University of Zurich^{UZH}

GoDDSSy: A DDoS Signaling and Control Network based on GossipSub

Tobias Frauenfelder
Zurich, Switzerland
Student ID: 18-914-986

Supervisor: Dr. Bruno Rodrigues, Jan von der Assen,
Prof. Dr. Burkhard Stiller
Date of Submission: November 15, 2022

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

Distributed Denial-of-Service (DDoS) Attacken treten täglich auf und nehmen aufgrund der Beliebtheit des Internets der Dinge immer mehr zu. Da viele dieser Geräte von vornherein unsicher sind, können Botnets diese nutzen, um gross angelegte Angriffe zu starten. DDoS-Angriffe sind stark verteilt, daher umfasst der beste Abwehrplan auch eine verteilte Verteidigung, um den Angriff an mehreren Standorten zu reduzieren, die möglicherweise näher an ihrem Ursprung liegen.

In dieser Arbeit wird versucht, ein DDoS-Signalisierungsnetzwerk mit dem Namen Gossip-Sub DDoS Signaling System (GoDDSSy) zu entwickeln, das in einer vertrauenswürdigen Umgebung betrieben wird und agil und widerstandsfähig ist.

Der Leser erhält zunächst einen theoretischen Überblick über die Grundlagen von DDoS-Angriffen und Publish-and-Subscribe-Systemen. Ausserdem werden ähnliche Arbeiten zusammengefasst. Anschliessend wird das Design von GoDDSSy vorgeschlagen, das anschliessend implementiert, eingesetzt und evaluiert wird.

Die Ergebnisse zeigen, dass GoDDSSy agil und widerstandsfähig funktioniert. Darüber hinaus wird deutlich, wo die Grenzen des implementierten Systems liegen und wie es im Vergleich zu ähnlichen Systemen abschneidet.

Distributed Denial-of-Service (DDoS) attacks occur daily and are growing in size due to the Internet of Things (IoT) popularity. Since many of these devices are designed to be insecure, botnets can use them to launch large-scale attacks. DDoS attacks are highly distributed; thus, the best counter plan also includes a distributed defense to reduce attacking traffic at several locations that may be closer to their origins.

This thesis tries to develop a DDoS signaling network called GossipSub DDoS Signaling System (GoDDSSy), which operates in a trusted environment and is agile and resilient. The reader is first provided with a theoretical overview covering the fundamentals of DDoS attacks and publish-and-subscribe systems. Furthermore, a summary of related work is given. Additionally, the design of GoDDSSy is proposed, which is then later implemented, deployed, and evaluated.

Results show that GoDDSSy operates in an agile and resilient way. In addition, we show the limits of the implemented system and how it performs compared to related systems.

Acknowledgments

I would like to thank my supervisors Dr. Bruno Rodrigues, Jan von der Assen, and Prof. Dr. Burkhard Stiller for letting me write my thesis at the CSG. In particular I would like to express my gratitude to Dr. Bruno Rodrigues for his support throughout the thesis process, for always being available to answer my questions and for always offering advice when things did not go as planned.

