP addresses and lastly the bots that are behind firewalls, meaning they cannot be connected to from the global internet. The bots in this group are called *client bots* because they will not accept incoming connections. Figure 1.6 shows an example of a hybrid setup.

The botnet communicates with a *peer list* contained in each bot. This list is a fixed list that is limited in size and each bot does not reveal their list to other bots. This means that defenders can only get access to a limited amount of bots if they catch some. The only bots that can be in the peer list are the servant bots since they have server features. The botmaster can issue different commands to the botnet, one being a *report* command. This command will instruct a part of the botnet or all bots to report to a specific machined, called a *sensor host*, controlled by the botmaster. To prevent defenders from targeting this sensor host, its IP address will change with each report command. Another command is the *update* command, which can be used after the report command if necessary. This command will tell all bots to contact a sensor host to update their peer list which will rearrange the layout of the botnet and further help against defenders trying target the botnet. [13]

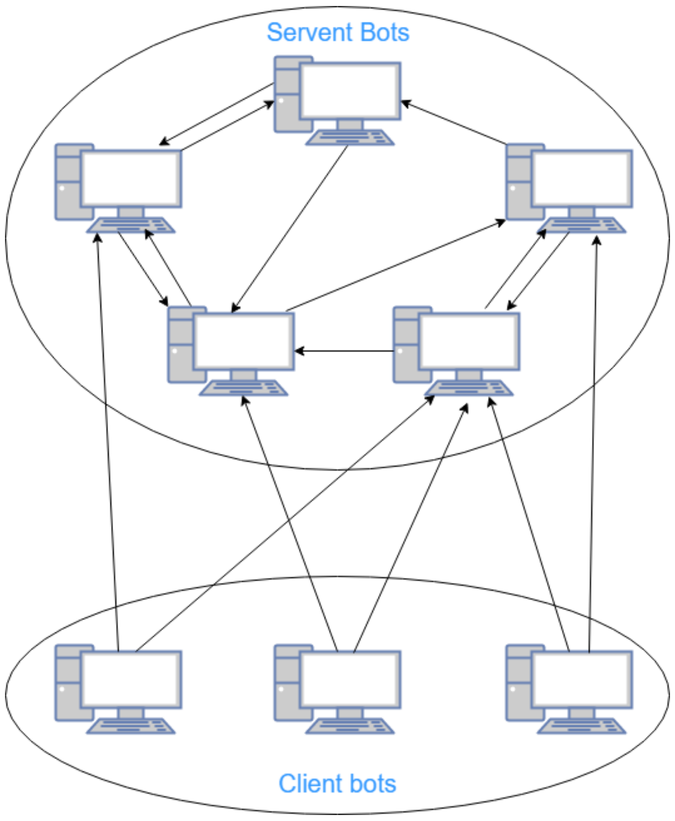


Figure 1.6: A botnet with a hybrid structure.

1.4.5 Evaluation of topologies

Table 1.1 summarizes the different topologies in general, according to their strengths and weaknesses.

Topology	Design Complexity	Detectability	Message Latency	Survivability
Client-server	Low	Medium	Low	Low
Peer-to-peer	Medium	Low	Medium	Medium
Random	Low	High	High	High

Table 1.1: We do not consider the sub topologies of the client-server topology, star, multi-server and hierarchical, but only the general topologies. [14].

1.4.6 The life cycle of a botnet

According to [3], the life cycle of a botnet can be divided into five phases, as seen on figure 1.7.

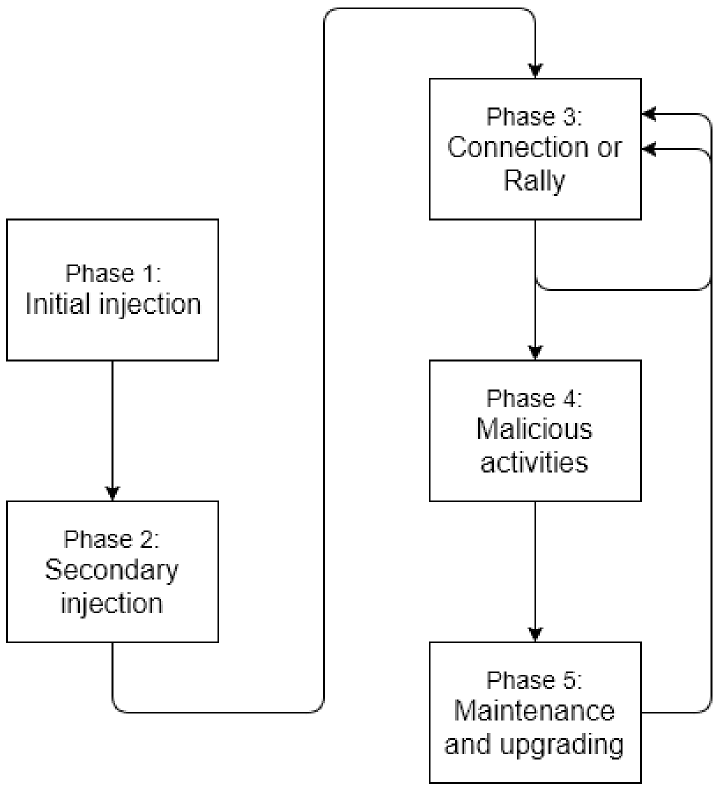


Figure 1.7: The 5 phases [3].

Phase 1 The initiation of the infection phase could be triggered by a host downloading an e-mail attachment that is infected, downloading malware from a website or inserting an infected removable disk into the computer. Infection can happen in a lot of different ways.

Phase 2 This second phase can only happen if the first phase was completed with success. The host, now infected, downloads binaries which are responsible for transforming the infected host into a proper bot. The infection contains a program and a location of a network database, from which the binaries are downloaded. Typical protocols used are the file transfer protocol (FTP), hypertext transfer protocol (HTTP) or a P2P protocol.

Phase 3 This is the phase in which a connection is set up, which allows the botmaster to send instructions to the bot. In a client-server topology this means establishing a connection with a C&C server. If an infected host is turned off it would lose the connection, which means that this phase would need to be repeated. This connection/re-connection is scheduled to happen each time a device is turned back on.

Phase 4 With a C&C connection established, the bot is now in phase 4, where it waits for instructions from the botmaster to start performing malicious activities, such as those mentioned in section 1.2.

Phase 5 The last phase is meant to ensure that the botmaster can keep the malware present in the hosts. The botmaster updates the bot/malware by adding new features, such that the bot can perform multiple types of malicious activities. This is also the phase where the botmaster can have the bots migrate to another C&C server or update code responsible for avoiding detection of the malware.

1.5 Areas of interest

We have analyzed botnets and have identified multiple relevant aspects, when considering how to disrupt the operations of these botnets. These aspects are summarized in table 1.2. In section 1.6 we discuss how to deal with botnets that are in phase 1 of figure 1.7. In section 1.7 we discuss ways of detecting the presence of a bot/botnet. This method is focused on botnets that are in phases 2 to 5.

Communication topologies	**Communication protocols**
Client-server	Internet Relay Chat (IRC)
Peer-to-peer	Hyper Text Transfer Protocol (HTTP)
Random	File Transfer Protocol (FTP)
Hybrid	Simple Mail Transfer Protocol (SMTP)
Social media as C&C server	
Infection methods	**Malicous activities**
Worms (using exploits)	Mining cryptocurrency
Viruses	DDoS
Default credentials - dictionary attack	Sniffing traffic (infected host)
	Keylogging (infected host)
	Send spam e-mail
	Hosting phishing websites (fast flux)
	Advertising/click fraud

Table 1.2: An overview of our previous analysis, which has pointed out the following relevant aspects of botnets, when considering how to address the problem of botnets.

With regards to table 1.2, we chose to include information about topologies, communication protocols, infection methods and malicious activities as these characteristics seem to be the most interesting aspects about botnets, when considering how to disrupt botnets. However, according to [11] botnets include aspects, which we have not mentioned. These are *obfuscation* and *deception*. When botnets communicate, the data that they send are routed through a network and an obfuscation mechanism can be used to hide some of the transmitted information, as well as the malware that is supposed to be executed by the host that receives the data. Deception refers to the activity of hiding the presence of a bot on a host, and a deception mechanism would be some specific implementation that a bot could use.

1.6 A taxonomy for botnet prevention

In section 1.3 we described the general ways in which a botnet is able to infect hosts and make them part of the botnet. From this we know that in order for a host to become part of a botnet it must first be infected by a virus, a worm or through social engineering. Thus by analyzing the mobile code that is downloaded to the host, it is possible to check if the mobile code is malicious and if so prevent the execution of the mobile code entirely.

An attribute common to all the prevention methods we will mention in this section, is that they are able to prevent infection of host devices. This means that they are able to distinguish between software with an intention to infect a host and software that does not have that intention. This will prevent a host from entering phase 1 of figure 1.7.

1.6.1 Program analysis

There are two types of program analysis, static and dynamic. Static analysis is concerned with analyzing a program, without actually running it. This is done by looking into the source code to find potential vulnerabilities and errors. This usually means analyzing source code, object code or similar. Dynamic analysis is concerned with how a program behaves at runtime. This can prove useful for monitoring, and eventually deciding whether or not the program can cause harm to the system. This is where we can monitor if the program is deemed to be part of a botnet. This way we can avoid the spreading of a botnet through *prevention*. We can track what actions the suspicious software performs on the system, such as operating system calls or network traffic, and by comparing this information to known botnet-malware behavior, we can determine whether the executable is malicious or not. All this would be undertaken in a sandboxed environment, where no damage can be done to the host system.

1.6.2 Verifying data integrity

Cautious users may want to validate that when they have downloaded a file, it has not been altered during transmission. Assuming the source is reliable, measures can be taken to ensure the integrity of downloaded files. This altering may simply have been caused by an error. The file may however have been changed by a malicious party as a means of infection.

One way to validate that no changes occurred is by using a checksum. A checksum function is used for simple error detection and may be done using various algoritms. A checksum is generated and attached to the data the function was used on. Once attached the checksum can be used to validate the integrity of the data. However a simple error detection method is seldom used when checking the validity of data transfer to ensure no tampering has occurred during transmission, as generating the same checksum as the original data can be trivial[15].

Instead of using a checksum, one could use a one-way hash functions, or a cryptographic hash functions. One-way hash functions have properties that make them ideal for validating data. Given that they are functions they will produce the same hash given the same data, even a small change would greatly alter the hash. Hash functions are also resistant to collisions, meaning the probability of two varying inputs generating the same hash is low. This makes it difficult to intentionally generate the same hash.

However it is still possible to achieve the same hash with two varying inputs due to the nature of the "birthday paradox"[16], which places an upper bound on collision resistance. This upper bound is dependent on the size of the output.

1.7 A taxonomy for botnet detection

Botnet detection can be achieved via different approaches. In this section we define a taxonomy for botnet detection. We describe monitoring approaches (active and passive) as well as various intrusion detection methods and information gathering. We distinguish between network analysis (global) and endpoint analysis (local), as each have their advantages and disadvantages. We will identify these features of each approach. Detection approaches apply to phases 3, 4 and 5 of a botnet lifecycle, as seen in figure 1.7.

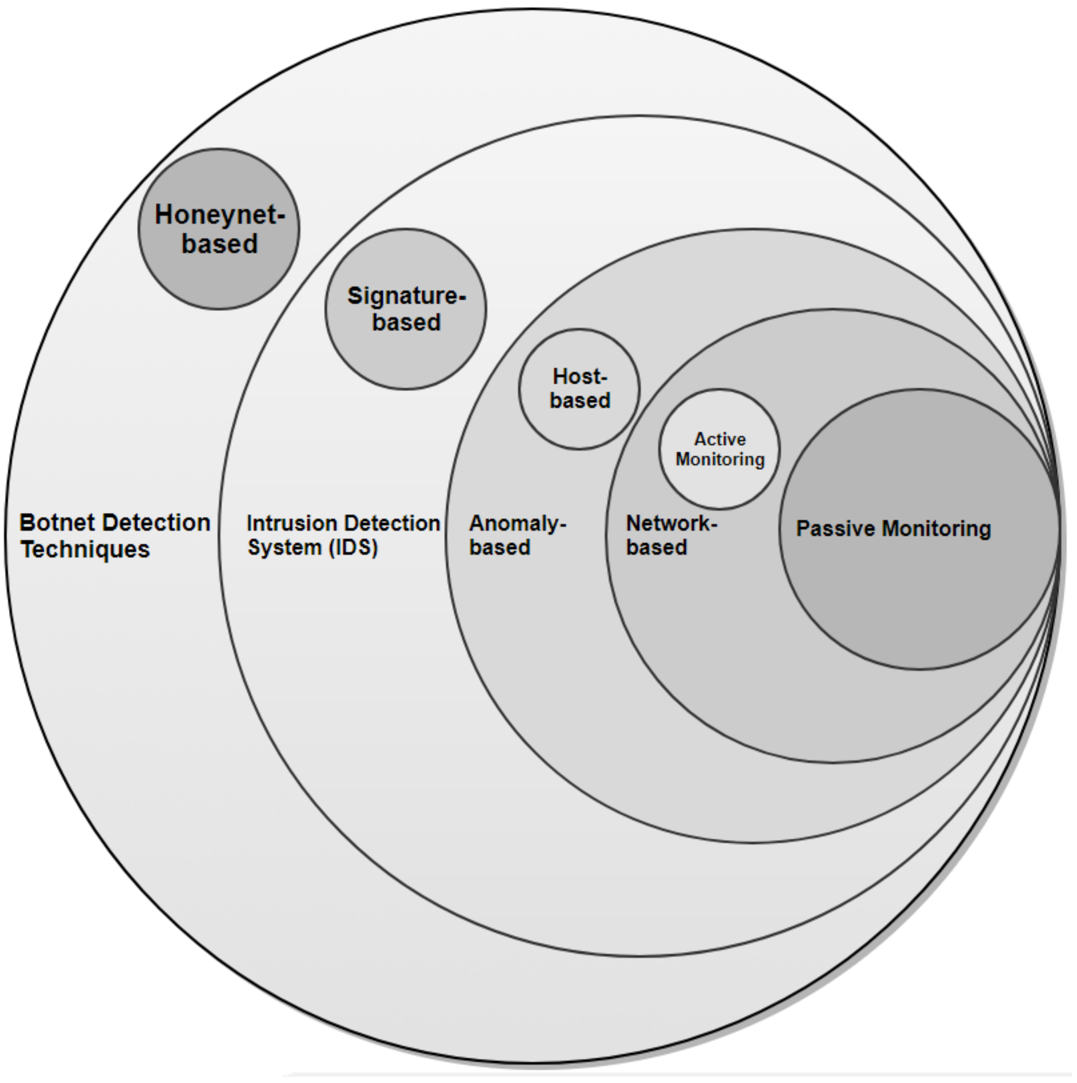


Figure 1.8: Botnet detection techniques diagram, based on table 3 in [3].

1.7.1 Monitoring

There are two approaches to monitoring for the presence of a botnet. Each of these approaches has its downsides and benefits, in respect to performance and legality. When we speak of active and passive monitoring, we are talking exclusively about monitoring a network.

Active monitoring

Active monitoring means that active measures are taken to determine whether a botnet is present. By dynamically sending probe packets to a host, based on the response it is possible to determine whether the host is likely to be part of a botnet. This approach to monitoring relies on a cause-effect correlation. The assumption here is that bots have a deterministic response-pattern. The advantage of using active monitoring is the response time, to determine the presence of a bot. If a bot is present, it only takes one roundtrip to the C&C to get a result. The disadvantage of this approach is that active monitoring might induce increased network traffic. Furthermore there might be legal issues, concerning the injection of packets. [3]

Passive monitoring

Passive monitoring, as the name suggests, does nothing aside from collecting observable data in a network. This monitoring technique looks for suspicious communications, that might be linked to botnet activities. Intrusion detection systems rely on this technique to passively observe traffic. Passive monitoring usually works by implementing the following three steps [17]:

1. Collection of network traffic to a database
2. Analysis of the collected traffic
3. Traffic characterization

To briefly elaborate on these three steps, traffic in the form of packets or packet headers are collected and stored in a database. Then the collected traffic from the database is analyzed to determine if it is normal or malicious. Lastly the traffic is characterized according to the results of the analysis.

In contrast to the active monitoring approach, passive monitoring does not introduce additional network traffic when the network is monitored since it only monitors traffic that is already present. Furthermore with passive monitoring it is possible to collect large amounts of data that can then be analyzed for malicious behavior. For these reasons passive monitoring may be preferable when gathering network traffic for analysis [18].

1.7.2 Intrusion detection systems

Network Intrusion Detection System (NIDS) and Host-based Intrusion Detection System (HIDS) refer to the two domains where we can perform intrusion detection, but they essentially share the same approach. They are both intrusion detection systems, the only difference being where they are deployed. NIDS is deployed on a network level, meaning network traffic from a whole network is being monitored for intrusions. HIDS is deployed on a single host, and therefore knows nothing of traffic on the network, aside from its own. HIDS has the advantage of being able to monitor local system activities, which NIDS cannot. [19]

1.7.3 Honeynet

A honeynet consists of one or more honeypots (hosts). The purpose of a honeynet is to attract bot-infections, in order to monitor the behavior of the malware on the host system. In this way it is possible to monitor what happens when a host gets infected. This makes it possible to see how the malware infects the system, such as changes made to the registry or file system. It also makes it possible to gather information on exactly what the network traffic related to the bot looks like. Honeynets also provide the opportunity to obtain bot malware binaries, and potentially infiltrate the botnet. All this intelligence gathering is very useful in the research and prevention of botnets. Honeynets provide forensic data on botnets, useful in the latter prevention or detection of such botnets. [3]

1.7.4 Network analysis detection

To detect whether or not a botnet is present within a network various detection techniques can be used. These techniques can vary greatly, however most follow some fundamental methodologies.

Signature-based NIDS

Signature-based NIDS is a method to identify the presence of a botnet, based on data signatures from already known botnets. This method assumes that a bot on a botnet has a given behavioral pattern (signature), which has been gathered from examining known infected clients. This can be data produced by a honeynet. This signature is applied comparatively on observed network data. If observed traffic matches a known signature, it can be concluded that botnet activity is present on the network. This method is very effective at revealing known botnets, but does not provide detection for unknown (zero-day) botnets. It is important to note that this method also cannot identify botnets with slightly different patterns, as they will produce different signatures. If this IDS is deployed, a database of signatures would have to be continuously maintained with new signatures. [3]

Anomaly-based NIDS

The anomaly-based IDS approach is concerned with monitoring a network for traffic anomalies, be it abnormal network latency, traffic volume, use of ports or similar. This approach cannot, with complete confidence determine a botnet. But it can indicate anomalies in traffic, that could be related to botnets. Several different techniques can be used to determine abnormalities, most common of which are statistical inference and machine learning methods. The advantage of anomaly-based NIDS is that it provides a detection method for zero-day botnets, and variations of known botnets. The disadvantage is that it can not be certain of botnet presence. Furthermore, applications that produce traffic akin to that of a botnet, can be falsely assessed as a botnet by this approach, such as BitTorrent or gaming related traffic which can cause lots of traffic or use uncommon ports. [3]

Concerning where in the life cycle of a botnet it can be detected by a NIDS, the NIDS can only monitor network traffic. Therefore NIDS cannot observe phase 1 and 2, as seen in figure 1.7, where infections take place. But while the bot is alive, during phase 3, 4 and 5, NIDS can observe network traffic. In respect to anomaly-based NIDS, phase 4 is where one would expect abnormal traffic patterns, and such is the phase where anomaly-based NIDS would prove effective.

1.7.5 Endpoint analysis detection

Endpoint or host based detection techniques analyze the behavior of the host. Below we describe two different methodologies for host-based detection, namely signature-based HIDS and anomaly-based HIDS.

Signature-based HIDS

The signature-based intrusion detection system using a host-based approach is used to identify if a bot is present on a host machine. With this approach the host is monitored by the HIDS and works in a similar way to the signature-based NIDS in that it monitors the host for already known bots. Signature-based HIDS is, like the similar NIDS approach, effective at revealing already known bots on the host device. However it can still not detect bot presence before phase 3 from figure 1.7 since the first chance we have to detect the bot with this method is when it is running. It also falls short in the same way as the signature-based NIDS because it cannot detect unknown (zero-day) bots and is unable to detect slightly different patterns. [3]

Anomaly-based HIDS

The anomaly-based intrusion detection system using a host-based approach works by having the individual host monitored by the HIDS. The host is monitored for any suspicious activity such as accessing suspicious files. As with anomaly-based NIDS this method cannot with complete certainty determine if a bot on the host machine is to blame for any suspicious activity. But can indicate that a bot might be the cause of the suspicious activity. As with the similar NIDS approach there are different ways to determine abnormalities such as statistical analysis and machine learning. Anomaly-based HIDS provides the same advantages as the anomaly-based NIDS approach, but with this method it is still only possible to detect the bot in phase 3 of figure 1.7 since again the bot has to be running on the host device. There are also downsides associated with this approach since it does not scale very well and works on a per machine basis where software must be installed on each individual machine. [3]

1.8 Limiting the scope

In the following section we will limit the scope of our project so that we may state the problem that will be the main focus for our project going forward.

Our analysis started by assessing some malicious activities performed by botnets, in section 1.2, and this section will form a reasonable entry into limitations of our scope. By focusing on a single malicious activity, we may be able to limit the effectiveness of said activity without reducing the use of botnets. The botmasters would simply change the bot to carry out one of the other activities. Therefore limiting our scope based on a malicious activity, would not be a sensible approach

A similar argument can be made with regards to infection methods, as we described in section 1.3. Due to the number of computer viruses and worms, as well as social engineering, it becomes difficult to develop a *catch-all* solution or even a solution that can counter all variations of a single infection method. Therefore limiting the scope based on a single type or kind of infection method, would not be sensible as well.