Contents

Abstract **i**

Acknowledgments **iii**

1 Introduction **1**

1.1 Thesis Goals 2

1.2 Methodology 2

1.3 Thesis Outline 2

2 Fundamentals **3**

2.1 Background 3

2.1.1 Distributed-Denial-of-Service (DDoS) 3

2.1.2 Publish/Subscribe Systems 5

2.2 Related Work on DDoS Signaling 6

2.3 Tooling 7

3 Design **9**

3.1 Overview 9

3.2 Architecture 9

3.2.1 Components 10

3.3 Communication 12

3.4 GoDDSSy Protocol 12

4 Implementation 15

4.1 Software Stack 15

4.2 Configuration 16

4.3 Core Functionality 16

4.3.1 Configuring the libp2p Node 16

4.3.2 Initialization 17

4.3.3 Peer Discovery 18

4.3.4 The GoDDSSy Protocol 19

4.4 Local Testing 23

5 Evaluation 25

5.1 Evaluation Setup 25

5.2 Scenarios 26

5.3 Results 26

5.3.1 Transfer Limit of Attack Information 26

5.3.2 Transmission Time Versus Amount of Attack Information 27

5.3.3 Determination of the Optimal Time Constraint per Step 29

5.3.4 Disruption by Flooding with Messages 30

5.4 Discussion 31

6 Final Considerations 33

6.1 Summary 33

6.2 Conclusions 34

6.3 Contributions 35

6.4 Future Work 35

Bibliography 35

Abbreviations 41

List of Figures 41

List of Tables **43**

List of Listings **45**

Chapter 1

Introduction

Distributed Denial-of-Service (DDoS) attacks are a major threat to Internet availability that remain unmitigated despite many commercial and research efforts [23]. DDoS attacks happen daily, and as society follows an increasing digitization trend, they can pose a great threat to businesses and individuals. A common reason is the large number of unsecured devices connected to the Internet, and their growing processing capacity, which allows attackers to take control of a vast amount of unsecured devices that range from connected cameras to smart fridges to launch malicious attacks [15, 33]. Many of these devices are insecure by design and are not often impossible to secure due to their hardware and software constraints.

Due to the widely distributed nature of DDoS attacks, an ideal counter strategy also involves a distributed defense to mitigate attacking traffic at different points, possibly closer to their origin [23, 27, 30]. Nonetheless, there exist challenges in different dimensions to provide agile and robust communication between cooperating domains. For example, technical, legal, economic, and social dimensions impact how such a signaling network is deployed and operated [26]. This thesis is limited in technical scope, answering how to provide an agile and resilient network for signaling DDoS attacks within trusted domains. In this sense, this thesis should propose and implement a communication protocol enabling a party to simultaneously communicate with one or multiple parties on an ongoing or past cyber attack.

Other work has covered related DDoS defense areas such as detection [9] and mitigation [19, 35]. Still, major related works can be found in the cooperative DDoS literature, such as [10, 35]. A previous approach [31] developed at the University of Zurich details a signaling protocol using blockchain and smart contracts as a basis to intermediate the cooperation between a target and a mitigator, which has benefits in terms of increased transparency and embedded financial incentive features. Nonetheless, this thesis should approach the issues considering a slightly different set of characteristics involving an agile and resilient signaling protocol to communicate cyber-threats in general.

The design of GoDDSSy is described in this thesis. GoDDSSy's architecture enables decentralized DDoS attack signaling over a publish-and-subscribe paradigm. A protocol is presented that allows actors to participate in the cooperative system. Results demonstrate that GoDDSSy is agile and resilient. Furthermore, observing the implemented system's limitations and how it performs concerning similar systems is possible.

1.1 Thesis Goals

This thesis has as its goals:

- Develop a solution for a cooperative DDoS signaling system that exchanges attack information between one or more systems operating in a trusted environment.
- A definition of agile and resilient should be presented to define quantitative boundaries for assessing the proposed signaling protocol.
- To fulfill the suggested signaling protocol against the definitions of agility and resiliency presented in the thesis, a proof-of-concept should be constructed and assessed.

1.2 Methodology

This thesis can be divided into two parts. The theoretical section includes the fundamentals of DDoS attacks and cooperative DDoS defense, which inspired GoDDSSy. As a result, a new signaling system is designed. The second section focuses on practical matters. The emphasis is on system implementation, configuration, and deployment. Following that, the system is evaluated in various scenarios to determine whether the goals were met.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 discusses the fundamentals of DDoS attacks, how publish and subscribe (pub/sub) systems work, and how other papers have addressed cooperative defense mechanisms. The GossipSub signaling system is proposed in Chapter 3, and the architecture with the individual components and protocol processes is presented. Chapter 4 describes how the system is run and how the functionalities were implemented. Chapter 5 presents various scenarios tested, evaluated, and discussed to evaluate the system. Finally, Chapter 6 summarizes and concludes the work and provides information about future work.