However, as described in section 1.4, botnets will cause some degree of network traffic, regardless of which botnet topology is used. This leads to the idea of detecting botnets, where there is a plethora of possible solutions available, as described in section 1.7. This means that detection of botnets might not necessarily be the problem we need to concern ourselves with, as existing systems already handle this. Another issue with trying to address the problem of detection, which of course is an important issue, is that it is difficult to develop a *catch-all* kind of solution. Another disadvantage is that for detection to be utilized, it implies that the host is already past phase 2 of figure 1.7 from subsection 1.4.6.

Therefore prevention seems like it would be an alternative approach, which as well contains plenty of possible solutions, since this approach could prevent a host from ever entering phase 1 of a botnet life cycle. However, this method is not a guaranteed *catch-all*, since there are multiple possibilities of circumventing the system in place. In the case of static program analysis from subsection 1.6.1, one could obfuscate the binary file, in such a way, that a static analysis would not be able to characterize the code as malicious [20]. Previously we argued that detecting botnets should not be the problem we focus on solving, as there are many solutions already available. That argument also applies against prevention.

As we know from section 1.7, the theoretical detection methods have certain properties, such as signature based detection being able to detect only known botnets. Therefore one system cannot be used against all types of botnets, which we here refer to known vs. unknown botnets. We see a need for a solution that can adapt to different botnets, by using an already existing detection system. This detection system would therefore need to have a high degree of flexibility through a modular architecture.

While all malicious activities cause distress, most are limited to individually infected hosts and users, such as stealing personal information or mining crypto currency. These can be considered to be on a lesser scale, while DDoS affects whole networks causing congestion and affects the ability for many users to access certain services. We therefore consider DDoS to be a more important problem. This does not fit well with the fact that, if we focus on a single malicious activity then we will not be able to detect botnets using any of the other malicious activities. However, the ability to detect other activities of botnets would likely have to wait for future development in any case. Due to time constraints of this project, our focus will be on botnets using DDoS.

Rather than taking a detection approach or a prevention approach to the problem, we will look at a reaction approach to tackle the problem at hand. That being said detection is still going to play a role in the project, since in order to react we will need to detect the presence of a botnet.

We therefore limit the project scope to concern ourselves with reaction methods, meaning as to how we can perform reactive measures against a discovered botnet. IDS solutions, implementing approaches as described in section 1.7, already exist and the topic has been thoroughly researched already. Therefore, we will focus on developing a form of *reaction protocol*, to be added on top of existing detection methods.

1.9 Problem statement

After the problem analysis and limiting the scope of this project to network analysis of botnet activity, we can now define a problem statement. The main focus of this project going forward will now be examining and solving the problem statement.

Statement

Is it possible to construct a system, that interfaces with at least one current detection solution[1]*, that can, based on the one or more integrated detection systems, monitor a network for DDoS activity, stemming from botnets, and subsequently take appropriate action, that either restricts the amount of DDoS traffic leaving the network or block it entirely?*

[1]Those detection systems described in section 2.2

With this statement some questions arise that need to be answered, these are the following:

- *What method will be used to monitor network traffic?*
- *Which existing detection systems should be used?*
- *On what kind of device should the system be implemented to monitor a whole network?*

We aim to do this, as a means of countering the widespread traffic generated in a DDoS attack. By having a reaction protocol between two or more networks, containing DDoS participants, we can coordinate counter measures to mitigate the effect of the DDoS attack. This way, when a DDoS is observed on a web of networks, the DDoS traffic can be timed out or throttled, effectively rendering the attack weaker or completely stopped.

1.10 Requirements

In this section we will define a set of requirements for our solution. These requirements will form the basis upon which we can determine whether or not the problem statement has been answered. We distinguish between hard and soft requirements, where hard requirements are *need to have* and soft requirements are *nice to have*.

1.10.1 Hard requirements

We define hard requirements, as requirements which have to be met for the problem to be considered solved.

Mitigation	Effective DDoS mitigation, to the point where the victim can operate and serve users, without being taken completely offline.
Invisible to end-user	The solution should not be visible to the end-user. The end-user (clients of a system employing our solution) should not notice a significant change in service, during normal day-to-day operations. Neither should they take any action themselves, after installing components and configuring network.
Dynamic scaling	Our solution should scale to mitigate both small and large sized DDoS attacks equally well. It should be able to handle larger attacks, proportionally similarly to smaller ones, meaning we would see a comparable effectiveness in mitigating an attack, regardless of the size of the attack. This means that we want to see a linear relation between the size of the attack, and the measured effects.

1.10.2 Soft requirements

We define soft requirements, as requirements which do not have to be met, as they are not essential to solving the problem.

Cross platform	Our solution should be able to be deployed on a multitude of different operating systems. Therefore it needs to be engineered to work on Unix, Linux, Windows and MacOS, in other words an OS agnostic solution.
Work with other IDS's	Our solution should be compatible with more than one of the existing intrusion detection solutions.

Chapter 2

Technology Analysis

In this chapter we take an in-depth look at how DDoS attacks can be constructed in section 2.1, for the purpose of gaining knowledge as to how we can most effectively mitigate these attacks. We also analyze and make a decision as to which detection software we will build upon in sections 2.2 and 2.3. This chapter will form the technical basis for the design in chapter 3.

2.1 A taxonomy for DDoS attacks

Since our problem statement concerns the monitoring of DDoS activity the purpose of this section will be to elaborate on what a DDoS attack is and how it works. We will explore which tactics can be employed by DDoS attacks, so that we can effectively mitigate them.

A distributed-denial-of-service (DDoS)-attack is an attempt to make an online service unavailable to users. This can be achieved by temporarily interrupting or suspending the target services of their hosting servers [21].

The form of the attack may vary depending on the target and desired effect on that target. Attacks may exploit weaknesses in the victim system to deny their service or may employ tactics which aim to overwhelm the victim with traffic, thereby hindering their ability to provide their service. We will explore some specific examples of DDoS attacks to gain further insight.

Teardrop attack

A teardrop attack revolves around a fault in the TCP/IP fragmentation reassembly. The fault occurs when IP fragments overlap; this could occur when two fragments from the same IP datagram have offsets that make them overlap each other within the datagram. Some older operating systems are not able to handle these overlaps correctly and may throw exceptions or otherwise fail. A teardrop attack works by first sending a fragment and a part of the attack, then subsequent fragments may overwrite the random data with the rest of the attack. This method is used to avoid detection by IDSs, and is undetectable if fragments are not assembled at the IDS[22].

IP Null Attack

Packet headers contain an identification number to specify which protocol the packets payload uses. The basis of the IP Null Attack is setting this number to null; this allows these packets to bypass firewalls configured for just TCP, UDP etc. If a flood of these packets are sent to the victim, their CPU resources will be wasted on these packets, hindering the victim in servicing legitimate packets[23].

Smurf Attack

The Internet Control Message Protocol (ICMP) is typically used for error messages generated by IP operations; these messages are sent to the source IP address found within the packet which caused the error. However the source IP address within a packet may be *spoofed*, which allows attackers to both hide their own IP address and opens a new avenue for attacks. A smurf attack employs this spoofing of IP addresses, where the source IP address is set to the IP address of the intended victim. The attacker will send out this spoofed packet which requires an ICMP response to either a large amount of IP addresses or a router responsible for a network. This will generate a lot of error messages flooding the victim, crippling the victim's servers[24].

2.1.1 Detection and Mitigation of DDoS traffic

We will now examine current techniques used for detection and mitigation of DDoS traffic, in the hopes of learning more about how and where our solution should be applied. It may seem simple to determine whether or not one is the victim of a DDoS attack, with the large influx of traffic halting normal service. However, this problem may be more nuanced then originally thought, as news about services can spread quickly via social media websites and news aggregators. So a sudden flood of traffic may also cause a *flash crowd* or *flash event*, referring to the situation of a large amount of people all going to the same place within a short time. Therefore one would have to be able to distinguish between legitimate users and attackers. A study by [25] looked at how to differentiate between flash crowds and DDoS attacks, they found that during a flash event a high percentile of requests were made by users which had previously accessed the site before. Whereas during a DDoS attack requests were made by new users which had not accessed the site before. They also observed that a large amount of the requests during a DDoS attack were given the response code 401 Unauthorized, meaning that pages which required a password were requested. While both during a flash event and a DDoS attack most requests were made for around 10% of the most popular pages on the site, however during the DDoS attack each user sent around 800-1000 requests per minute.

Ingress filtering

As many DDoS attacks use packet spoofing in a variety of ways such as the aforementioned smurf attack and may also just be used to hide the address of the attack. As a way to counter this there is a technique called Ingress Filtering. Ingress filtering may be applied at a router for a network or at the ISP level. It works by checking each outgoing packet against a list of all IP addresses within its subnet or the list of IPs provided by the ISP. So if the source IP address within the packet is not in this list the packet is dropped[26].

2.1.2 Summary

There are three points where attacks can be detected and defended against at: source-end, intermediate, victim-end. There is an inverse correlation between detection and defense, meaning it is easier to detect a DDoS at the victim-end, however it is harder to mitigate the attack[27]. On the other end it is harder to detect an attack from the source-end, but it is easier to mitigate. Most detection methods operate at the victim-end, however the disadvantage as mentioned before is that it is harder to defend against the attack at this point, as it can be very resource intensive.

We know from this section that the IDS we choose to use for our protocol should ideally be able to detect the different attacks including those listed throughout the section. This will help ensure that we detect as many DDoS attacks as possible and thus as many attacks as possible can be mitigated by the protocol. Furthermore if we can distinguish between the different attacks we might be able to mitigate them more effectively.

2.2 Existing software solutions

IDS and DDoS detection systems are nothing new, and many products already exist. Some of these are commercial products, and others are open-source. We find it important to get an overview of existing solutions. Therefore in this section we look at each of these products, their capabilities as well as which detection approach they take (based on approaches described in section 1.7 and knowledge from section 2.1)

2.2.1 FastNetMon (DDoS detection)

FastNetMon is a DoS/DDoS analyzer. FastNetMon exists as a commercial product as well as an open-source community edition. The commercial version has an extended feature set, which we will not look at. It relies on multiple *packet capture engines* to capture traffic. It claims to be able to detect DOS attacks in 1-2 seconds, and supports many network environments. It is supported on most Linux distros, FreeBSD and MacOS. FastNetMon can detect the most common types of DoS/DDoS attack methods, including SYN-, UDP-, ICMP- and IP fragmentation-flood. The documentation contains no info concerning which specific detection method FastNetMon utilizes, but it seems to take a passive monitoring approach, since it monitors data as it flows through the network. As far as we can tell, it uses an anomaly-based approach to detect DoS traffic. [28]

2.2.2 FlowTraq (Anomaly-based traffic detection / DDoS detection & mitigation)

FlowTraq is a commercial suite of network analysis tools. FlowTraq provides botnet detection capabilities, using an anomaly-based approach, powered by machine learning algorithms. FlowTraq builds a *behavioral fingerprint* of network behavior. Network Behavioral Anomaly Detection (NBAD) is then used to detect anomalies in traffic, which might be unusual, interesting, or possibly malicious. Furthermore, FlowTraq can detect DDoS attacks, and take action to mitigate it, by picking and using existing DDoS mitigation solutions. Both hardware and software DDoS solutions are supported. As this is a commercial product, concrete information about the FlowTraq's implementation is not available. [29]

2.2.3 Bro (IDS framework)

Bro is an open-source framework that supports signature-based and anomaly-based approaches [30]. It achieves this by passively monitoring the network traffic, flagging known malware and producing a set of log files for external programs to further analyze. Bro also facilitates the automation of tasks via its scripting language, making it possible for external programs or custom scripts to respond to events generated by Bro. As such, the architecture of Bro is layered into two modules: the *event engine*, which generates events based on the traffic being analyzed and the *script interpreter*, which executes Bro's own event handlers, as well as any custom written handlers. This design allows Bro to be the framework for a custom traffic analyzer, rather than a hardcoded NIDS [31]. By itself, Bro does not offer any specific way of detecting botnets or DDoS, instead having to rely on third-party programs or custom scripts for detection.

2.2.4 OSSEC

OSSEC is a free and open-source HIDS. OSSEC runs on all common platforms, UNIX, MacOS, Windows and common Linux distros. OSSEC will monitor system logs, check file integrity as well as run rootkit detection. It also allows for *active response*, meaning immediate actions can be taken if a threat is detected, based on the configuration. It offers a centralized management system, where many hosts on a network can be managed, across many platforms. This solution seems to be based on passive monitoring. The documentation mentions no specific abilities to handle botnets or DDoS [32].

2.2.5 Snort

Snort is an open-source NIDS, capable of real-time traffic analysis, packet logging, protocol analysis and can search/match content [33]. It is also able to detect probes such as stealth port scans. Snort can be used in three modes: sniffer, packet logger and network intrusion detection [34]. A highly desirable feature of Snort is its rules. These rules allow for implementation of various detection techniques. The anatomy of Snort rules is divided into two sections. The first section is the rule header, which describes what will trigger the rule, such as the flow of network traffic, IP addresses, ports, etc. The header also includes which action should be taken if the rule is triggered. Rule options make up the second section of the rule. This section may contain additional criteria and what kind of output is expected from Snort when the rule is triggered. In listing 2.1 we see an example of a Snort rule. The first part of the rule, up until the first parentheses, is the rule header. In this example the rule should executed if there is any TCP traffic from any IP address and port, to the destination *130.225.198.224/32* on port *80*. Within the parentheses are the rule options. In the example it will check if the packet, which was the rule header, contains a string "GET", in which case an alert with message *"WWW GET detected"* is sent to the user[35].

```
alert tcp any any -> 130.225.198.224/32 80
   (content:"GET"; msg:"WWW GET detected"; sid:1;
   rev:1;)
```

Listing 2.1: Example of Snort rule

2.2.6 Hogzilla (Anomaly-based traffic detection)

Hogzilla is an open-source NIDS, using an anomaly-based detection approach [36]. It offers interoperability with Snort and graphic tools to make the results of its analysis more human-readable. Hogzilla has bad documentation available, so the basic information concerning its capabilities and features are not readily available to us, without digging into the implementation [37].

2.2.7 Suricata

Suricata is an open-source signature-based NIDS, and like Snort, it too implements the use of rules [38]. The main innovation of Suricata is its use of multithreading, using the processing power of multiple cores to speed up traffic analyzing [39]. Apart from the focus on efficiency, Suricata offers the same tools as Snort, which are used for detecting botnets or DDoS.

2.3 Choosing an IDS

In our problem statement we decided that we would use one or more existing software solutions for detection. Because of this the purpose of this section is to choose one or more of the existing detection solutions described in section 2.2 and use that going forward.

For us to compare the different IDS, we specify the following requirements:

- The IDS must be open-source, as we may need access to the source code of the IDS to facilitate the integration with our solution.
- The IDS must have good documentation available to minimize time devoted to discovery of its capabilities.
- The IDS must be network based.
- The IDS must be able to detect a DDoS attack.
- Ideally be able to perform real-time analysis of traffic as per the spontaneous nature of DDoS attacks.

Programs	Open-source	Documentation	Network based
FastNetMon	X	X	X
FlowTraq		X	X
Bro	X	X	X
OSSEC	X	X	
Snort	X	X	X
Hogzilla	X		X
Suricata	X	X	X

Table 2.1: Comparison of IDS

After the IDS comparison, we are left with four IDS that fulfill our requirements: FastNetMon, Bro, Snort and Suricata. All the examined software is capable of real-time analysis and DDoS detection, however there are some differences that can help us decide. When it comes to DDoS detection, FastNetMon needs the threshold for unusual traffic set manually, while both Bro and Snort have code and rules pre-made and available online, written by people with more experience then us. Suricata has no such premade rules, leaving us no choice but to write all the DDoS detection rules ourselves. Given the option, we would rather choose software with already existing solutions for DDoS detection.

Choosing between FastNetMon, Bro and Snort became a discussion about which software would be easiest to integrate our solution with. We ultimately chose *FastNetMon* as our detection software, due to the software being more focused on DDoS detection than the wider thread detection of Bro and Snort. We therefore would not need to implement software with features we do not need.

2.4 Placement of an IDS in a network

In order to better understand what is meant by the term placement in this context, we consider figure 2.1.

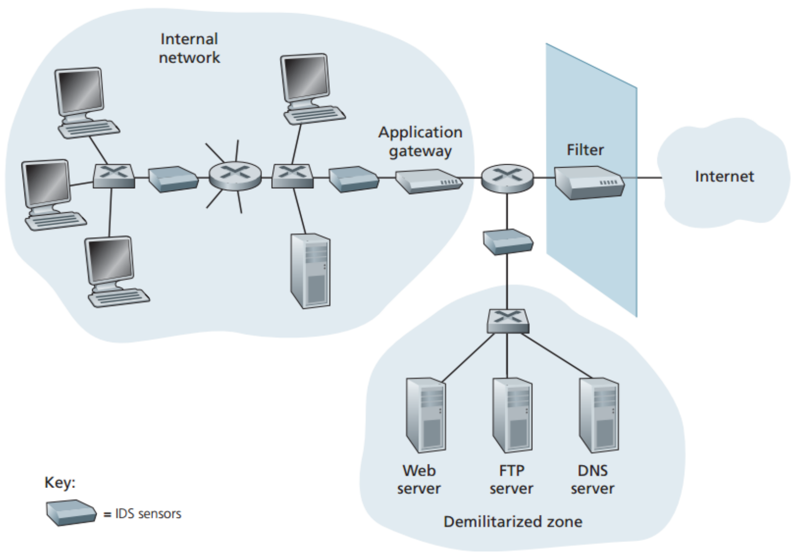


Figure 2.1: Placement of an IDS [9].

In this example there are three IDS *sensors* placed between various other network components. The sensor analyzes the incoming traffic and sends alerts to the IDS, which can take input from multiple sensors. The minimum amount of sensors required is one. The typical placement of the sensors is as illustrated on figure 2.1, where there is no sensor outside of the filter (firewall), which has the advantage that the sensor does not receive and subsequently has to process all information coming from the outside. With regards to testing in a closed environment, we would not need multiple sensors nor would we need a firewall, since we can control all generated traffic.

Chapter 3

Design

In this chapter we describe the design of our solution. In section 3.1 we will look at the concept of how the protocol should work once finished and give a brief overview of what steps the solution takes to help mitigate a DDoS attack. In section 3.2 we look at the difference between open and closed systems and discuss what we have to keep in mind about these in regards to our solution. In section 3.3 we look at different ways for our solution to throttle the traffic. Section 3.4 describes how multiparty protocols work and provides protocol examples using arrow notation, message sequence chart and scribble. Lastly in section 3.5 we describe the actual design of the protocol for our solution.

3.1 Concept

The main idea behind our solution is that we counter DDoS attacks on the attacker's side, as opposed to the idea behind known anti-DDoS systems, where the victim has to employ proxies or similar, in order to move the target away from itself. Our goal is to handle the DDoS traffic on its way to the victim, as opposed to when it reaches the victim. This means that our solution is meant to be built into the structure of the internet. This means that our solution acts as part of the internet, as we expect it to be implemented at the ISP level (Internet Service Provider).

For our concept to work, we have to make some assumptions about the internet:

- Every ISP carries our protocol. Because, if not, then a subset of bots on a botnet will not be affected by this protocol, and can therefore attack the victim without being affected by our protocol.
- Bots only reside behind ISPs.

In figure 3.1, we see a diagram showing two groups of attacker bots, each residing behind their respective ISPs, A and B. We also see non-malicious clients connected to the server through ISP C. The bots start a DDoS attack on the victim. The victim is running a DDoS detection system, which detects the attack, which in turn will trigger our protocol (on the victim) to send out a *panic* message, to ISP A, B and C. These ISPs will then throttle traffic going to the victim, and thereby mitigating the DDoS attack. Our protocol should make it possible for the victim and the ISPs to communicate when the victim is being attacked. This behavior is represented by the green arrows in figure 3.1. In this example the parties of the protocol are the victim and ISP A, B and C.

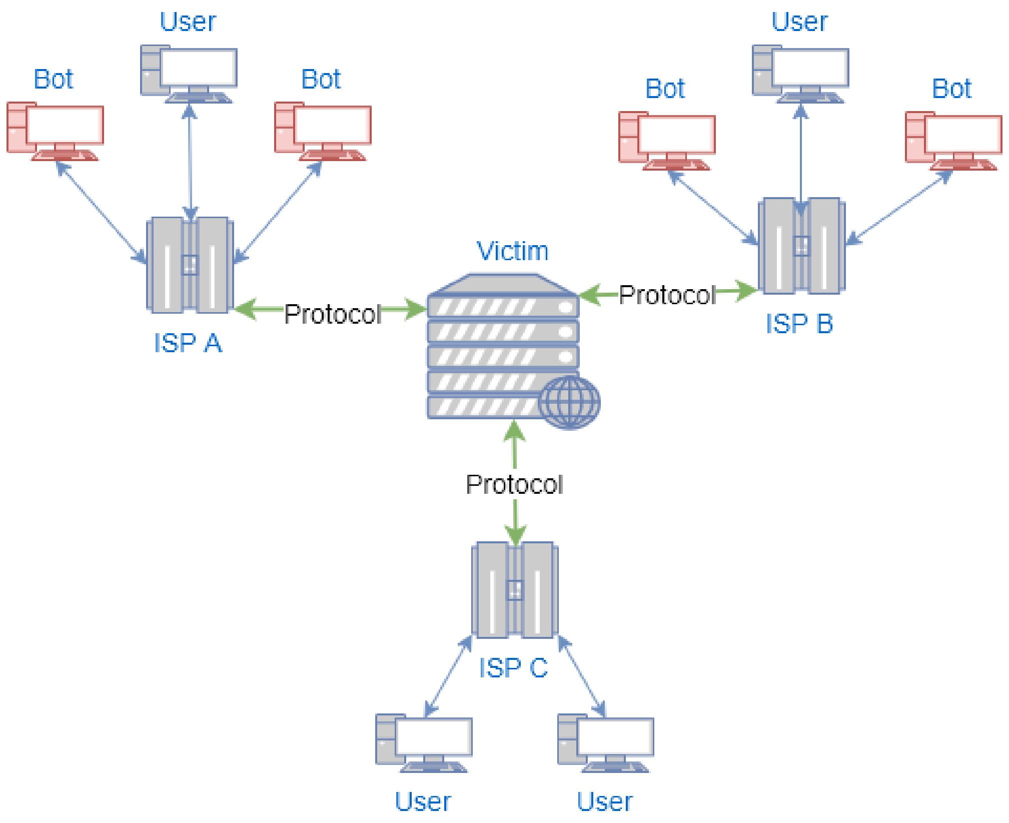


Figure 3.1: An overview of how our solution works.

As this system does not know the difference between an attacker bot and a regular user, the regular users on ISP C will be treated similarly to bots. Our protocol forms a decentralized network, consisting of ISP A, B, C and the victim. In the following section we will describe in-depth how this solution will work in a network, as well as a formal definition of the protocol.

3.2 Open and closed systems

To help facilitate the testing of our protocol and to help understand the behavior of a distributed system, it is worth clarifying the problem with testing a open system versus a closed system.

- A *closed system* is a system that does not communicate with or receive information from outside itself. An example of a closed system could be a single host with no network access. However, an entire network can also be considered closed, even though it consists of multiple hosts communicating [40]. The internal network of a company where the network is used to share files, send emails and print requests within the company, is only a closed network if there are no connections to hosts or networks outside of the company, as their behavior would be beyond the control of the company.
- An *open system* is a system that communicates with or receives information from outside of itself. Any computer or smartphone with internet access is an open system and any network capable of communicating with other networks is an open network [40].

The problem with testing our protocol on an open system lies in writing a test capable of compensating for the effects of actions from unknown actors on the internet. With a closed system, the actors and effects can be known and tested for, but the results will not be true for an open system. If our protocol is to work on the internet, an open system, then we have to design our test and network by making assumptions about the traffic from the internet, based on what is important for our protocol.

Assumption A The traffic is generated by a DDoS attack targeting a client of an ISP with our protocol active.

Assumption B The traffic contains legitimate requests from users, asking for the service provided by the client.

By making these assumptions, we can build a test system that is closed, but emulates an open system.

3.3 Throttling methods

Traffic targeted towards the victim needs to be throttled, in order to mitigate the attack. In this subsection we will discuss a few different approaches towards throttling from an ongoing attack. There are multiple issues to address and each approach has its strengths and weaknesses. In this subsection, when talking about *percentages*, we are purposefully vague, because these variables will need to be tuned in our implementation. Our approaches take inspiration from the TCP's congestion control algorithm [9].

Cooldown

In figure 3.2, we describe the behavior of this method, by plotting the allowed traffic towards the victim as a function of time. When our protocol is triggered, and the panic signal is propagated to the ISP, we start throttling traffic. At first we throttle down to a predefined throughput, for example 10%. Slowly we raise the allowed throughput, until we again receive a new panic signal. We reset to the predefined default throttle percentage, and again slowly increase the allowed traffic. This way we put a hard limitation on the traffic when a new attack is detected, and slowly ease this limitation. We apply *slow start* at the beginning, meaning when we throttle the throughput, we will start slowly to increase it over time, giving the graph a curve similar to what is depicted in the figure.

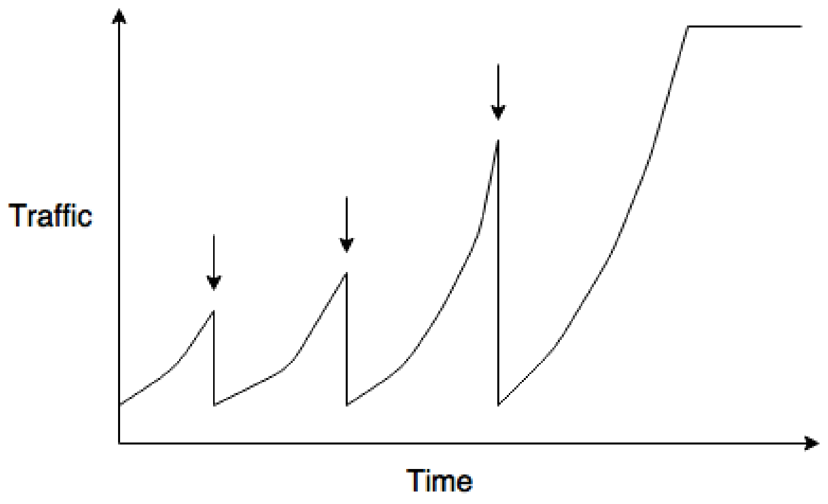


Figure 3.2: Cooldown approach

In figure 3.2, each arrow indicates a new panic signal. Therefore when a new panic occurs, the throttling is reset, meaning we limit the allowed throughput back down to a defined percentage. This pattern of behavior is repeated until a panic signal is not received again, and allowed traffic throughput returns back to 100%.

This approach has one major disadvantage. If, at any point the victim becomes unresponsive, or for any reason cannot send out a panic signal, the system will assume that everything is returning back to normal, and ease up on the throttling, which might flood the victim further. Therefore, we attempt to solve this weakness by introducing a threshold.

Cooldown with a threshold

If a victim is experiencing an attack, it might not always be able to send out panic signals during the attack, or it might be too slow to do so in proper time. To accommodate this, there is a different spin on the cooldown approach, as seen in figure 3.3. We introduce a threshold, so that we ease the traffic limitations slowly, just like previously, but when a threshold is reached, we make a decision. If we have received an *okay*-signal (indicated by the arrow) from the victim, we know that the attack is no longer active, and we can slowly ease back the throttling. If we have not received anything from the victim, we assume that the attack is still ongoing, and therefore reset the throttling to a predefined percentage. If, during the attack we receive new panic signals, we reset the throttle amount back down, and lower the threshold by some amount. This way, if the applied throttling and threshold is not aggressive enough, and the victim sends out further panic signals, we will adjust the threshold variable to accommodate this.

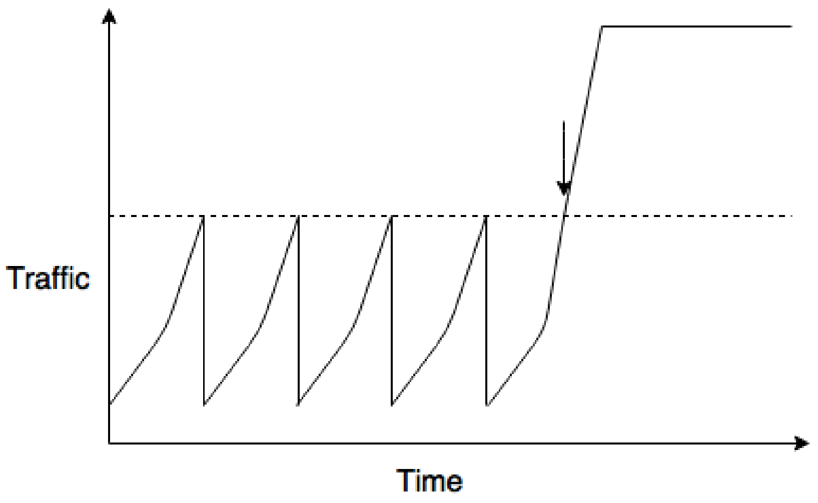


Figure 3.3: Cooldown with a threshold approach

As previously, we use *slow start*, at the beginning of each reset. If an *okay* signal has been received, and we have reached the threshold, we use a linear increase in traffic throughput. This means that when we know that the victim is okay, we can ease back on the throttling in a faster manner, than when we do not know how the victim is doing.

We will use the latter approach, *cooldown with a threshold* in our implementation. But we might need to change or adjust certain aspects of it, when we get to the implementation and testing phase. Therefore the specifics of our throttling technique is subject to change.

3.4 Describing multiparty protocols

In order to solve our problem we are going to need to construct a protocol that can be implemented by both ISPs and their clients. From this point on:

A **protocol** *represents an agreement on how participating systems interact with each other.* [41]

It is also important to distinguish between two types of protocols, namely *global* and *local* protocols. A global protocol is described from an endpoint neutral perspective. Meaning that the role of each participant of the protocol is described in the same protocol. A local protocol, also known as an endpoint protocol, describes the role of one of the participants of the global protocol. This means that a global protocol with two participating systems will have two local protocols, one for each participant.

We now present the different ways we will describe our protocols. We will use the following three notations in the presented order:

1. Arrow notation
The arrow notation describes the individual steps of a global protocol by describing interactions between the participants of the protocol. This is an informal notation.

2. Message sequence charts
With the overview of a given protocol provided by the arrow notation we can then describe the protocol with message sequence charts. This is a formal and graphical notation.

3. Scribble
With the language and tool Scribble we will provide an implementation specific description of how the given protocol will work. This is a formal language with a syntax, operational semantics and a type system.

3.4.1 Arrow notation for describing protocols

The purpose of this notation is to give a structured overview over the global protocol in question. The notation will provide a step-by-step walk-through of the global protocol which will make use of a syntax described in this subsection. The syntax for this notation can be seen in table 3.1 below. In the syntax we make use of the following metavariables:

S – Metavariable for statements in this notation

P – Metavariable for participants that take part in the global protocol

M – Metavariable for messages sent between the participants of the global protocol

After the syntax has been presented and explained we will write an example global protocol making use of the given syntax, the syntax can be seen in table 3.1.

S	$::= \; 1.\; P_{1i} \rightarrow P_{2i} : M_1, \ldots, n.\; P_{1n} \rightarrow P_{2n} : M_n$

Table 3.1: Syntax for the arrow notation

As is seen the syntax allows for multiple statements S which make up the numbered steps in the global protocol. For each step we also have a message M_i to be sent from P_{1i} to P_{2i}. P_{1i} and P_{2i} are nonempty sets of participants where $1 \leq i \leq n$. The last statement also represent the end of interactions between participants. The message M_i can be one of the following:

$\mathcal{P}-$**Panic**: A panic signal sent by a *client* to an *ISP*.

$ST-$**Stop Throttle**: Signal sent by *client* used when throttling is to be stopped.

$B-$**Broadcast**: Used when an *ISP* broadcasts signals to other *ISPs*, $B(\mathcal{P})$ or $B(ST)$.

$A-$**Acknowledge**: Response that acknowledges a message M has been received.

Protocol example

With the syntax for the arrow notation explained we can now give an example of how it can be used to describe a protocol. The example that we are about to give will be expanded upon throughout this section with the other notations that we have not yet presented. The example will consist of a short description of what the purpose of the protocol is, supplemented by use of the arrow notation for the parts of the protocol that it can describe. These parts are also global protocols themselves and we will refer to them as sub-protocols.

In this global protocol the *client* will make a choice on what message to send to its *ISP* based on if the *client* is the target of a DDoS-attack or not. This leads us to the two following descriptions for each sub-protocol and their arrow notations:

Under attack

If the *client* is a victim of a DDoS attack it will send a panic signal $\mathcal{P}$. The *ISP* will thereafter send back an acknowledgement, A, and broadcast the panic signal, $B(\mathcal{P})$, to the other *ISPs* that participate in the protocol. Each of the *ISPs* will send back an acknowledgement to the broadcasting ISP. Table 3.2 shows this sub-protocol using the arrow notation.

Not under attack

If the *client* is not under attack it will send the signal, ST, to its *ISP* informing the *ISP* that the attack is over and the throttling of traffic can be ended. The *ISP* acknowledges this and broadcasts this signal to the other *ISPs* as has previously been done with the panic signal. Table 3.3 shows the sub-protocol just described using the arrow notation.

1.	*Client* $\rightarrow$ *ISP*	$\mathcal{P}$
2.	*ISP* $\rightarrow$ *Client*	A
3.	*ISP* $\rightarrow$ *ISPGroup*	$B(\mathcal{P})$
4.	*ISPGroup* $\rightarrow$ *ISP*	A

Table 3.2: Arrow notation for the sub-protocol where the client is being attacked

1.	*Client* $\rightarrow$ *ISP*	ST
2.	*ISP* $\rightarrow$ *Client*	A
3.	*ISP* $\rightarrow$ *ISPGroup*	$B(ST)$
4.	*ISPGroup* $\rightarrow$ *ISP*	A

Table 3.3: Arrow notation for the sub-protocol where the client is not being attacked

Once the sub-protocol from either table 3.2 or 3.3 has been executed the *client* will make the choice on what message to send to the *ISP* again, thus making the protocol recursive.

To make the arrow notation more versatile we might at a later point add more messages to M, as this will greatly increase what global protocols we can describe with this notation. When this is done we will briefly explain the new messages added and assign a letter to represent these, as has been done in this section.

3.4.2 Message sequence charts

We will make use of message sequence charts (MSC) in order to visualize the interactions that take place between the individual participants in the protocol on the global level.

A MSC with the structure we are going to use later can be seen in figure 3.4. The figure depicts a protocol with two participants where a message and a response are sent. Each participant in the MSC has a *time line* followed from top to bottom. We will also make use of *loops* to model recursion and *alternatives* to model choice and branching. These appear as frames in a MSC labeled **loop** and **alt** respectively.

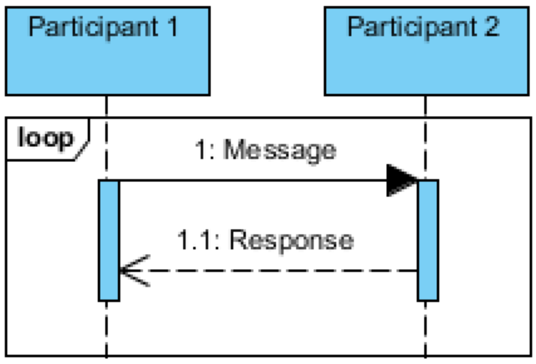


Figure 3.4: Structure of a MSC

Protocol example

We will now continue the example from section 3.4.1. Here we will translate the global protocol supplemented by the arrow notation into a MSC. This can simply be done by making a time line for each participant from the protocol. In this case there are 3 participants, namely *Client*, *ISP* and *ISPGroup*. Since it is possible to model choice with **alt** we can have the two arrow notation sub-protocols from tables 3.2 and 3.3 in the same MSC by having each of them in their own section of the **alt** frame.

Since we also make use of recursion in our protocol we will have a **loop** frame that, for this protocol, encapsulates all interactions. For the steps given with the arrow notation we simply send a message between the two participants from the given step. An example would be step 2 from the arrow notation in table 3.2. Here we would send a message with the name: *acknowledgement* or simply A from the ISP to the Client. Once done for the remaining steps we end up with a MSC for the global protocol. The MSC for the example given in section 3.4.1 can be seen in figure 3.5.

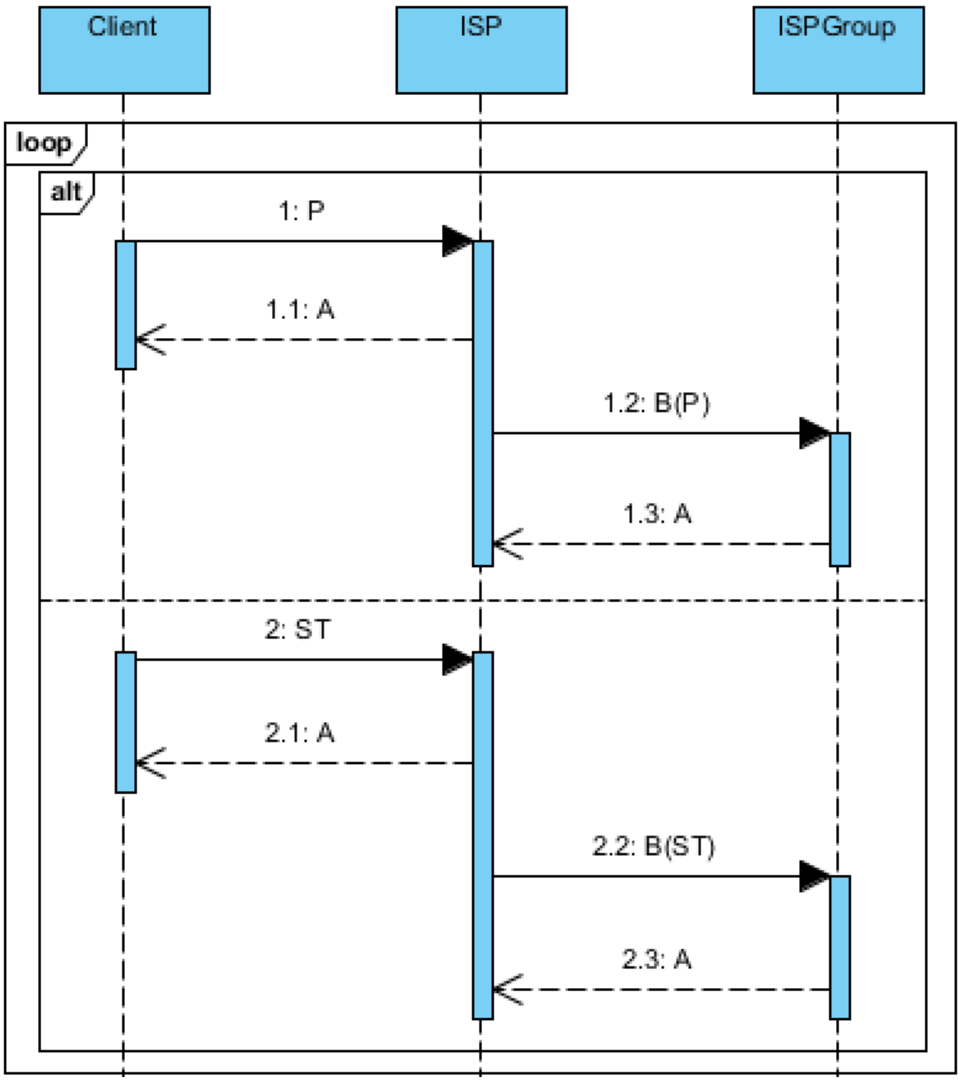


Figure 3.5: MSC for the global protocol example

When observing the arrow notation from tables 3.2 and 3.3, and the MSC in figure 3.5 it is clear that these two descriptions of the protocol are very similar. We will use Scribble to further formalize what we have presented thus far.

3.4.3 The Scribble language and tool

Writing protocols and ensuring that they are free of deadlocks can be a complicated affair. Thus we have chosen to use Scribble to help us design valid protocols since the features it provides will be useful for implementing our protocol. In this section we will go over how to describe multiparty protocols with the tool and language Scribble [41][42]. With this tool we are able to formally describe our global protocols from an endpoint neutral perspective using the Scribble language.

The Scribble language is an adaptation on Multiparty Session Types (MPST), which is a way to describe the communication between multiple participants. This is used for validation of the global protocols written in the Scribble language. MPST is based on π-calculus with session types which is then extended in order to handle multiparty sessions. A *session* is defined by a series of interactions between *participants* which becomes a *conversation*, thus a session can be used to describe protocols. Sessions are established among multiple *participants* by the use of a *shared name* which makes up a public interaction point.

Some of the useful features Scribble provides are *validation* and *projection*. Scribble is able to validate protocols ensuring that they are free from deadlocks as well as live-locks. After checking if the written protocols are valid we are able to use Scribble for projection, with this global protocols are projected in order to identify local protocols. Scribble uses the MPST algorithm for projection to ensure the dependencies of the global protocol. This means that the local types keep payload types like $string$ or int, and core interaction structures such as rec and $choice$ that involves the projected role. If the local protocol for each participant is correct then we know that the global protocol is correct as well. All following descriptions in this section are based on the work by Honda, Yoshida and Carbone [43].

With binary session types the type for one of the two endpoints is sufficient for describing the whole conversation, since you are able to construct the other type from the one you currently have. This is not the case for multiparty sessions, because of this a type abstraction that describes global conversation for multiparty sessions is needed. This type abstraction is global session types which is what Scribble implements. The syntax for global types can be seen in table 3.4.

Global G	::=	$p \to p' : k\langle U \rangle.G'$	values
	|	$p \to p' : k\{l_j : G_j\}_{j \in J}$	branching
	|	G, G'	parallel
	|	$\mu\mathbf{t}.G$	recursive
	|	$\mathbf{t}$	variable
	|	end	end
Value U	::=	$\tilde{S} \mid T@p$	
Sort S	::=	bool | nat | ... | $\langle G \rangle$	

Table 3.4: Syntax of global types from [43]

In the grammar from figure 3.4 global types are denoted $G, G', \ldots$, participants are denoted p, communication channels are denoted k. Lastly, for our work, we consider U

a single base type, which is the type of a message. Though they are normally a vector of sorts $S, S', \ldots$ consisting of types for shared names, or of those for session channels. The type $p \to p' : k\langle U \rangle.G'$ can be explained as participant p sending a message of type U to participant p' through channel k, following this the interactions described in G' takes place. The type $p \to p' : k\{l_j : G_j\}_{j \in J}$ is for branching and can be explained as p sending one of the labels l_j to channel k, which is then received by p'. When p' receives l_j interactions described in G_j take place. The type G, G' represents the concurrent execution of G and G'. Type $\mu\mathbf{t}.G$ is used for recursion allowing conversation structures to reoccur after each other. Type $\mathbf{t}$ is used to describe a variable.