Chapter 2

Fundamentals

2.1 Background

This section provides background on the thesis's fundamentals. It is divided into two sections, one covering the fundamentals of DDoS and providing a broad overview of different types of these attacks, as well as where the challenges lie in detecting them (Subsection 2.1.1), and the other providing insight on publish and subscribe systems and how they work in general (Subsection 2.1.2). Both topics are combined later because they both contribute to the achievement of these thesis's goals.

2.1.1 Distributed-Denial-of-Service (DDoS)

A DDoS attack is a large-scale distributed attack that aims to disrupt the victim's system's service availability. There are several methods of carrying out an attack, but the majority of the time, a lot of traffic is given to the victim to overwhelming its resources. Due to the bandwidth used by the attackers, this attack stops legitimate users from accessing a particular network resource [4]. The duration of DDoS attacks varies. An average DDoS attack lasted just under four hours in the second quarter of 2022, whereas the longest recorded attacks lasted 140 hours [11].

Many DDoS attacks are carried out by zombie botnets. They consist of hijacked devices, which are controlled by the attackers. The growing popularity of Internet of Things (IoT) devices is a benefit for attackers. IoT devices are ideal zombies, since they are continuously connected to the internet and therefore ideal hosts for DDoS Attacks. To hijack a new zombie for a botnet, victim devices are infected with malicious software, which causes them to respond to commands from the command and control server. The control server identifies the target and directs the zombies, which subsequently attack the target. One popular botnet that was discovered for the first time in 2016 is called Mirai. After the creators of Mirai published the source code, other hackers used it to create zombie botnets, which they rented out. Some of them contained as many as 400'000 bots [13].

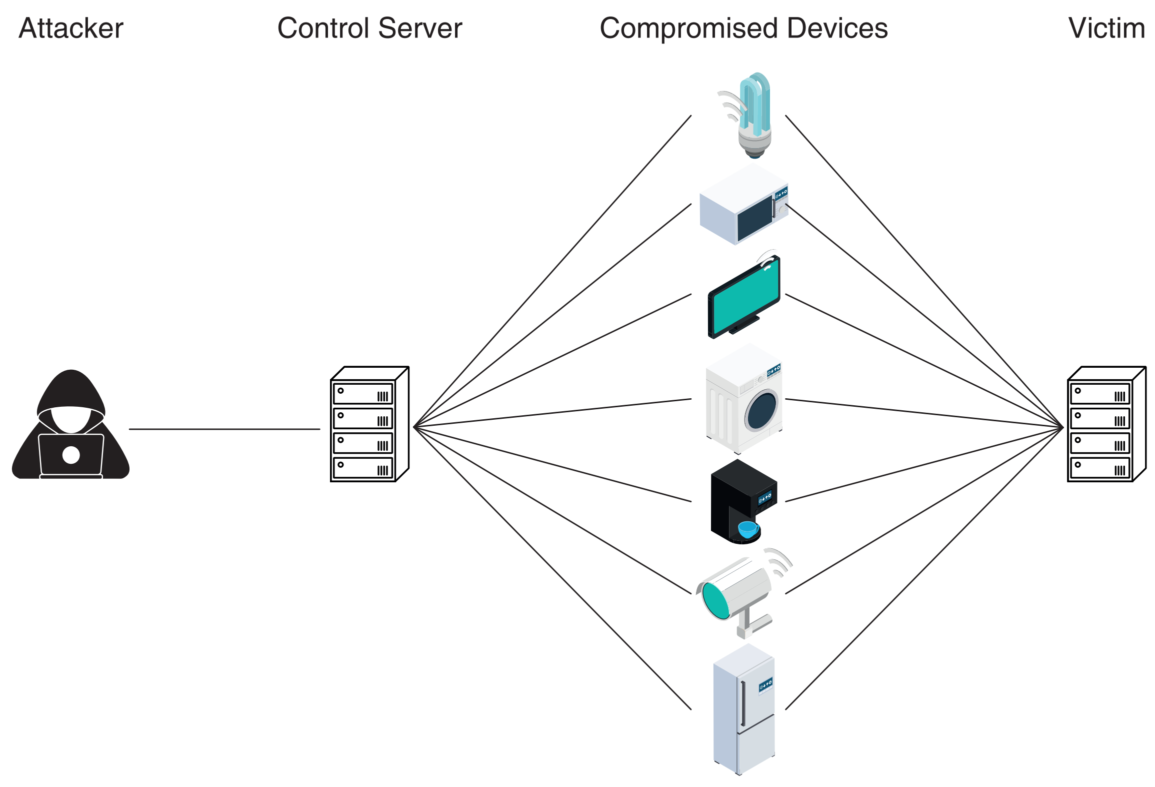

Figure 2.1: DDoS attack carried out by a botnet of compromised devices.

2.1.1.1 Types of DDoS Attacks

There are two primary types of attacks. It can be distinguished between DDoS attacks on the application layer and attacks on the network/transport layer.

Attacks on the application level, which are also called Layer 7 attacks, are used to exhaust a server's resources and target application characteristics such as HTTP, DNS. To carry out an attack like this, knowledge about the application has to be acquired beforehand. As an example, in an HTTP DDoS attack a lot of GET or POST requests are sent to the target which requests or sends images, documents, or other resources. By performing these operations, the application running the service can be overloaded.[35]

Network/transport-level attacks do not directly target an application on a server. They are trying to bring down a server through overtaxing its bandwidth by flooding the network. The most common attacks use SYN flooding and UDP flooding. During an SYN-Flood attack, the 3-Way TCP handshake is exploited. The attacker sends a high amount of SYN packets to the server. The server answers with an SYN/ACK packet and leaves a port open to receive an answer and waits for the ACK packet of the attacker, which he will never send. During a specific waiting time, the server blocks that port. If the amount of SYN-packets exceeds the number of available ports, the server cannot function anymore. [2]

2.1.1.2 Detection of DDoS Attacks

One difficulty of DDoS attacks is detecting them, since it can be challenging to distinguish between malicious and legitimate network traffic. For example, a flooding attack could be mistaken for a web service access request from a flash crowd, which occurs when many legitimate users utilize a web service. However, the following methods exist to discover a DDoS attack [35].

In the Activity Profile method described in [3], network flows are monitored with the help of header information of packets. It is possible to inspect packets with similar header information like address, port, and protocol. A network flow, which consists of numerous consecutive packets with similar characteristics, can be used to calculate the average flow, by measuring the time between them. Then, average packet rates of the inbound and outbound flows can be added to determine network activity. Similar flows can be clustered to reduce high-dimensionality issues and are then monitored. A rise in activity in one cluster can indicate that some agents are increasing their attack rate, while a rise in activity in different clusters can signify a Distributed DoS attack [3].

The Change-Point detection is a different technique for spotting DDoS attacks. This process can be carried out posteriorly or sequentially. This method involves observing a time series of network traffic and determining if it is statistically homogeneous. Because it can be done instantly, the sequential technique is utilized to identify attacks, while the posterior method analyzes the traffic after it has been collected [34].

2.1.1.3 Signaling DDoS Attacks in a Cooperative Defense

Since huge DDoS attacks can potentially overwhelm a single organization's defense system, sharing resources for detection and mitigation is crucial. A signaling platform is used to exchange bidirectional messages, which are used for exchanging information about the attack that is relevant for the mitigation [25, 29]. These messages may include requests for mitigation as well as various status reports [16].