We can use global session types to create a protocol showing how participants communicate on a global level. This means that the conversations between all the participants should be described. Global session types are used to give a complete view of all the participants. We give our previously used example using global session types in table 3.5

$$
\begin{array}{llll}
\mu t. \quad \text{Client} \to \text{ISP} \; : \; i\{\text{Panic} & : \text{ISP} \to \text{Client} & : & c\langle Acknowledgement \rangle. \\
 & \text{ISP} \to \text{ISPGroup} & : & ig\langle BroadcastPanic \rangle. \\
 & \text{ISPGroup} \to \text{ISP} & : & i'\langle Acknowledgement \rangle.t, \\
\text{StopThrottle} & : \text{ISP} \to \text{Client} & : & c\langle Acknowledgement \rangle. \\
 & \text{ISP} \to \text{ISPGroup} & : & ig\langle BroadcastStopThrottle \rangle. \\
 & \text{ISPGroup} \to \text{ISP} & : & i'\langle Acknowledgement \rangle.t\}
\end{array}
$$

Table 3.5: Protocol example in global types

Our global example uses the recursive syntax $\mu\mathbf{t}.G$ to repeat itself over and over which allows us to use branching to always accept either a *Panic* or a *StopThrottle* signal. Whenever the protocol has done the conversations for either of the signals it goes back to the start.

Being able to describe sessions from a global perspective we will now move on to describing the local participants using MPST. MPST makes use of base sets to describe multiparty sessions using processes. Those sets are:

- *shared names* or *names*, ranged over by $a, b, x, y, z, \ldots$
- *session channels* or *channels*, ranged over by $s, t, \ldots$
- *labels*, ranged over by $l, l', \ldots$
- *process variables* ranged over by $X, Y, \ldots$

Processes ranged over by $P, Q, \ldots$ and *expressions* ranged over by $e, e', \ldots$ are described through the syntax seen in table 3.6. In the syntax for hiding n is used either as a single shared name or a vector of session channels, while it also denotes a natural number in the syntax for multicast session requests. It is also worth mentioning that p in the syntax for session acceptance denotes participants.

P	$::=$	$\bar{a}[2..n](\tilde{s}).P$	multicast session request
	$\mid$	$a[p](\tilde{s}).P$	session acceptance
	$\mid$	$s!\langle\tilde{e}\rangle; P$	value sending
	$\mid$	$s?(\tilde{x}); P$	value reception
	$\mid$	$s!\langle\langle\tilde{s}\rangle\rangle; P$	session delegation
	$\mid$	$s?((\tilde{s})); P$	session reception
	$\mid$	$s \triangleleft l; P$	label selection
	$\mid$	$s \triangleright \{l_i : P_i\}_{i\in I}$	label branching
	$\mid$	if e then P else Q	conditional branch
	$\mid$	$P \mid Q$	parallel composition
	$\mid$	$\mathbf{0}$	inaction
	$\mid$	$(\nu n)P$	hiding
	$\mid$	def D in P	recursion
	$\mid$	$X\langle\tilde{e}\tilde{s}\rangle$	process call
	$\mid$	$s : \tilde{h}$	message queue
e	$::=$	$v \mid e$ and $e' \mid$ not $e \ldots$	expressions
v	$::=$	$a \mid$ true $\mid$ false	values
h	$::=$	$l \mid \tilde{v} \mid \tilde{s}$	messages-in-transit
D	$::=$	$\{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i\in I}$	declaration for recursion

Table 3.6: Syntax for multiparty sessions from [43]

Now that we have looked at the syntax for multiparty sessions we can give our previously used examples as processes. We will have a process for each participant in our example which means that we will have 3 in total, these 3 being for Client, ISP and ISPGroup. The different processes are shown in tables 3.7, 3.8 and 3.9

$$\begin{aligned}
\text{Client} \overset{\text{def}}{=}\ & \text{def}\ C(i,i',c,ig)\ =\ \text{if}(OngoingDDoS)\ \text{then}\ i \lhd \text{Panic};\\
& \quad c?(acknowledgement);\ C(i,i',c'ig)\\
& \text{else if}(DDoSStop)\\
& \quad i \lhd \text{StopThrottle};\ c?(acknowledgement);\ C(i,i',c,ig)\\
& \text{in}\ \bar{a}[2..3](i,i',c,ig).C\langle i,i',c,ig\rangle
\end{aligned}$$

Table 3.7: Local process for client

$$\begin{aligned}
\text{ISP} \overset{\text{def}}{=}\ & \text{def}\ I(i,i',c,ig)\ =\ i \rhd \{\text{Panic}:\ c!\langle Acknowledgement\rangle;\\
& ig!\langle BroadcastPanic\rangle;\ i?(acknowledgement);\ I(i,i',c,ig),\\
& \text{StopThrottle}:\ c!\langle Acknowledgement\rangle;\\
& ig!\langle BroadcastStopThrottle\rangle;\ i?(acknowledgement);\ I(i,i',c,ig)\}\\
& \text{in}\ a[2](i,i',c,ig).I\langle i,i',c,ig\rangle
\end{aligned}$$

Table 3.8: Local process for ISP

$$\begin{aligned}
\text{ISPGroup} \overset{\text{def}}{=}\ & \text{def}\ IG(i,i',c,ig)\ = ig?(broadcast);\ i!\langle Acknowledgement\rangle; IG(i,i',c,ig)\\
& \text{in}\ a[3](i,i',c,ig).IG(i,i',c'ig)
\end{aligned}$$

Table 3.9: Local process for ISPGroup

We have now covered how global types as well as local processes are described. We will now make a connection between these two methods for describing protocols. As we know the global types describe interaction on a global, or endpoint neutral, perspective. However if we wish to validate the global type we cannot simply compare it to the local processes, we will need to project the global type into types for each of the participants. When a global type is projected onto each of its participants they each get a local type. The local types can be used to compare with the local processes in order to validate them and thus act as a link between the global types and local processes. They can also be used as a starting point for the implementation of the local processes, since the local types also act as a template for the local processes.

Local session types, or local types, ranged over by $T, T', \ldots$, are used for describing local behavior of processes. The syntax for local types can be seen in table 3.10. The syntax for U and S are the same as the ones used for global types. In the syntax $T@p$ is a *located type*, which represents a local type T assigned to a participant p. Type $k!\langle U\rangle;T$ is used for sending a message of type U through channel k and type $k?\langle U\rangle;T$ is for receiving a message. Type $k\&\{l_i : T_i\}_{i\in I}$ is used for branching, if we need different outcomes depending on specific circumstances we can use this. Type $k \oplus \{l_i : T_i\}_{i\in I}$ is used for selecting some different options given by the branching type. Type $\mu\mathbf{t}.T$ and $\mathbf{t}$ is used the same way as recursion and variables from the global type syntax.

Value U	::=	$\tilde{S} \mid T@p$	
Sort S	::=	bool | ... | $\langle G\rangle$	
Local T	::=	$k!\langle U\rangle;T$	send
	|	$k?\langle U\rangle;T$	receive
	|	$k \oplus \{l_i : T_i\}_{i\in I}$	selection
	|	$k\&\{l_i : T_i\}_{i\in I}$	branching
	|	$\mu\mathbf{t}.T \mid \mathbf{t} \mid$ end	

Table 3.10: Syntax of local types from [43]

We can now move on to describe the example protocol in Scribble. In the example we will use the same participants as described with MPST in tables 3.7, 3.8 and 3.9. The Scribble implementation begins with a module declaration. The example we are about to give is self contained, however protocols in Scribble can be made up of multiple modules. Following this types are declared. These types can be imported from other languages. For example the *string* payload type is declared using the following *type <java> "java.lang.String" from "rt.jar" as String;*. Following the type declarations is the actual global protocol description which is similar to the global types described earlier. The global protocol for our example can be seen in listing 3.1. There is a clear similarity between the global protocol and the global type from table 3.5.

```
module Example;

type <java> "java.lang.String" from "rt.jar" as String;

global protocol Example(role Client, role ISP, role
    ISPGroup) {
        rec START {
        choice at Client {
                P(String) from Client to ISP;
                A() from ISP to Client;
                BP(String) from ISP to ISPGroup;
                A() from ISPGroup to ISP;
                continue START;
        } or {
                ST(String) from Client to ISP;
                A() from ISP to Client;
                BST(String) from ISP to ISPGroup;
                A() from ISPGroup to ISP;
                continue START;
        }
    }
}
```

Listing 3.1: The example global protocol implemented in Scribble

When the global protocol from listing 3.1 is projected, we get 3 local protocols, one for each participant. The local protocol for the Client participant can be seen in listing 3.2, it is clear that it is very similar to the Client process from table 3.7. This is not surprising since the local protocols that Scribble outputs (local types), are abstract descriptions of the processes.

```
module Example_Client;

type <java> "java.lang.String" from "rt.jar" as String;

local protocol Example at Client(role Client,role ISP,role
    ISPGroup) {
        rec START {
                choice at Client {
                        P(String) to ISP;
                        A() from ISP;
                        continue START;
                } or {
                        ST(String) to ISP;
                        A() from ISP;
                        continue START;
                }
        }
}
```

Listing 3.2: The example global protocol implemented in Scribble

3.5 Protocol design

In this section we will describe the design of our reaction protocol. In subsection 3.5.1 we will go over the environment in which our protocol will operate. In subsection 3.5.2 we will give the actual design for our reaction protocol described using the 3 different notations presented in section 3.4. In subsection 3.5.3 we will describe the information that is being sent between the participants of the reaction protocol.

3.5.1 Environment and assumptions

To provide a description of our protocol design, we will start with describing the network environment and the assumptions we make about it. Figure 3.6 shows a possible network environment in which our reaction protocol could operate. This environment is based on the description of networks and how ISPs are structured from chapter 1 in [9].

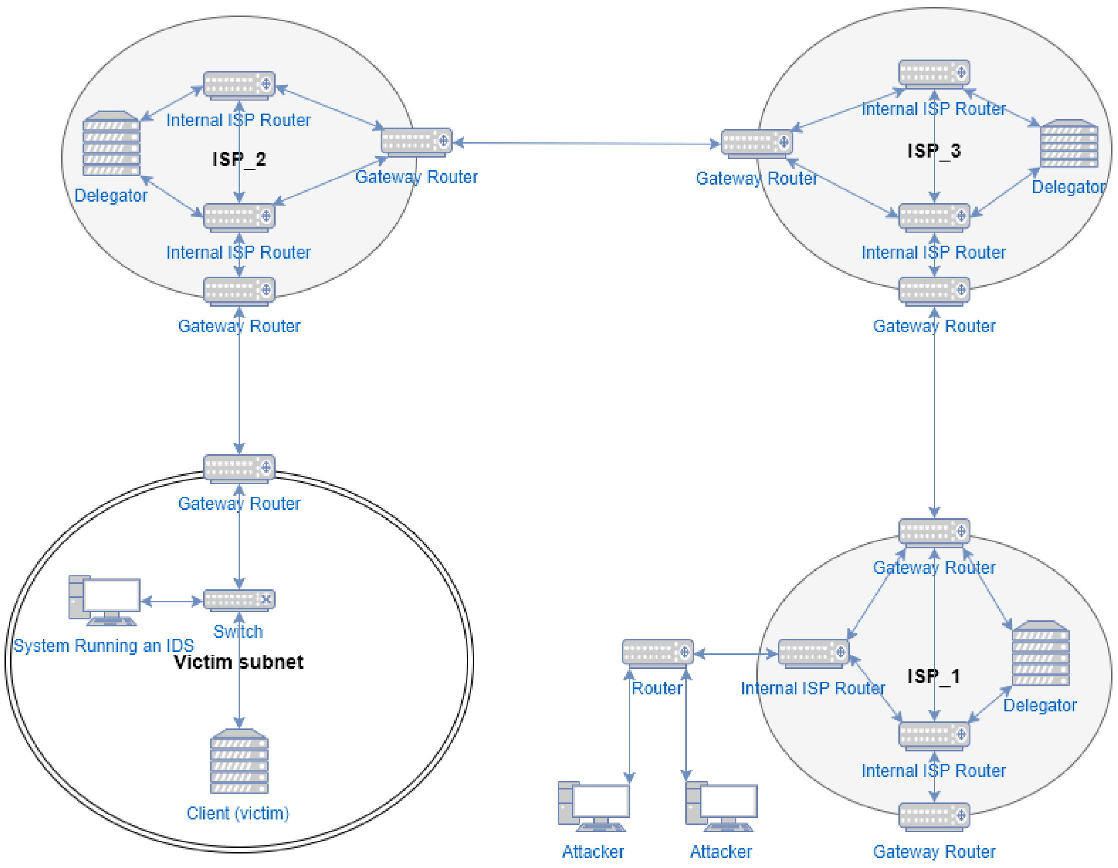


Figure 3.6: An example of a possible network setup with multiple ISP's involved. Each ISP is connected through a gateway router to another ISP. The victim subnet contains a switch that outputs the incoming traffic to the victim interface as well as the IDS interface, with the purpose of analyzing the traffic and generating/sending alerts to the *closest* delegator. In this example the closest delegator would be the one from ISP_2.

The representation of the environment given in figure 3.6 is not the exact structure in which our protocol is going to operate, but we do however assume that a client (victim subnet) and an ISP are structured in the way they are presented in the figure when we design our reaction protocol.

We have also made the following assumptions about the environment in which our reaction protocol is going to operate:

Assumption 1
: All ISPs that have implemented our protocol can communicate with each other, if there is a connection, either direct or indirect, between them.

Assumption 2
: A victim of a DDoS attack can always send out a signal to its ISP to initiate our protocol.

Assumption 1 and 2 were made as we need guaranteed communication between the participants to ensure that our reaction protocol is viable. We will now discuss if these assumptions can be considered fair.

As for assumption 1. We consider this a fair assumption to make since the ISPs will communicate through the use of the protocol we are going to implement. Let us consider the environment from figure 3.6. If we assume that ISP_2 and ISP_1 both implement the protocol but ISP_3 does not we would still be able to forward a message from ISP_2 to ISP_1 through ISP_3 and thus they can communicate. This does not hold if ISP_3 goes offline, but this is beyond what we can control.

As for assumption 2. If the victim is unable to signal its ISP, it can mean one of two scenarios. One scenario involves the victim crashing before the signal can be send. The other involves losing the connection to the ISP and by extension to the internet. In either case, the problem is beyond the aid of a protocol. Also, due to the fact that no protocol can work if it is impossible to communicate between two parties, we think the assumption of being able to signal is fair.

3.5.2 Protocol description

We can now begin describing the reaction protocol. We will begin by giving a description of the protocol using text. After the text description we use the three notations presented in section 3.4. We first describe the global protocols and later we use Scribble to obtain the local protocols from the global.

In the description of our reaction protocol we make use of sets of participants. For the sake of clarity we list the sets of participants the protocol includes here:

- *Delegator*$_1 \cdots$ *Delegator*$_j$ where $j \geq 1$
- *Router*$_1 \cdots$ *Router*$_k$ where $k \geq 1$

The *client* is not a member of these sets as there is only one client in the protocol. The *client* will begin by making a choice between which of the two signals it will send to the *delegator* of its *ISP* based on whether the client is the target of a DDoS-attack or not.

Under attack

If a client registers a DDoS attack, it will send a panic signal, $\mathcal{P}$, to the delegator of its ISP. This delegator will then send back an acknowledgement, A, and begin to broadcast the panic signal, $B(\mathcal{P})$, to the *j* delegators that implement the reaction protocol. Each of these delegators will send back an acknowledgement to the broadcasting delegator. Each delegator will then send a signal to throttle traffic towards the client, TT, to its *routers* of which there are *k* in total. If the client is still under attack, it will send the panic signal again.

Not under attack

If the client does not register a DDoS attack it can send a signal to stop throttling, ST, to the delegator. The delegator will acknowledge this and broadcast the signal, $B(ST)$ to the *j* delegators. The delegators will then send back an acknowledgements and all delegators send a signal, STT, to the *k* routers under each delegators control to stop the throttling of traffic towards the client.

From the above description of the reaction protocols it becomes clear that we need to introduce more messages M. The new messages we are adding are the following:

$TT-$**Throttle Traffic**: Signal sent by a delegator to its routers to throttle traffic.

$STT-$**Stop TT**: Signal sent by a delegator to its routers to stop traffic throttling.

Since these are the last messages we will introduce we will give the complete syntax for messages M used in the arrow notation in table 3.11.

M	$::= \mathcal{P} \mid ST \mid B \mid A \mid TT \mid STT$

Table 3.11: Syntax for messages M used in the arrow notation

We can now use the arrow notation to describe the reaction protocol. The arrow notation for the **Under attack** sub-protocol can be seen in table 3.12 and the **Not under attack** sub-protocol in table 3.13.

1.	$Client \rightarrow Delegator$	$\mathcal{P}$
2.	$Delegator \rightarrow Client$	A
3.	$Delegator \rightarrow Delegator_{2..j}$	$B(\mathcal{P})$
4.	$Delegator_{2..j} \rightarrow Delegator$	A
5.	$Delegator_{1..j} \rightarrow Router_{1..k}$	TT
6.	$Router_{1..k} \rightarrow Delegator_{1..j}$	A

Table 3.12: Arrow notation for the **Under attack** sub-protocol

1.	$Client \rightarrow Delegator$	ST
2.	$Delegator \rightarrow Client$	A
3.	$Delegator \rightarrow Delegator_{2..j}$	$B(ST)$
4.	$Delegator_{2..j} \rightarrow Delegator$	A
5.	$Delegator_{1..j} \rightarrow Router_{1..k}$	STT
6.	$Router_{1..k} \rightarrow Delegator_{1..j}$	A

Table 3.13: Arrow notation for the **Not under attack** sub-protocol

It is worth mentioning that in the arrow notation we use Delegator$_{2..j}$ for the steps in the notation where a delegator broadcasts to other delegators. This is because we do not want the broadcasting delegator to notify and send acknowledgements from itself, to itself.

We can now move on to describing our reaction protocol with a MSC which is given in figure 3.7. In the given MSC we deviate a bit from the method given in section 3.4.2. This is because we wish to reflect that there are multiple delegators and switches that take part in the reaction protocol whenever a client sends out a signal.

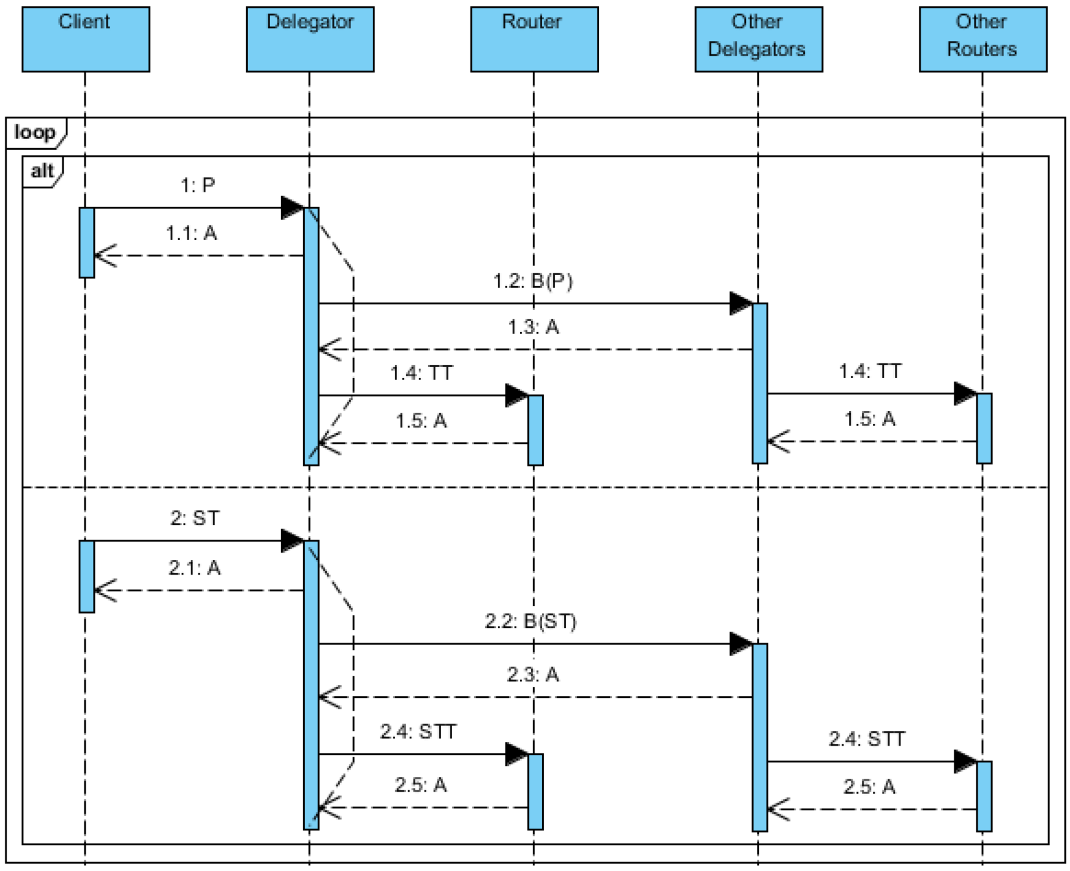


Figure 3.7: Message sequence chart for our reaction protocol.