2.1.2 Publish/Subscribe Systems

Publish/subscribe is a paradigm in which publishers can raise an event and subscribers can declare their interests in events so that they are always notified when this kind of event occurs. One can also see the publisher as the producer and the subscriber as the consumer. The event is the information they exchange, and the notification is the information delivery. They communicate with one another via an event manager (EM), who serves as a neutral mediator. If a subscriber is interested in a new event, it uses the EM's subscribe function without knowing where the event is coming from. On contrary, the publisher is unaware of the recipients of his information [8].

There are two established addressing methods in the publish and subscribe pattern. While publish is group-based, subscribe is subject-based. When using the group-based technique, an agent subscribes to a particular user group. The agent is informed if any of these users publish something. The subject-based system is built on a predefined or arbitrary string. Every event has a subject attached to it; under this system, an actor can select the subjects in which he is interested. If an occurrence relating to his topic happens, he is informed [12].

The two main types of architectures of a publish/subscribe system can be split up into a centralized and distributed approach. In the centralized approach, there is only one EM. The EM holds the information about which agent is interested in which information and takes care of new subscriptions and notifications from the subscribers.

The centralized approach is limited to a single EM, which can serve as a bottleneck. To overcome this problem a distributed architecture can be used, which consists of multiple EMs. The distributed architecture can be divided into distributed broadcast and multicast. Every time a message is published, in the distributed broadcast, the message is sent to all the EMs. This can cause a lot of network traffic, since all the EMs are flooded with messages. To bypass this problem, distributed multicast can be used which assures that the EM only forwards the messages, if one the subscribers are subscribed to this event.

If an event is published, the event is forwarded through any EM, who is responsible to inform the other EMs. This approach can create a lot of network traffic. For this reason, there exists an alternative approach called the multicast method. In this method, an event is only forwarded to the next EM, if the EM has a subscriber to the specific event[12].

2.2 Related Work on DDoS Signaling

In the past, there has been a lot of research in the area of Cooperative DDoS Defense and various approaches have been propose to address this problem. In this section, different signaling systems that already exist are presented.

The Cossack, developed by Papadopoulos et al. [20], is an early technique for Cooperative DDoS defense. The project developed an architecture that helps coordinate effective countermeasures to avoid DDoS attacks. With the Cossack, each organization on the network employs watchdog software. The watchdogs then communicate with one another using a peer-to-peer multicast technique. If a target is attacked, a watchdog close to the victim detects the attack and receives data from the intrusion detection system. The data is thereafter sent to other watchdogs via multicast. The signaling is done on an out-of-bound channel to ensure that communication is maintained during an assault. When the other peers receive the attack information, they examine it and take steps to mitigate the attack.

Rashidi and Fung [22] proposed another approach for collaborative DDoS defense. They created the CoFence system, which uses Network Function Virtualization technology and can thus offer a low-cost, flexible solution. CoFence provides a collaborative framework for mitigating DDoS attacks. If one peer is attacked, other peers can offer their spare network resources to assist. The target's traffic is then redirected to other nodes that can help filter the traffic and return the filtered traffic to the target.

CoFence employs three distinct sorts of requests. One for seeking and providing assistance to others, one for adding neighbors to work with, and one for removing them. The system routes these requests through a separate defensive network rather than using the network for normal traffic.

One recent paper about cooperative DDoS defense is the Blockchain Signaling System (BloSS) [28]. BloSS offers a platform on which various autonomous systems can work together to defend against DDoS attacks. BloSS utilizes Blockchain and Smart Contracts to reach agreements regarding services provided by various autonomous systems, such as the exchange of mitigation services [24]. The signaling system also ensures that the reputation of each system actor is monitored. Due to its size and the need to reduce signaling latency, an off-chain P2P network is used to exchange blacklisted address lists.

A strictly defined collaboration protocol is used by BloSS for the signaling to make sure that mitigation services are provided reliably. The protocol defines the interaction between the target (T) and the mitigator (M) [23]. If T is under attack, it sends a mitigation request to M. If M approves the T sends funds to the M, which are locked in the Smart Contract. Then the T writes attack information onto an off-chain storage device that is accessible to the M. From then on, the M has a Service Deadline in which the M has time to mitigate the attack and upload proof for the M. After this deadline, the T has time to rate the services of the M. Subsequently, the M has time to proof, that it has done its work. In the best case, money and services are exchanged.