The MSC in figure 3.7 clearly reflects the fact that delegators have different behavior dependent on if they receive a panic signal $\mathcal{P}$ from a client or a broadcast $B(\mathcal{P})$ from the broadcasting delegator. The same behavior is displayed with the stop throttle signal ST and broadcast $B(ST)$.

Now we have described our protocol using arrow notation and message sequence chart it is time for the last notation. However, since the number of participants in our protocol is unknown, we are not able to use Scribble alone to describe it since Scribble only describes a specific number of participants. Therefore we will use an extension of Scribble called Pabble, developed by Nicholas Ng and Nobuko Yoshida [44]. Pabble is an extension of Scribble including parameterisation, allowing us to describe the number of participants using indices. We need to use Pabble for our protocol because we do not know the number of delegators or routers using our protocol and the number can always change. Our global protocol can be seen in listing 3.3. Note that in the global protocol we have delegators ranging from 1 to N and routers ranging from 1 to H. When looking at the global protocol N describes all the delegators and H describes all the routers using the protocol. However, when the protocol is projected into the local protocols each delegator has a subset of routers that belong to them. Therefore each subset of routers will be notified by their own delegator.

```
module Reaction;

global protocol Reaction(role Client, role Delegator[1..N], role Router[1..H]){
    rec START{
        oneof (Delegator[j in 1..N]){
            choice at Client {
                PanicSignal(string) from Client to Delegator[j];
                SignalReceived() from Delegator[j] to Client;
                foreach (x:1..N except j){
                    PanicSignal(string) from Delegator[j] to Delegator[x];
                    SignalReceived() from Delegator[x] to Delegator[j];
                }
                foreach (y:1..H){
                    ThrottleTrafic() from Delegator[j] to Router[y];
                    SignalReceived() from Router[y] to Delegator[j];
                }
                continue START;
            } or {
                SeizeThrottling(string) from Client to Delegator[j];
                SignalReceived() from Delegator[j] to Client;
                foreach (x:1..N except j){
                    SeizeThrottling(string) from Delegator[j] to Delegator[x];
                    SignalReceived() from Delegator[x] to Delegator[j];
                }
                foreach (y:1..H){
                    StopThrottleTraffic() from Delegator[j] to Router[y];
                    SignalReceived() from Router[y] to Delegator[j];
                }
                continue START;
            }
        }
    }
}
```

Listing 3.3: Global reaction protocol described with Pabble

Looking at the roles in the protocol we see that the client role only consists of one participant. The reason for this is that the protocol is initiated by a single client which means that in the given protocol the delegators only handle one client per protocol. This does however not mean that multiple clients cannot implement the protocol, only that distinct clients are not part of the same protocol instance.

The local protocols gained from projecting each participant in the global protocol into their respective local protocol are shown in appendix A.

3.5.3 Information sent between local protocols

In our message sequence chart in figure 3.7 and in the Pabble protocol shown in listing 3.3 we alluded to which parameters the different signals include. This subsection will contain an overview of the signals, parameters and the rationale of why they are included.

The following are the signals from our protocol.

- $Client : PanicSignal(expires, ipAddress, signalTimestamp)$
- $Delegator : PanicSignal(expires, ipAddress, signalTimestamp, panicIncident)$
- $SeizeThrottle(ipAddress, signalTimestamp)$
- $StopThrottleTraffic()$
- $SignalReceived()$

The signals includes up to four different parameters. The following description contains arguments as to why we included these parameters:

Expires This field can be specified by the client. It determines for how long throttling will remain active, without further interference from the client.

Panic incident A client might want to continuously send panic signals, to make sure that throttling stays active and that a panic signals does not expire. If we had no panic incident number, a delegator receiving a second signal from the same client would reject the second signal, since it would already be in its table, which can be avoided with this field.

Client (victim) IP address The first delegator reached does not require that the application layer message contains the IP address of the victim, since this could be extracted from the IP datagram header. Subsequent messages however, would require this information, since these originate from the delegator and not the client. Including the information in the first packet sent, or from the first delegator and onwards would essentially have the same effect.

Timestamp We cannot guarantee that messages arrive in the same order that they were sent, and therefore we use a timestamp. For example, if a $SeizeThrottle()$ signal arrives *before* the $PanicSignal()$, then the delegator that receives the $PanicSignal()$ would actually notify the routers that they should throttle network traffic, while it should actually not be throttling traffic. By having delegators keep track of timestamps, which are generated by clients, we can eliminate the need for having signals arrive in order.

Each delegator would have to keep track of this information. Table 3.14 contains example entries of transmitted panic signals, which a delegator would keep in main memory. Another table is used for keeping track of signals, which are meant to seize throttling. Table 3.15 contains example entries of such a table.

	Panic incident	Expires (seconds)	Client(victim) IP address	Timestamp
Example entry #1	0	600	10.0.2.15	12:30:17 CET
Example entry #2	1	600	10.0.2.15	12:30:42 CET
Example entry #3	0	300	102.204.56.1	13:37:00 CET
Example entry #4	2	600	10.0.2.15	12:31:48 CET

Table 3.14: An example of a table that contains all the necessary fields to keep track of information regarding panic signals

	Panic Incident	Client (victim) IP address	Timestamp
Example entry #1	0	10.0.2.15	12:32:27 CET
Example entry #2	0	102.204.56.1	13:36:30 CET
Example entry #3	1	102.204.56.1	13:38:52 CET

Table 3.15: An example of a table that includes all the necessary fields of information for keeping track of seize throttling signals.

Chapter 4

Implementation

In chapter 3 we designed an application layer protocol[45]. Figure 4.1 shows how the internet, which is the designated network for our protocol, is layered. At the application layer it is possible to use transport layer protocols such as TCP and UPD, to send messages between hosts. In our implementation we use TCP to send messages, since it provides reliable data transfer, which eliminates the concern of a panic signal not reaching a delegator.

To implement and test our protocol we would have to set up a realistic environment, which matches that of a real physical network. To achieve this we could set up a physical network, however, this would be a cumbersome task requiring a lot of hardware. The first section of this chapter, section 4.1, provides an overview of an ideal system, in which our protocol would operate. In section 4.2 we describe how we can emulate a real network, with physical switches and hosts, in a virtual environment. In section 4.3 we will go over the principles of software defined networking (SDN) and the influence it has on our project. Section 4.4 is about queuing and throttling and how we can implement throttling in a virtual network. This same method is also applicable to real networks. We would like to be able to manually configure how much throughput there is in the network, with a bias against certain packets. In section 4.5 we describe how we can practically achieve this goal and that this same method is applicable to physical networks and not just virtual ones. In section 4.6 we will describe how we monitor traffic through an IDS in order to detect DDoS attacks. At the end of this chapter, in section 4.7, we perform some proof-of-concepts.

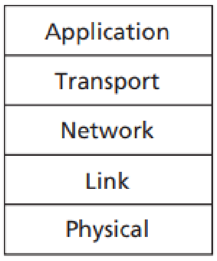


Figure 4.1: Five layer protocol stack, which represents how the internet is structured [9]. The physical layer is the first layer. Link layer the second etc.

4.1 System structure overview

In order to create an overview of how all the components in our solution are put together, we have made a complete overview as seen in figure 4.2.

This overview shows a simple Mininet network [45], with four hosts connected to two ISPs. Each of the two ISPs have a delegator and two switches. The victim is connected to ISP A. In this scenario, together with the victim there is a *Linker*, which is also a host, but in a real scenario could be another process running on other hosts or the victim itself. The two attackers are connected to ISP B, which are the hosts conducting the DDoS attack. Delegator A and B are virtual hosts in Mininet communicating with the Pox controller [46]. In our setup Pox is running externally to Mininet (hence the dotted lines between the controller and the delegators). The Linker sends a signal to the Delegators and in turn the Delegators request initiation, and the ceasing, of throttling through the pox controller. The controller in turn sends messages with the OpenFlow protocol to the Open vSwitches, which implement a part of the OpenFlow protocol.

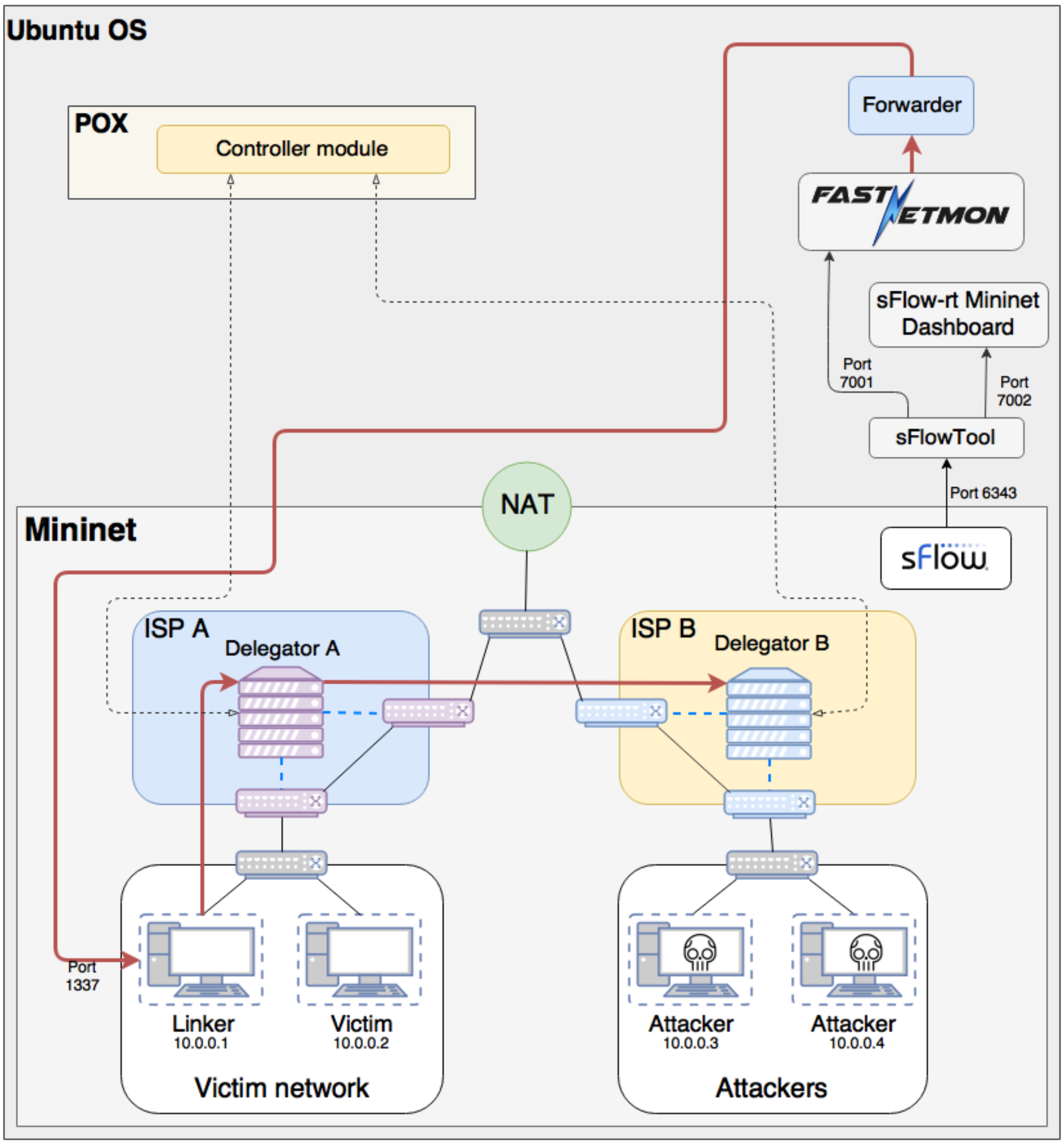


Figure 4.2: A complete overview of how our implementation is set up.

The red arrows, of figure 4.2, navigate through the setup as if they are the signal being transmitted, when a DDoS attack is detected. When an attack is detected at the victim, by the IDS FastNetMon, the *Forwarder* is launched, sending the attack data to the *Linker*, where a *panic signal* is constructed and transmitted. From the Linker it is sent to *Delegator A*. Delegator A receives the signal data and establishes throttling according to the data. Delegator A also propagates the panic signal to all other ISPs, in this case only *Delegator B* in ISP B. Delegator B enacts a similar throttling based on the same panic signal. DDoS traffic from the attackers is now being throttled, at all four ISP switches, where the attacking traffic flows through. This results in less traffic at the victim, which would mitigate the attack.

The idea with the setup from figure 4.2 is that it would be very close to a realistic network, in which our protocol would operate. However, during the implementation phase we discovered that, due to the amount of different components that would have to interact with the Mininet platform, the setup became quite complex and using this setup for testing our protocol proved to be a challenge. We spent much time configuring components rather than actually spending time testing the protocol. Therefore, in section 4.7, we stray away from this setup and perform some tests on a smaller scale with fewer components. In the sections that come before section 4.7 we will continue to elaborate on the components of the design presented in this section.

4.2 Mininet - a network emulator

Mininet can be used to emulate a physical network through process based virtualization. When a host, switch, controller or link in Mininet is instantiated, it becomes a process in itself. This makes it possible to have each host execute different programs, and communicate, via the switches and links in the virtual network.

Mininet can generally be used in two ways:

- Through Mininets command-line interface (CLI)
- Through the Python API

Listing 4.1 is an example of using the Python API of mininet to instantiate a virtual network with a predefined topology. Mininet currently only supports Python 2.7.

```python
from mininet.net import Mininet
from mininet.topo import LinearTopo
from threading import Thread
import os, time
linear = LinearTopo(k=2)
net = Mininet(topo=linear)
net.start()
def Server():
        h1 = net.get("h1")
        result = h1.cmd("sudo python
           /etc/pythonprograms/pythonServer.py")
        print(result)
def Client():
        h2 = net.get("h2")
        h2.cmd("sudo python
           /etc/pythonprograms/pythonClient.py")
```

```
try:
        thread1 = Thread(target=Server)
        thread2 = Thread(target=Client)
        print("starting thread 1")
        thread1.start()
        time.sleep(3)
        print("starting thread 2")
        thread2.start()
except Exception as e:
        print("Could not start threads: %s"%(e))
net.interact()
net.stop()
os.system("sudo mn -c") #removes instances created by
    mininet
```

Listing 4.1: Example of two hosts in mininet, each running a different Python program. One acts as the client and the other as the server.

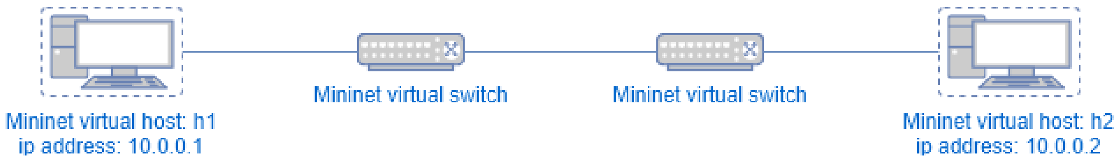


Figure 4.3: The topology that was instantiated in listing 4.1. In Mininet hosts are automatically assigned names and IP addresses, although it is possible to manually assign them, as well as MAC addresses. When Mininet automatically assigns host names, it follows the sequence: "$h1$","$h2$" … "hn", where $n \geq 1$. IP addresses are similarly incremented from the right side. "10.0.0.1","10.0.0.2" … etc. Mininet hosts have a 32-bit IP address space.

In listing 4.1 we use the *Mininet* class to create a virtual network. In this case we gave it a parameter, *linear*, which is an already defined topology in Mininet. This topology includes two hosts, two switches and three links. Figure 4.3 is a visual representation of the topology that was instantiated in listing 4.1. A reference to the two host objects is obtained by using the *net* object and subsequently calling the hosts by their assigned names. The two hosts are given a command through the *cmd()* function. Hosts in Mininet have access to the entire file system, and can therefore run any program that can be run on the Ubuntu machine. The program that each host runs, *PythonClient* and *PythonServer*, can be found in listing 4.2 and listing 4.3.

```
import socket
serveripaddress = "10.0.0.1"
serverPort = 5555
```

```python
msg = "Hello"
msg = msg.encode("ascii")
clientSocket = socket.socket(socket.AF_INET,
    socket.SOCK_DGRAM)
clientSocket.bind(('', 5556))

clientSocket.sendto(msg, (serveripaddress, serverPort))
```

Listing 4.2: A client encoding the message "Hello" as ASCII characters and sending it through its socket, to server, which has the IP address "10.0.0.1" and is listening on port 5555

```python
import socket
serverSocket = socket.socket(socket.AF_INET,
    socket.SOCK_DGRAM)
serverSocket.bind(('', 5555))
print("server running")

msg = serverSocket.recv(1024)
msg = msg.decode("ascii")
print(msg)
```

Listing 4.3: A server listening on port 5555 for messages and afterwards prints the received message to the terminal. The reason that the server is not running *serverSocket.recv()* in a while loop is because the standard function in Python is blocking, which means the process will not continue executing until it has received some data in its socket.

These two programs are very simple, however they show that Mininet does what it is supposed to do, which is to emulate how a real network would behave. These two programs might as well have been executed on physical hosts, which would produce the exact same result, perhaps except for a greater likelihood of packet loss and a greater *round trip time* (RTT).

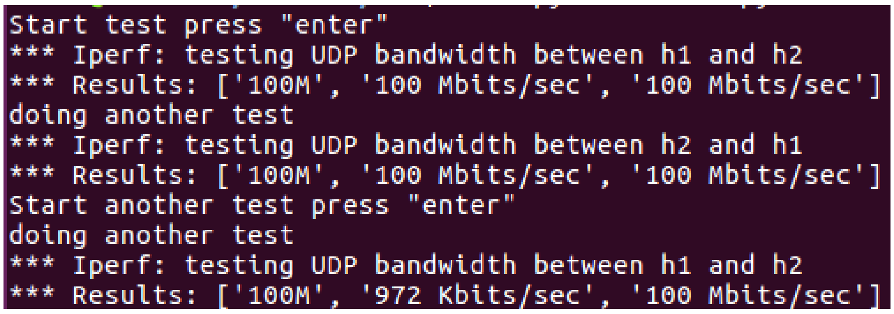
4.2.1 Creating custom topologies in Mininet

In the previous example we used a higher level API of Mininet. In listing 4.4 we add hosts, switches, links and a controller, which is considered part of Mininet's mid-level API.

```python
from mininet.net import Mininet
from mininet.node import OVSSwitch, RemoteController, Host
from mininet.cli import CLI
import os, time

net = Mininet(switch = OVSSwitch)
try:
        poxcontroller = net.addController(name="pox",
           controller=RemoteController,
                                ip="127.0.0.1",
                                   protocol="tcp",
                                   port=6633)
        h1 = net.addHost("h1")
        h2 = net.addHost("h2")
        s1 = net.addSwitch("s1")

        net.addLink(h1, s1) #s1_eth1
        net.addLink(s1, h2) #s1_eth2
        net.start()
        #wait until all switches have connected to the
           controller
        net.waitConnected()

        #bandwidth testing, 100M == 100megabit. Connection:
           host1 to host2
        listOfHosts = [h1, h2]
        net.iperf(listOfHosts, l4Type="UDP", udpBw="100M",
           seconds=5)
        #Testing the opposite connection: host2 to host1
        listOfHosts.reverse()
        net.iperf(listOfHosts, l4Type="UDP", udpBw="100M",
           seconds=5)

        net.stop()
        os.system("sudo mn -c") #build-in mininet command
           that cleans up after stopping the net
except Exception, e:
        print("Caught exception %s", e)
        os.system("sudo mn -c")
```

Listing 4.4: Manually adding components to the virtual network *net*.

We use the Python API, rather than the CLI, since our system should be able to run without user interaction. We use the function *iperf()* to test bandwidth. The function initiates a client and a server, and returns how much data the client sends and how much the server received, as can be seen in figure 4.4. This utility is therefore a good candidate for testing how much throttling occurs between a client and a server.



```
Start test press "enter"
*** Iperf: testing UDP bandwidth between h1 and h2
*** Results: ['100M', '100 Mbits/sec', '100 Mbits/sec']
doing another test
*** Iperf: testing UDP bandwidth between h2 and h1
*** Results: ['100M', '100 Mbits/sec', '100 Mbits/sec']
Start another test press "enter"
doing another test
*** Iperf: testing UDP bandwidth between h1 and h2
*** Results: ['100M', '972 Kbits/sec', '100 Mbits/sec']
```

Figure 4.4: Result of running the three iperf tests from listing 4.4. The third test shows that the server only received 972kbit/sec, while the client tried to send the usual 100Mbit/sec. We will explain in section 4.4 and section 4.5 how we can achieve this.

4.2.2 An ISP as the basis for a topology

As listing 4.4 shows we can use Mininet to create any number of hosts, switches and links. In subsection 3.5.1 we discussed the structure of an ISP and the environment of our protocol. Therefore, we would like to create topologies in Mininet that resemble the structure, which we described in subsection 3.5.1. Figure 4.5 shows the class diagram for the python class that has the responsibility of creating these ISP topologies.

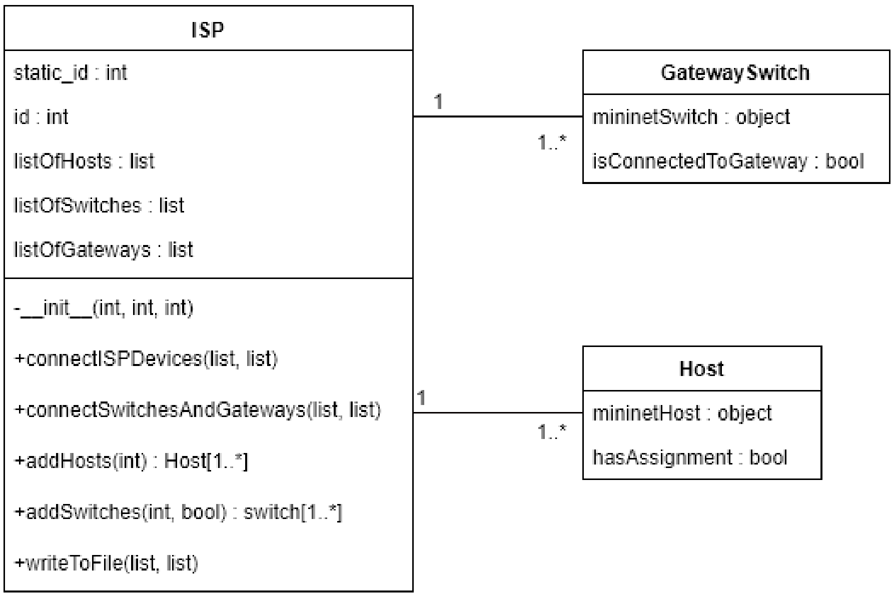


Figure 4.5: The ISP class is responsible for linking hosts with switches and switches with gateways. To do this, the ISP class uses methods and attributes from Mininet.

We use the class from figure 4.5 to instantiate topologies in Mininet, such as the topology in figure 4.6. One of the advantages of using this abstraction is to easily avoid creating topologies with cycles, which normal switches cannot handle. The way we can visualize instantiated topologies is through a tool called Gephi [47].

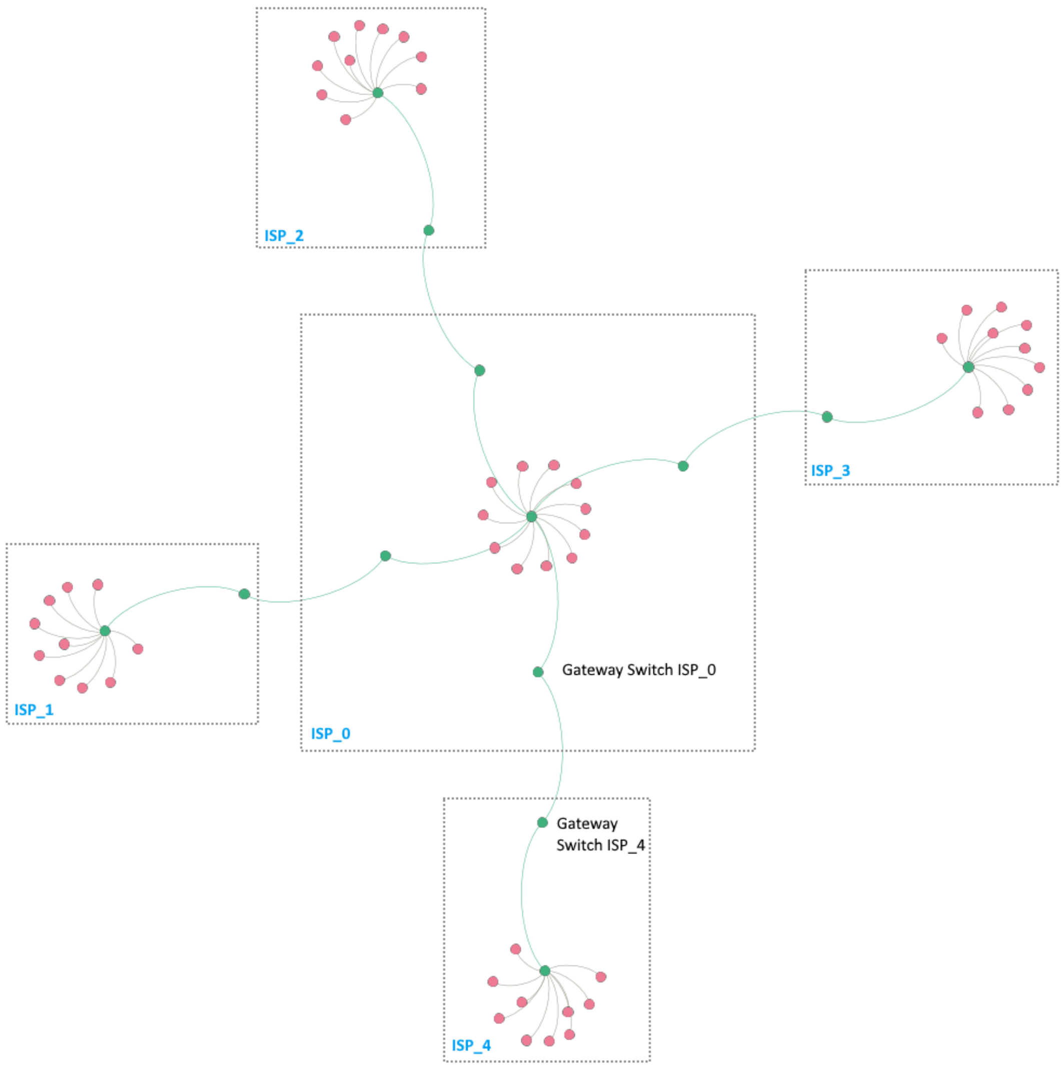


Figure 4.6: Example of a topology, which consists of 5 instances of the ISP class. The red vertices are hosts, and the green ones are switches. In our model of an ISP gateway switches do not have any edges to hosts, but only other switches.

4.3 An architectural view of software defined networking

The idea in our solution, as described in chapter 3, is to have a network dynamically adapt to the needs of an individual client. To do this, it would require more than Mininet, since Mininet only provides the infrastructure of a network and no other functionality.

It also requires us to be able to influence the behavior of a router or switch, and we will see how that is possible with software defined networking.

Figure 4.7 shows a general architecture of SDN. In our implementation the infrastructure layer is provided by Mininet, the control layer and the accompanying API is provided by Pox[46], and the application layer will consist of our implemented protocol. The Pox controller will be discussed in section 4.5.

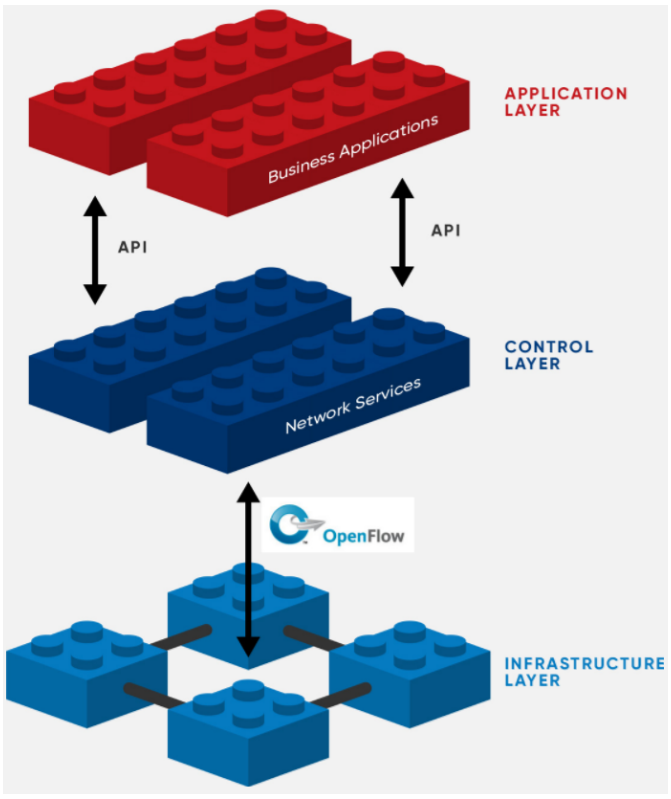


Figure 4.7: Architecture in software defined networking [48].

To be able to understand software defined networking we discuss what functionality SDN is supposed to replace. To do so we first discuss regular routers and switches.

Routers

Routers are L3 forwarding devices, where L3 refers to the third layer of the internet protocol stack, as seen on figure 4.1. Routers consist of a control plane(software) and a data plane(hardware), and perform forwarding based on the internet protocol. A router decides how to perform forwarding based on their forwarding tables, which use *longest prefix matching* and match against the header of a IP layer datagram.

The table contains entries consisting of an IP address and a port. If a packet matches against some entry, the router can perform a look-up, which returns which port the router should output the matched packet. The way a forwarding table is populated is by using a routing algorithm, such as the global Link-State algorithm or the decentralized Distance-Vector algorithm [9].

Switches

Switches are L2 devices that perform a similar task to routers, although their forwarding capabilities have nothing to do with the network layer, since they only interact with the link layer. At the link layer there is no notion of the internet protocol nor IP addresses. Therefore forwarding is based on MAC addresses. Similarly to routers, switches have to decide where to output a link layer frame. The switch has a switch table, similar to a routers forwarding table, but instead of IP addresses it contains MAC addresses, as well as which interface to output a matched frame. A populated switch table is acquired through a switches *self-learning* property, as described in [9].

Open vSwitch

Open vSwitch (OVS) is an OpenFlow switch implemented in software. An OpenFlow switch, like OVS, also has a control plane and a data plane, which figure 4.8 shows. OpenFlow switches are quite different from typical hardware routers and switches described previously. That is because the control plane of an OpenFlow switch can be configured through the OpenFlow protocol. Figure 4.9 shows the typical components of an OpenFlow switch.

When an OpenFlow switch processes packet input it goes through each flow table of its pipeline, as depicted on figure 4.9. If it finds a match, it will perform an *action*. If it gets a table miss, it can send a message, which conforms to the OpenFlow specification, to the controller. The controller can then send a message back to the switch, and *install* a flow in the flow table, which handles the packet that triggered the table miss to begin with. This way a Flow table is populated by contacting the controller through the OpenFlow protocol, which allows the controller to install flows based on any packet header information, such as the header information from an IP datagram or the MAC address from the header of a link-layer frame.

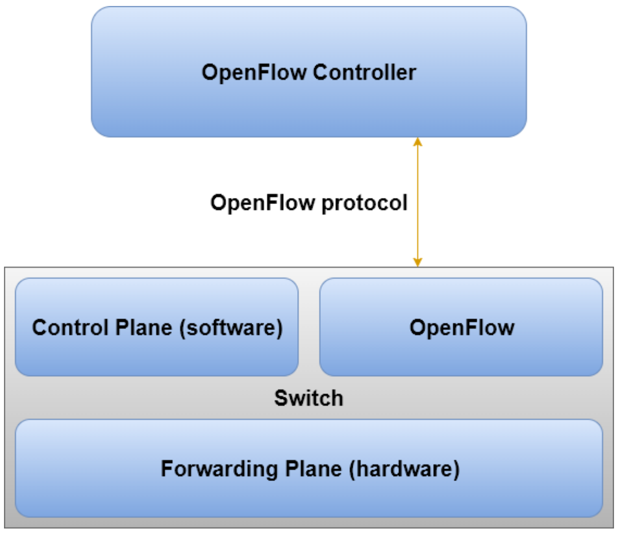


Figure 4.8: Example of an OpenFlow switch called Open vSwitch(OVS Switch) and how it is setup to communicate with an external controller [49].

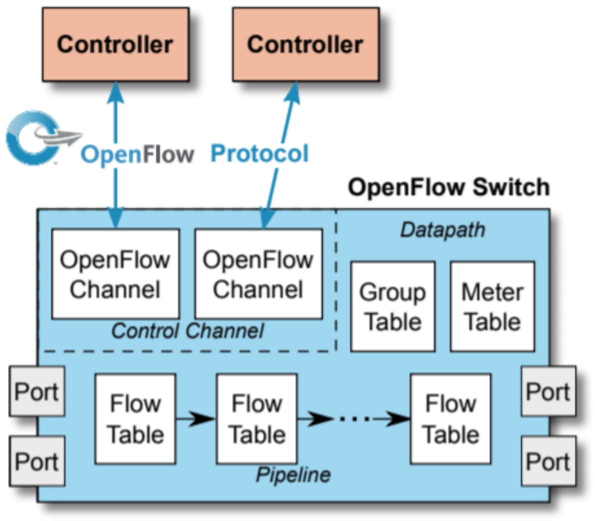


Figure 4.9: OpenFlow switches have *in* ports and *out* ports like typical switches. Instead of a MAC table they have flow tables, which perform a similar role, but can contain more information than a MAC address and corresponding interface. [50].

Flow entries of an OpenFlow switch

An OpenFlow flow table contains entries in the form of flows. Figure 4.10 shows a general flow table and the structure of the individual flows. Each individual flow can have fields to match against. These are listed under rule, as depicted on figure 4.11. A typical action results from *matching* a packet against a rule, which then has one or more associated actions. Usually a packet is forwarded to some port, however it is possible to specify multiple actions, such as forwarding a packet to both, for example, port 5 and port 6.

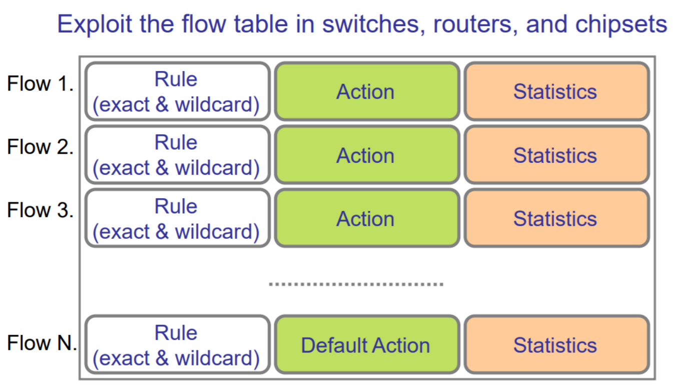


Figure 4.10: An OpenFlow flow table. Each flow has a rule, action and associated statistics [49]

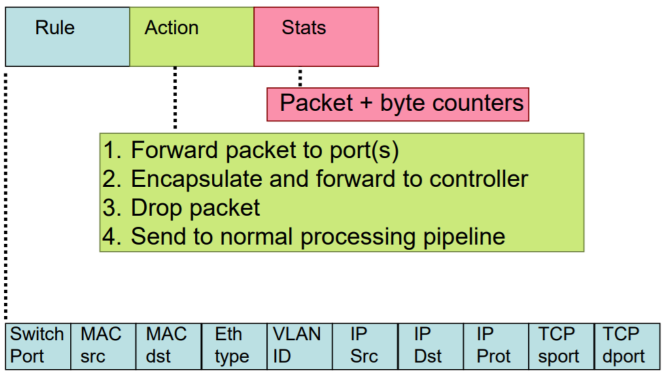


Figure 4.11: All the information that describes a single flow entry [49]

A simple match could consist of a single field, as seen on figure 4.12. If a packet matches the rule, by having the given MAC address, it will be forwarded to port 6, as indicated by the *action*.

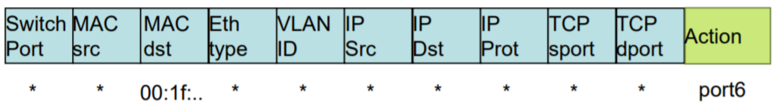


Figure 4.12: Example of a single flow. [49]

4.4 Queuing and throttling

In order to achieve throttling, we are using dedicated queues for traffic we wish to throttle. Our Pox component, when it receives a panic signal from a delegator, sends an OpenFlow message to its switches. These switches redirects traffic going to the victim into a queue that allows less throughput. This means that we, for each switch interface, have a queue for attack related traffic, while unrelated traffic will pass through unaffected. We can then set OpenFlow rules to throttle the bandwidth of this queue to some artificial value. As the degree to which we need to throttle this queue changes, we change the value - resulting in a dynamic throttling behavior.

When we initialize a Mininet network we can immediately create queues for each switch through a command. This command directly manipulates the underlying database of an OVS switch [51]. Since we typically have many switches in a Mininet network, we use a function to install the queues. Listing 4.5 shows the code for that function.

```python
def InitializeThrottleQueue(switchInterface, minBitsPerSecond=0,
    maxBitsPerSecond=1000000, queueSize=1000, queue_id=0):

    os.system("sudo ovs-vsctl -- set Port {0} qos=@newqos -- "
        "--id=@newqos create QoS type=linux-htb other-config:maxrate={2} queues={3}=@q{3} -- "
        "--id=@q{3} create Queue other-config:min-rate={1} other-config:max-rate={2}"
        .format(switchInterface, minBitsPerSecond, maxBitsPerSecond, queue_id))
```

Listing 4.5: A very small function that sends a command to the ovs-vsctl tool. The most interesting parameter in our case is the maxBitsPerSecond parameter, which specifies how much throughput the queue is limited to.

4.5 The Pox controller

Each delegator has to be able to influence switches in the network. This is done through a controller and in this case we use the Pox controller. Pox is an open source controller written in python [46] and supports the OpenFlow protocol by providing an API, thereby fulfilling the role of the control layer as indicated by figure 4.7.

If we were to instantiate a Mininet network with OVS switches, and subsequently run a *pingall* test, then no host would be able to reach any other host. This is because the OVS switches would not know what to do, since they have no built in functionality. This would be similar to a router having no control plane routing algorithm. Therefore, if an OpenFlow switch is to provide any forwarding functionality, then a controller is required. This makes OpenFlow controllers very modular, since we through code can decide how to perform, for example, IP forwarding by matching packets by their IP header or simply have OpenFlow switches act like regular hardware switches.

Our protocol requires that there is some throttling to be done, but with regards to everything else, we want normal network behavior. This means that we want switches to forward packets to the correct hosts. Pox provides some components to do this, which would make an OpenFlow switch act like, for example, a normal learning switch. This Pox component is a python file that is given to Pox as an argument. We use a Pox component called *l2_multi*, which is a stock component and comes with the Pox installation. The multi component is a slight extension of the *l2_learning* component, which is also a stock component. The multi component has the advantage that all switches learn where a MAC address is, when one switch does. In this way they share acquired information between them [52].

In chapter 3 we defined our protocol such that delegators would interact with routers (switches). But if we want to use the Pox API for communicating with switches, we would have to send a message from a delegator to the controller, which in turn sends the message to the switches. To do this, we implemented a Pox component, which would be given as an argument to Pox, just like the stock components are given as arguments.

The controller interacts with switches and delegators. When delegators have received a signal (panic or stop), it processes it and relays a *start throttle* or *stop throttle* message to the controller. The victims IP address is sent along with whether or not throttling should be started or stopped. Upon receiving this message, the controller either enacts a new throttle rule for switches or removes an old one.

In section 4.4 we showed how we can install queues for OVS switches. Now that queues have been installed we can use them by installing new flows in a switch's flow table. An example of such a flow modification message is seen in listing 4.6.

```python
def SendFlowMod(clientIPAddress, port,
    switchConnectionObject, priority=100, expires=600,
    queue_id=0):
        log.debug("sending throttle message")
        msg = of.ofp_flow_mod()
        msg.priority = priority
        msg.idle_timeout = expires
        msg.hard_timeout = expires
        msg.match.dl_type = 0x0800
        msg.match.nw_dst = clientIPAddress
        msg.actions.append(of.ofp_action_enqueue(port=port,
            queue_id=queue_id))
        switchConnectionObject.connection.send(msg)
```

Listing 4.6: A standalone function, which can be used to send a flow modification to an OVS switch. Idle_timeout means that a flow is dropped after a specified amount of seconds, if that flow has not been used. Hard_timeout will drop a flow no matter what.

The function from listing 4.6 requires that a *queue_id* is specified, which means that we can specify a rule that would have an accompanying action called *enqueue* (on line 9), which would place a packet into that queue. And the function from listing 4.6 is set up in such a way that the IP address of a victim can be passed to it, such that the rule only matches traffic going to the victim address. The throughput of traffic going towards the victim is thereby limited.

4.6 Detection & monitoring of traffic data

To to able to view all data, which is being transmitted in Mininet, we use sFlow. We have the following reasons for being interested in this data:

- We would like to be able to view all traffic that is being generated, such that it is possible to determine if our protocol has an effect on the transmitted data.
- FastNetMon needs to monitor a host, or a set of hosts. By sending all data to FastNetMon it can be configured to only listen to traffic directed towards victims. FastNetMon provides very fast DoS detection, based on thresholds (packets-per-second, bandwidth usage or flows).

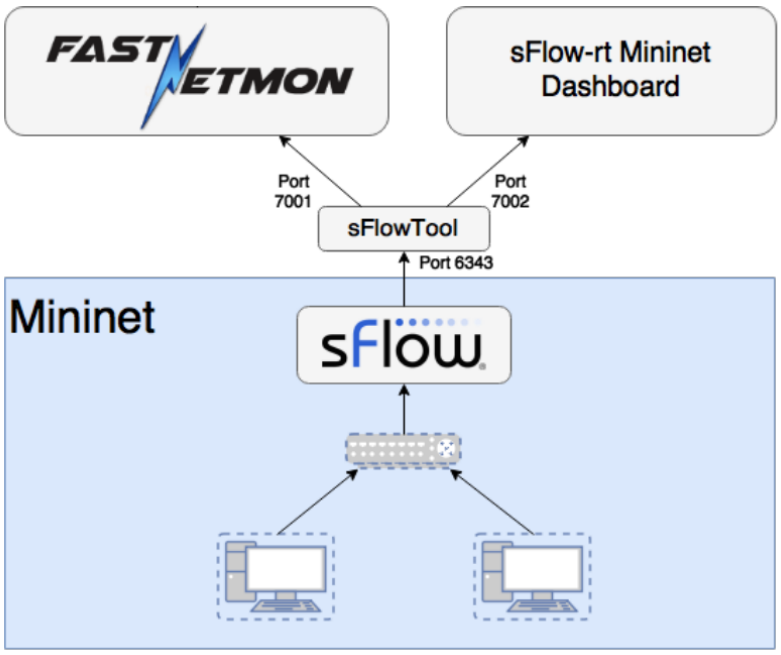
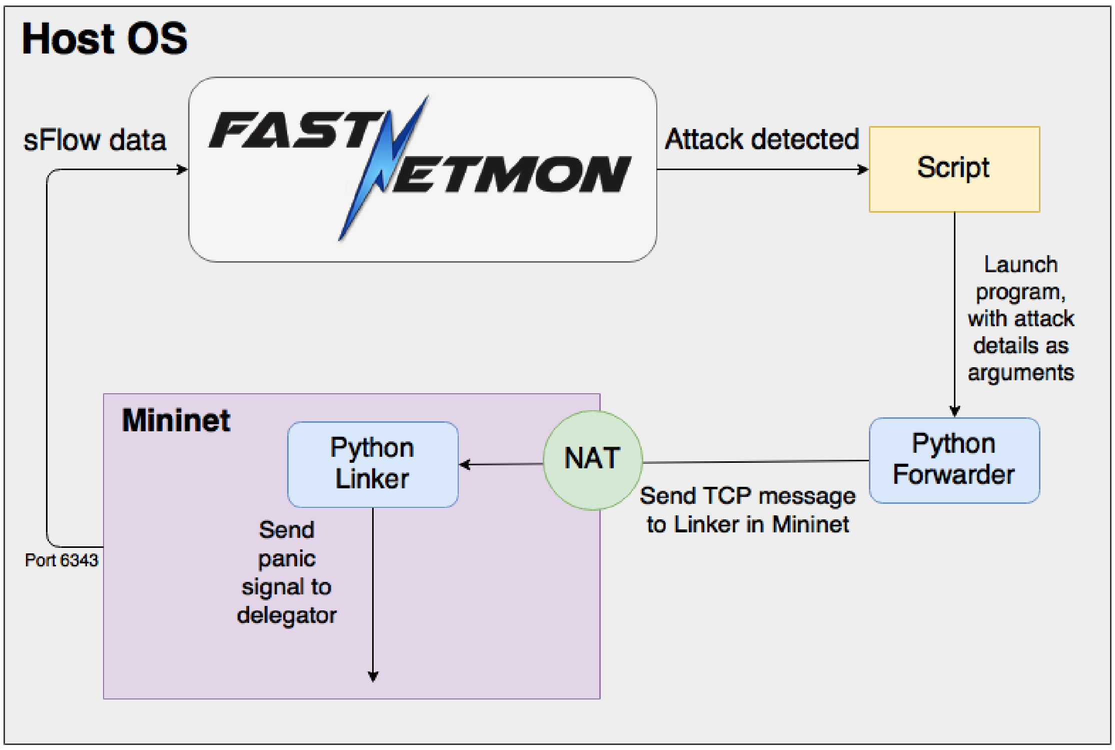


Figure 4.13: A diagram showing how we can use sFlow to monitor traffic flows in Mininet.

As for visualization of data, we are using Mininet dashboard [53], attached to FastNetMon to accurately monitor traffic for the victim specifically. This data is the data that FastNetMon uses for its detection, and therefore is very useful to visualize. As for the entirety of Mininet, we are using a simple dashboard addon for sFlow-rt to plot out traffic for the whole network. sFlow-rt is another service running outside of Mininet, listening to sFlow data from Mininet. As sFlow within Mininet only outputs data on port *6343*, we use *sFlowTool* to forward this port to port *7001* and *7002*, where FastNetMon and sFlow-rt can listen respectively. If we did not employ this forwarding, only one process could listen to port *6343* at a time. Figure 4.13 shows how we have set up the configuration.



Host OS
sFlow data
FASTNETMON
Attack detected
Script
Launch program, with attack details as arguments
Mininet
Python Linker
NAT
Python Forwarder
Send TCP message to Linker in Mininet
Port 6343
Send panic signal to delegator

Figure 4.14: The figure shows what happens when an attack is detected by FastNetMon.

In figure 4.14, we show how we have constructed the sequence for what happens when an attack is detected by FastNetMon. When FastNetMon detects an attack, a *notify_about_attack.sh* script is run, parsing details about the attack, via *stdin*. In this script we read in the details, and execute a small Python program (*Forwarder*), which will open a socket and send the attack information to a *Linker* inside Mininet, where the data will be forwarded to a delegator. For outside processes (the Forwarder) to communicate with hosts inside Mininet, a *network address translation* component (NAT) is necessary. This NAT can be provided by Mininet itself.

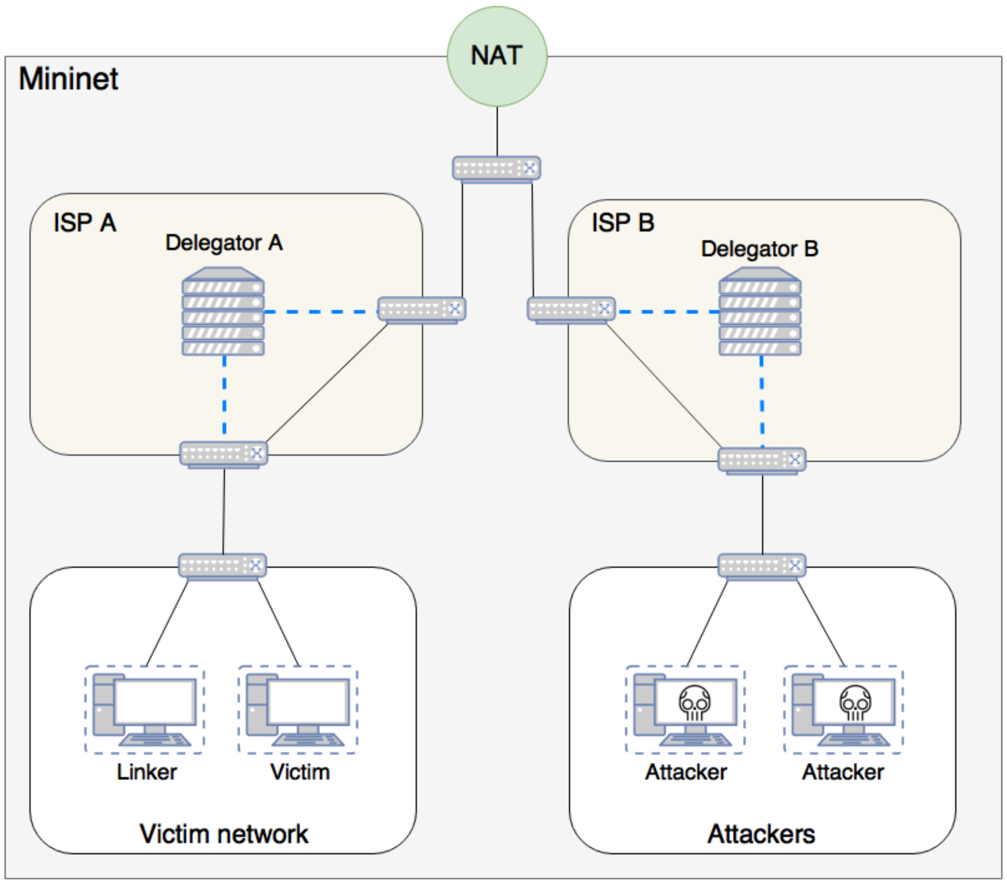


Figure 4.15: An example configuration in Mininet, where we implement ISPs between a victim and attackers. While these are not technically ISPs and subnets, we will refer to them as such.

In figure 4.15 we see a network in Mininet, consisting of two ISPs, each of which has a subnet behind them. This means that traffic from the *attackers* will pass through ISP A and B, where our protocol will be responsible for throttling this traffic. In figure 4.15 the blue dashed line means that OpenFlow is used to manage the ISP's switches. When an attack has been detected, the *Forwarder* sends a message to the *Linker*. The Forwarder sits outside of Mininet, but reaches the Linker through its IP, available through the NAT. The Linker then sends the panic signal out to Delegator A. Here, Delegator A will process this signal and proceed to propergate the signal to Delegator B. Now both delegators have recieved the panic signal, and the protocol will take effect. We use the Forwarder/Linker solution, because our detection tool (FastNetMon), runs externally to Mininet. Had it run on a host within Mininet, in the *victim network*, the Forwarder could have interacted directly with the delegator. But we have chosen not to do this, for the simple reason that we would like to avoid the volatility that comes with running programs on Mininet hosts, as well as the performance issues it might cause. In the real world, the detection tool would run on a host within the victim network, alongside potential victims.

4.7 Testing

In this section we will perform tests on a proof-of-concept implementation of our reaction protocol. The reason for doing a proof-of-concept implementation is to determine if the reaction protocol, by itself, works in code without also configuring detection software and other components. To begin with we will go over the structure of the implementation we are going to test. We will then test the designed reaction protocol on this implementation as well as test the throttling of traffic.

4.7.1 Small scale proof-of-concept

The structure of the proof-of-concept topology we are going to implement can be seen in figure 4.16. We will still be using Mininet as well as the POX controller to construct and control the implementation. In this setup we used the Pox controller with the *l2_learning* component, which is all that is necessary considering we are using a very small topology.

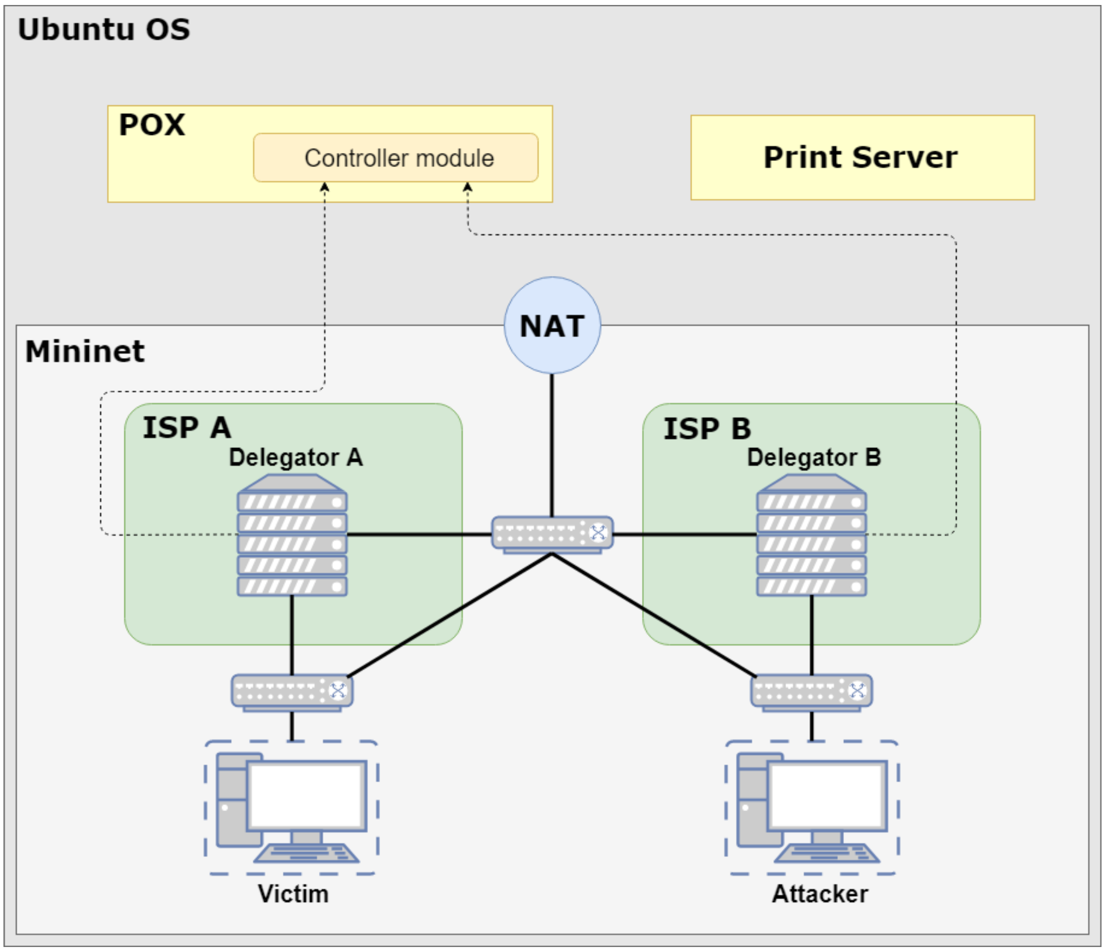


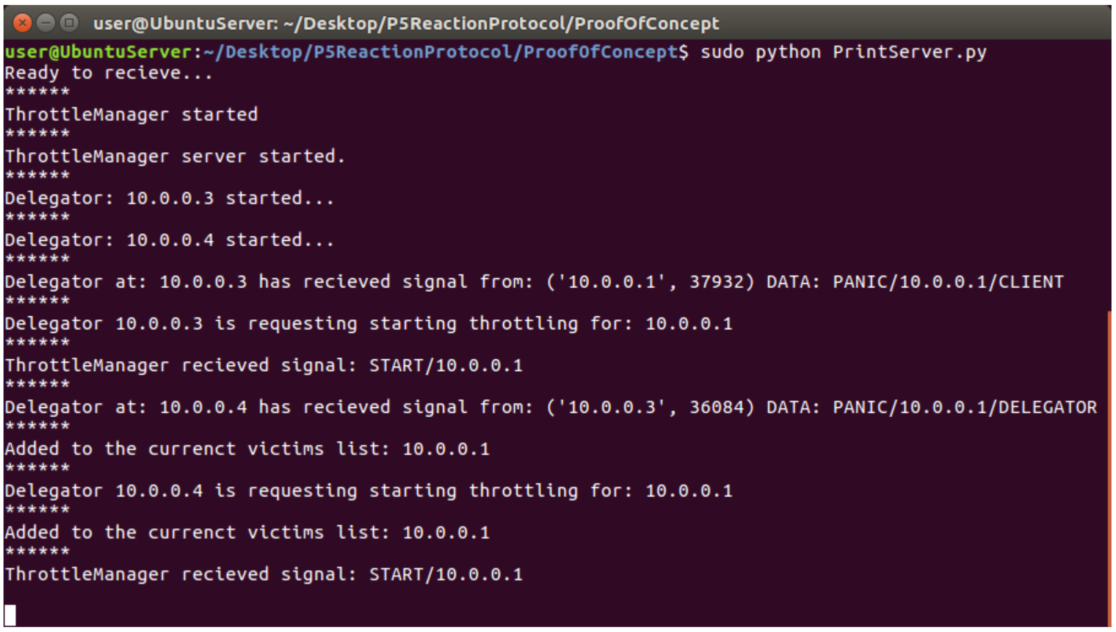
Figure 4.16: A complete overview of how the proof-of-concept implementation is set up.

The overview shows two hosts, a *victim* and an *attacker* connected to two ISPs *A* and *B* through two switches. The victim sits behind ISP A and the attacker sits behind ISP B. The two ISPs in this case consist of a delegator with access to two switches. Each delegator is also a host in Mininet. The external Pox controller receives commands from the two delegators, visualized by the dotted lines, and it controls the throttling of all the switches. Data from the hosts, switches and delegators is send to a print server component, so we can see whats happening in the network without having individual terminals open for each unit.

Running our reaction protocol in Mininet

With the structure of the proof-of-concept topology explained we will now show that the designed reaction protocol can run on an implementation of the topology from figure 4.16.

Figure 4.17 shows an example of the messages exchanged between hosts and delegators, when the client gets attacked. This is an implementation of our reaction protocol and the progress through the statements correspond to the steps in the reaction protocol. In this case we have not included messages for acknowledgements. In the figure we can see that the delegator with IP 10.0.0.3 receives a signal from IP 10.0.0.1 which is our client. The signal contains the data *PANIC*, 10.0.0.1 and *CLIENT*. The first data field contains the type of signal, in this case it is a panic signal. The second data field is the IP that incident originates from and the last data field is who where the current signal comes from.



```
user@UbuntuServer: ~/Desktop/P5ReactionProtocol/ProofOfConcept
user@UbuntuServer:~/Desktop/P5ReactionProtocol/ProofOfConcept$ sudo python PrintServer.py
Ready to recieve...
******
ThrottleManager started
******
ThrottleManager server started.
******
Delegator: 10.0.0.3 started...
******
Delegator: 10.0.0.4 started...
******
Delegator at: 10.0.0.3 has recieved signal from: ('10.0.0.1', 37932) DATA: PANIC/10.0.0.1/CLIENT
******
Delegator 10.0.0.3 is requesting starting throttling for: 10.0.0.1
******
ThrottleManager recieved signal: START/10.0.0.1
******
Delegator at: 10.0.0.4 has recieved signal from: ('10.0.0.3', 36084) DATA: PANIC/10.0.0.1/DELEGATOR
******
Added to the currenct victims list: 10.0.0.1
******
Delegator 10.0.0.4 is requesting starting throttling for: 10.0.0.1
******
Added to the currenct victims list: 10.0.0.1
******
ThrottleManager recieved signal: START/10.0.0.1
```

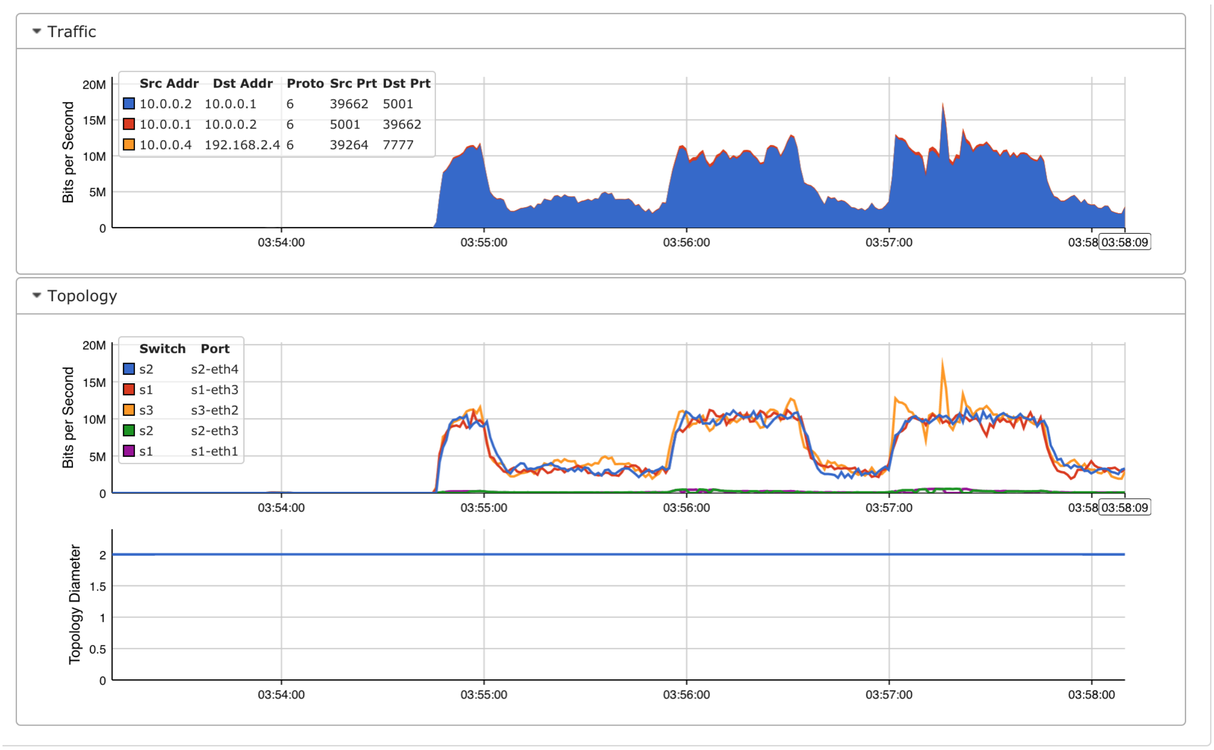
Figure 4.17: Print statements from the different hosts showing the progress of the reaction protocol

In subsection 3.5.3 we discussed what data should be included in the transmitted messages, and subsequently how the delegators would have to keep track of this data. Since this was a proof-of-concept, we did not find it necessary to send the signals exactly the way we described, however the points made in subsection 3.5.3 still stand, which means that in a full implementation we would have to adhere to those design choices.

Performing throttling through the Pox controller

With the reaction protocol running on the proof-of-concept implementation we can now implement throttling of traffic in the topology and test if it works.

We tested our protocol by performing an *iperf* test against the client with IP address 10.0.0.1. When the iperf test was running, we manually instructed the client to send panic signals and stop throttle signals. Figure 4.18 shows how traffic is affected when the different signals are being transmitted.



	Src Addr	Dst Addr	Proto	Src Prt	Dst Prt
🟦	10.0.0.2	10.0.0.1	6	39662	5001
🟥	10.0.0.1	10.0.0.2	6	5001	39662
🟧	10.0.0.4	192.168.2.4	6	39264	7777

	Switch	Port
🟦	s2	s2-eth4
🟥	s1	s1-eth3
🟧	s3	s3-eth2
🟩	s2	s2-eth3
🟪	s1	s1-eth1

Figure 4.18: The top graph of this figure shows throttling decreasing at 03:55 and then an increase in traffic at about 03:56. Then about 30 seconds later another panic signal is sent, and at 03:57 another stop throttle signal is sent, which decreases and increases the amount of traffic being transmitted in the network, respectively. The second graph shows the different interfaces for all the switches that have relatively matching traffic patterns, which is as expected, since all switches are instructed to throttle.

Chapter 5

Conclusion

We will in this chapter evaluate the project and the implementation of the reaction protocol as a whole and discuss to which degree we have satisfied our problem statement and the requirements. Through this we will reach a conclusion, to see if the stated problem can be considered solved. Additionally, we will explore what we see as major issues of the project and what could or should have been done differently in order to avoid these issues.

5.1 Result of the project

In this section we will sum up the result from the project. The result consist of the design and implementation of our reaction protocol.

The result achieved through this project has been the design of a reaction protocol verified through Pabble. The protocol has been tested through a proof-of-concept implementation. The proof-of-concept showed that we were able to establish connections between the different roles described in the Pabble language and have those entities communicate as dictated by the design. We also implemented throttling in the test and were able to illustrate that we could reduce the incoming traffic to a victim, through the use of a virtual network setup and a software defined networking controller.

5.2 Discussion

In this section we will discuss the weaknesses of the project, mostly concerning the reaction protocol as well the implementation thereof. We will be examining the severity of the issues, their causes, possible ways of mitigation and how they affect the viability of the implemented protocol.

Full implementation vs. a proof-of-concept

In section 4.1 we described the intended implementation and showed a figure of how the different components would interact. During the full implementation phase there were many separate issues with the different components. One of the main difficulties was running python scripts on different hosts in Mininet. We showed an example in section 4.2 of how to run two python scripts on two different hosts, however it turned out that when scaling this operation to 8-15 hosts resulted in some unexpected behavior, where sometimes a command to a host such as $h1.cmd("pythonsomepythonscript.py")$, required that the operation be executed in a separate thread, since otherwise the main thread would unexpectedly halt. It was also necessary to have timers between initiating scripts on Mininet hosts. Sometimes the threading library would result in an exception being thrown, because separate instances would try and retrieve some object, which would result in a race condition, hence the exception. Using $time.sleep()$ between each assignment would usually "solve" this problem.

Another more general issue was configuring components to be able to be used in more general cases, such that they were scalable to any topology with any number of ISPs, delegators, victims and attackers. In the beginning we wanted to use FastNetMon to listen to a specific victim, which was easily achievable. But during implementation we thought of more interesting ways to use an IDS. It is for example possible to initialize FastNetMon to act as an IDS for multiple hosts in Mininet, however this added additional complexities such as having to dynamically allocate victims to FastNetMon and have FastNetMon monitor all victims at the same time.

In hindsight we think it would have been a good idea to first make a proof-of-concept implementation of the reaction protocol to determine if it had merit. After this we could then have expanded upon the proof-of-concept to work towards an implementation that resembles the proposed solution in the beginning of chapter 4.

Joining the reaction protocol

From section 3.5, which revolves around the design of our reaction protocol, we observe that it is currently not possible for clients and ISPs to join the reaction protocol. When we run the reaction protocol there is a set amount of clients and ISPs for each run. However, this would most likely not reflect how this would be handled in a real world scenario. This is an oversight on our part. A possible solution to this issue would be to also design and implement a protocol that would allow clients and ISPs to join the reaction protocol.

Specifically for when ISPs join the reaction protocol, we would have to make sure that other ISPs that already implement the reaction protocol, are informed about this event. The reason for this is because, as we know, all ISPs have to be able to communicate with each other through the broadcasts that are sent between them. This means that the ISPs that are already part of the reaction protocol will have to broadcast, and expect an acknowledgement from, the newly joined ISP. This also means that a newly joined ISP will have to know the ISPs that are already part of the reaction protocol.

Reaction protocol authentication

Looking at the network setup in figure 3.6 we observe that there is a need for authentication if such a setup was to be used. We need authentication between clients and their ISPs in order to not have a situation where an attacker can simply make use of IP spoofing and send a signal while pretending to be a client in order to throttle the traffic towards the actual client. Authentication is not something we implemented for the current reaction protocol, since it was not a part of the requirements for our solution. However if we were to implement authentication the following requirements would need to be fulfilled:

Lightweight

The authentication process needs to be lightweight and fast. The reason for this is because we need to handle the client's request in a timely manner so that a given DDoS attack has a negligible effect.

Centralized point of failure

In section 3.5 we look at possibilities for our protocol design. The network setup shown in figure 3.6 and the subsequent protocols can lead to problems. While the protocol is designed to mitigate DDoS attacks on clients, it does not do anything to protect the delegators in the system. This means that attackers could just target the delegators to make sure that they are not able to broadcast the panic signals from other targets. If we were to improve on our DDoS prevention to mitigate this problem, the solution would need to fulfill these requirements:

- Detect when a delegator is victim of a DDoS attack
- Alert the other delegators of the attack so they can respond by restricting traffic to the affected delegator

Some possible solutions to this problem are presented below.

- One solution requires that the ISP provide the delegators and itself with DDoS detection. The ISP itself becomes a client, except it needs to send the panic signal to multiple delegators, including ones outside of its control. Without this ability, the ISP risks a DDoS attack blocking the panic signal to all its delegators before the signal can be broadcast to the delegators of other ISPs. This will require writing a new protocol specifically for ISP clients.
- Another solution involves writing a protocol that makes the delegator periodically contact a number of other delegators. If the delegator can not establish a connection, it has either been disconnected from the network or is itself experiencing a DDoS attack and therefore begins broadcasting a panic signal with itself as the victim.

Choosing between the given solutions and implementing them is a problem that can be dealt with in the future.

Handling of different DDoS attacks

The reaction protocol relies on third-party software for detecting a DDoS attack, the Intrusion Detection Systems (IDS), in this case FastNetMon. While FastNetMon is capable of detecting different forms of attack, the reaction protocol can only protect the victim from being flooded. In order to defend against attacks like a teardrop attack or other types of attack, we would need to extend the current reaction protocol, such that the reaction protocol can send different kinds of messages, depending on the type of attack. A different kind of message would be transmitted based on what the IDS detected. Extending the protocol would not be the time consuming part, however implementing a throttling method to handle each situation would be.

Implemented throttling

The proof-of-concept was implemented with a binary throttling method. The network throttling is either fully active or off, depending on what signal is sent from the victim. This means we still need to develop a dynamic throttling method for the reaction protocol. Without this, the protocol would have to throttle all victims to the predefined value set by the ISP until a stop throttling signal is received.

5.3 Conclusion

We will conclude the project, answer how much has been achieved, and determine if the goal of the project has been reached. To evaluate this we will take a look at the problem statement from section 1.9 and answer the questions stated therein. The problem statement is the following:

Is it possible to construct a system, that interfaces with at least one current detection solution[1]*, that can, based on the one or more integrated detection systems, monitor a network for DDoS activity, stemming from botnets, and subsequently take appropriate action, that either restricts the amount of DDoS traffic leaving the network or block it entirely?*

From this statement questions arose, it is these questions we will answer and through these evaluate if we have satisfied the problem presented in the problem statement. The questions are listed below:

- *What method will be used to monitor network traffic?*
- *Which existing detection systems should be used?*
- *On what kind of device should the system be implemented to monitor a whole network?*

For the first question we decided to use passive monitoring. In contrast to active monitoring, the collection of network data for analysis using passive monitoring was possible without increasing network traffic. This method is used by a number of IDS, giving us a real choice between several possible existing tools.

For the second question we decided to use FastNetMon which is an existing IDS specifically for detecting DDoS attacks. FastNetMon was easy to set up and to configure in order to be used in the solution. This meant that we could easily set it up to detect when a client is getting attacked which was what we needed for our protocol to work. In section 4.6 looked into how we use FastNetMon and how it works together with our other software.

For the third question the system in question turned into a reaction protocol that both clients and ISPs will implement. The reaction protocol was made in order to mitigate DDoS attacks targeted at the clients of the protocol by having ISPs throttling traffic going towards the client under attack. The design of the protocol can be seen in section 3.5, the throttling methods can be seen in section 3.3. It is worth mentioning that we currently handle the different types DDoS attacks in the same way by using our throttling method. This is however not effective for all types of attacks.

We can from the above answers to the questions from the problem statement conclude that the project has reached its goal, with a few shortcomings. Namely that we mitigate the different types of DDoS attacks in the same way. Another shortcoming is that we do not have a full implementation. However, the design and proof-of-concept implementation of the reaction protocol have shown that the problem is solvable.

[1]Those detection systems described in section 2.2

5.4 The project in retrospect

In this section we will reflect upon possible changes that we could have made to make the project and implementation of the reaction protocol better fulfill the problem statement.

Finish the intended implementation

The reason for this is that with this implementation, as we described it, we should able to do more interesting tests, such as easily being to create complex, but realistic, topologies. Perform tests in a more flexible environment, meaning that we can easily scale the amount of clients/delegators/attackers in the network. With this implementation our protocol should also be able to handle multiple panic signals, from different clients, being sent into the network. Therefore if we could continue the project, our first priority would be to finish the implementation as described in 4.1, where the subsequent priorities would be to implement authentication etc. as mentioned in 5.2. One of the ways we would go about the implemention would be to go back and design components, through methods from OOA&D [54], as the complexity of the *support*[2] components as a whole were much greater than expected.

Reflections

In this project we not only acquired factual knowledge regarding networks and protocols, but also knowledge about our work in the group.

Knowledge sharing

Our group began having problems with sharing knowledge as the timetable slipped more and more behind. This will need addressing in future projects, either by implementing regular meetings for knowledge sharing in our schedule or by having some of our team members in rotation between the bigger tasks.

Timetables and plans

We learned that we need backup plans for when we fall behind schedule. Changes to our project close to deadline gave us an unnecessary workload that might have been avoided, if we had thought of an alternative plan for the project for if we would fall behind. A backup plan involving a change in scope might save us some problems in the future.

[2]Components such as the forwarder, which have no real application other than being important in our setup, because we had a need for applications to be able to communicate from inside Mininet to the outside, and vice versa.

Bibliography

[1] Felix C. Freiling, Thorsten Holz, and Georg Wicherski. “Botnet Tracking: Exploring a Root-cause Methodology to Prevent Distributed Denial-of-service Attacks”. In: *Proceedings of the 10th European Conference on Research in Computer Security*. ESORICS’05. Milan, Italy: Springer-Verlag, 2005, pp. 319–335. ISBN: 3-540-28963-1, 978-3-540-28963-0. URL: http://dx.doi.org/10.1007/11555827_19.

[2] Yury Namestnikov. *The Economics of Botnets*. https://securelist.com/the-economics-of-botnets/36257/. 2009. (Visited on 09/21/2017).

[3] Sérgio SC Silva, Rodrigo MP Silva, Raquel CG Pinto, and Ronaldo M Salles. “Botnets: A survey”. In: *Computer Networks* 57.2 (2013), pp. 378–403.

[4] Ahmad Karim, Syed Adeel Ali Shah, and Rosli Salleh. “Mobile Botnet Attacks: A Thematic Taxonomy”. In: *New Perspectives in Information Systems and Technologies, Volume 2*. Ed. by Álvaro Rocha, Ana Maria Correia, Felix . B Tan, and Karl . A Stroetmann. Cham: Springer International Publishing, 2014, pp. 153–164. ISBN: 978-3-319-05948-8. URL: https://doi.org/10.1007/978-3-319-05948-8_15.

[5] Tim April Manos Antonakakis, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. “Understanding the Mirai Botnet”. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 1093–1110. ISBN: 978-1-931971-40-9.

[6] F.E. Froehlich and A. Kent. *Security of the internet*. Encyclopedia of Telecommunications. Taylor & Francis, 1997, pp. 231–255. ISBN: 9780824729134.

[7] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. 1st. New York, NY, USA: John Wiley & Sons, Inc., 2001. ISBN: 0471389226.

[8] Eng Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. “A Survey and Comparison of Peer-to-peer Overlay Network Schemes”. In: *Commun. Surveys Tuts.* 7.2 (2005), pp. 72–93. ISSN: 1553-877X. DOI: 10.1109/COMST.2005.1610546. URL: http://dx.doi.org/10.1109/COMST.2005.1610546.

[9] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach (6th Edition)*. 6th. Pearson, 2012. ISBN: 0132856204, 9780132856201.

[10] Laszlo Lovasz. “Random walks on graphs: A survey”. In: *Combinatorics, Paul Erdos in Eighty* 2 (1993).

[11] Paul Barford and Vinod Yegneswaran. “An Inside Look at Botnets”. In: *Malware Detection*. 2007, pp. 171–191. DOI: 10.1007/978-0-387-44599-1_8. URL: https://doi.org/10.1007/978-0-387-44599-1_8.

[12] *Peer-to-Peer Botnets for Beginners*. URL: https://www.malwaretech.com/2013/12/peer-to-peer-botnets-for-beginners.html (visited on 09/21/2017).

[13] Ping Wang, Sherri Sparks, and Cliff C. Zou. “An Advanced Hybrid Peer-to-Peer Botnet”. In: *First Workshop on Hot Topics in Understanding Botnets, HotBots’07, Cambridge, MA, USA, April 10, 2007*. 2007. URL: https://www.usenix.org/conference/hotbots-07/advanced-hybrid-peer-peer-botnet.

[14] Evan Cooke, Farnam Jahanian, and Danny McPherson. “The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets.” In: *SRUTI* 5 (2005), pp. 1–6.

[15] Steven DeFino and Larry Greenblatt. *Official Certified Ethical Hacker review guide*. Delmar, 2012.

[16] Eric W. Weisstein. *"Birthday Problem." From MathWorld–A Wolfram Web Resource.* http://mathworld.wolfram.com/BirthdayProblem.html. (Visited on 10/02/2017).

[17] Pierrick Guingo Chao Kan. *Passive network monitoring system.* https://www.google.com/patents/US7483379. 2002.

[18] Dan Sullivan. “What’s on Your Network? The Need for Passive Monitoring”. In: (2013).

[19] Karen A. Scarfone and Peter M. Mell. *SP 800-94. Guide to Intrusion Detection and Prevention Systems (IDPS)*. Tech. rep. 2007.

[20] Giovanni Vigna. “Static Disassembly and Code Analysis.” In: *Malware Detection* 27 (2007), pp. 19–41.

[21] Imperva. *DDoS Attacks*. https://www.incapsula.com/ddos/ddos-attacks/.

[22] *Understanding Teardrop Attacks*. URL: https://www.juniper.net/documentation/en_US/junos/topics/concept/denial-of-service-os-teardrop-attack-understanding.html (visited on 11/13/2017).

[23] *DDoS Attack Types: Glossary of Terms, subsection IP Null*. URL: https://www.corero.com/resources/glossary.html#IP%20NULL (visited on 11/13/2017).

[24] *Smurf DDoS Attack Type*. URL: https://www.corero.com/resources/ddos-attack-types/smurf-ddos-attack.html (visited on 11/13/2017).

[25] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. "Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites". In: *Proceedings of the 11th international conference on World Wide Web*. ACM. 2002, pp. 293–304.

[26] *Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing*. 2000. URL: https://tools.ietf.org/html/rfc2827.

[27] Monowar H Bhuyan, Hirak Jyoti Kashyap, Dhruba Kumar Bhattacharyya, and Jugal K Kalita. "Detecting distributed denial of service attacks: methods, tools and future directions". In: *The Computer Journal* 57.4 (2013), pp. 537–556.

[28] Pavel-odintsov. *Fastnetmon*. https://github.com/pavel-odintsov/fastnetmon. 2017.

[29] John Murphy. *FlowTraq*. www.flowtraq.com.

[30] Johanna, Vlad Grigorescu, Jon Siwek, Matthias Vallentin, Robin Sommer, and Seth Hall. *Bro IDS*. www.bro.org.

[31] Johanna, Vlad Grigorescu, Jon Siwek, Matthias Vallentin, Robin Sommer, and Seth Hall. *Bro Github*. https://github.com/bro.

[32] Daniel B. Cid, Jeremy Rossi, Dan Parriott, Scott R. Shinn, Santiago Bassett, Brad Lhotsky, Andrew Widdersheim, Vic Hargrave, and Jia-Bing (JB) Cheng. *OSSEC*. www.ossec.github.io.

[33] Martin Roesch et al. "Snort: Lightweight intrusion detection for networks." In: *Lisa*. Vol. 99. 1. 1999, pp. 229–238.

[34] Cisco. *Snort frequently asked questions*. URL: https://www.snort.org/faq/what-is-snort.

[35] Cisco. *Snort User Manual 2.9.1, subsection: Writing Snort Rules*. URL: http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node27.html.

[36] Paulo Angelo. *Hogzilla*. www.ids-hogzilla.org.

[37] Paulo Angelo. *Hogzilla Github*. www.github.com/pauloangelo/hogzilla.

[38] Open Information Security Foundation. *Suricata*. www.suricata-ids.org.

[39] Open Information Security Foundation. *Suricata Github*. www.github.com/OISF/suricata.

[40] Duncan Langford. “Ethical Issues in Network System Design”. In: *Australasian J. of Inf. Systems* 4.2 (1997). URL: http://journal.acs.org.au/index.php/ajis/article/view/367.

[41] *Scribble: Describing Multi Party Protocols*. URL: http://www.scribble.org/index.html (visited on 11/13/2017).

[42] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. “The Scribble protocol language”. In: *International Symposium on Trustworthy Global Computing*. Springer. 2013, pp. 22–41.

[43] Kohei Honda, Nobuko Yoshida, and Marco Carbone. “Multiparty Asynchronous Session Types”. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’08. San Francisco, California, USA: ACM, 2008, pp. 273–284. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328472. URL: http://doi.acm.org/10.1145/1328438.1328472.

[44] *Pabble Parameterised Scribble*. URL: http://mrg.doc.ic.ac.uk/tools/pabble/ (visited on 11/13/2017).

[45] *Mininet*. URL: http://mininet.org/ (visited on 12/17/2017).

[46] *The POX Controller*. https://github.com/noxrepo/pox. 2017.

[47] *Features*. https://gephi.org/features/. (Visited on 12/20/2017).

[48] *Software-Defined Networking (SDN) Definition*. https://www.opennetworking.org/sdn-definition. (Visited on 12/06/2017).

[49] Nick McKeown. “Software-defined networking”. In: *INFOCOM keynote talk* 17.2 (2009), pp. 30–32.

[50] *OpenFlow Switch Specification - Version 1.5.1*. URL: \url{https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf}.

[51] The Linux Foundation. *Open vSwitch Manual*. http://openvswitch.org/support/dist-docs/ovs-vsctl.8.txt.

[52] *Pox wiki*. https://openflow.stanford.edu/display/ONL/POX+Wiki. (Visited on 12/20/2017).

[53] *Mininet Dashboard*. URL: https://github.com/sflow-rt/mininet-dashboard (visited on 12/18/2017).

[54] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel, and Nielsen Jan Stage. "Object-oriented analysis & design". In: (2000).

Appendices

A Local Pabble protocols

```
module Reaction;

local protocol Reaction at Client(role Delegator[1..N],
    role Router[1..H]) {
     rec START {
         choice at Client {
             PanicSignal(string) to Delegator[j];
             SignalReceived() from Delegator[j];
             continue START;
         } or {
             SeizeThrottling(string) to Delegator[j];
             SignalReceived() from Delegator[j];
             continue START;
         }
     }
}
```

Listing 1: Local protocol for the client participant

```
module Reaction;

local protocol Reaction at Delegator[1..N](role Client,
    role Delegator[1..N], role Router[1..H]) {
     rec START {
         choice at Client {
             if Delegator[j] PanicSignal(string) from Client;
             if Delegator[j] SignalReceived() to Client;
             foreach (x:1..N except j) {
                 if Delegator[x] PanicSignal(string) from
                    Delegator[j];
```

```
                if Delegator[j] PanicSignal(string) to
                   Delegator[x];
                if Delegator[j] SignalReceived() from
                   Delegator[x];
                if Delegator[x] SignalReceived() to
                   Delegator[j];
            }
            foreach (y:1..H) {
                if Delegator[j] ThrottleTrafic() to
                   Router[y];
                if Delegator[j] SignalReceived() from
                   Router[y];
            }
            continue START;
        } or {
            if Delegator[j] SeizeThrottling(string) from
               Client;
            if Delegator[j] SignalReceived() to Client;
            foreach (x:1..N except j) {
                if Delegator[x] SeizeThrottling(string)
                   from Delegator[j];
                if Delegator[j] SeizeThrottling(string) to
                   Delegator[x];
                if Delegator[j] SignalReceived() from
                   Delegator[x];
                if Delegator[x] SignalReceived() to
                   Delegator[j];
            }
            foreach (y:1..H) {
                if Delegator[j] StopThrottleTraffic() to
                   Router[y];
                if Delegator[j] SignalReceived() from
                   Router[y];
            }
            continue START;
        }
    }
}
```

Listing 2: Local protocol for the delegator participant

```
module Reaction;

local protocol Reaction at Router[1..H](role Client, role
    Delegator[1..N], role Router[1..H]) {
    rec START {
        choice at Client {
            foreach (y:1..H) {
                if Router[y] ThrottleTraffic() from
                   Delegator[j];
                if Router[y] SignalReceived() to
                   Delegator[j];
            }
            continue START;
        } or {
            foreach (y:1..H) {
                if Router[y] StopThrottleTraffic() from
                   Delegator[j];
                if Router[y] SignalReceived() to
                   Delegator[j];
            }
            continue START;
        }
    }
}
```

Listing 3: Local protocol for the switch participant