2.3 Tooling

Pub/sub systems are widely used to support fast communication across distributed systems. This thesis briefly describes selected pub/sub systems outlined in Table 2.3.

	Pub/Sub Frameworks			
Name	**Kafka**	**Emitter**	**RabbitMQ**	**GossipSub**
Developer	Apache	Emitter.io	Pivotal Software	Protocol Labs
Architecture	Distributed	Distributed	Distributed	Decentralized
Written in	Scala, Java	Go	Erlang	Go, Javascript, Rust
Open Source	Yes	Yes	Yes	Yes

Table 2.1: A comparison of different pub/sub libraries.

Emitter is a basic implementation of a pub/sub system. Real-time communication is supported, and it can adapt to various programming languages. It is no more complex than conventional one-to-one communication because it only consists of a simple event broker conveying messages from a sender to a receiver. It supports many-to-one and many-to-many communication, in addition to straightforward one-to-one communication [6].

Many pub/sub systems employ native flooding, which forwards messages to subscribers. This generates a lot of traffic, which is sometimes unnecessary. Because of this, the

GossipSub uses, as the name implies, a method based on gossip that is much more effective. It is an unstructured and unmanaged P2P pub/sub with a networked mesh of agents, with gossip and lazy push functionality added. The lazy push strategy involves only sending metadata to the other peers. Peers can specifically ask for the entire message if they are interested in it. In addition to the mesh used for the exchange of metadata only, each peer also has a limited number of full-message connections. To establish new connections and exchange messages, the control flags GRAFT, PRUNE, IHAVE, and IWANT exist in this protocol. Combining all those technologies makes GossipSub extremely responsive to changing network conditions, which is a key characteristic[15, 32].

Apache Kafka provides a different pub/sub platform. The distributed event store that was described in section 2.1.2 consists of both producers and consumers. The Kafka cluster is split up into various partitions made up of various topics. Different partitions are used to divide up topics. The system may consist of clusters with one or more event brokers, and the topics may be duplicated across various nodes such that a copy is always present. In Kafka, events can be requested as often as necessary and are not lost after delivery. How long each topic's lifetime for an event should last can be specified [1].

Chapter 3

Design

3.1 Overview

The main module built in this thesis is GoDDSSy, short for GossipSub DDoS Signaling System. GoDDSSy consists of different nodes forming a decentralized overlay network, where each node belongs to another independent system and can be integrated on top of it. The system relies on another module that the autonomous system has to have, called the mitigation module. The mitigation module can detect and mitigate DDoS attacks. However, since mitigation is not in the scope of this thesis, it is not further elaborated upon. GoDDSSy's design has different interfaces considering different purposes. One is (a) to exchange information with the mitigation module. In one way, GoDDSSy receives a list of IPv4-Addresses from the Mitigation Module; in another, GoDDSSy delivers a list of IP-Addresses to mitigate, serving a mitigation request from another module. The other interface (b) is an interconnection between other nodes with the same system, where they exchange mitigation requests if one node is the target of such a DDoS attack.

3.2 Architecture

The system is designed in a way that it can be added to an existing web service. The core of GoDDSSy is a cooperative signaling protocol that handles attack and mitigation requests. So that these actions can be performed, it has different interfaces with which it can communicate with other parties on the system. For example, GoDDSSy can be interfaced with network protection systems, such as a network monitoring system, and trigger signaling to other parties in a cooperative defense.

One interface concerns the mitigation module. If the node is under attack, the mitigation module notifies GoDDSSy for mitigation, and if another node is under attack, GoDDSSy informs the mitigation module. In this regard, GoDDSSy exposes a RESTful interface that can be used locally so they can communicate with each other. The other interface is for communication between the nodes. This interface uses a pub/sub protocol to notify

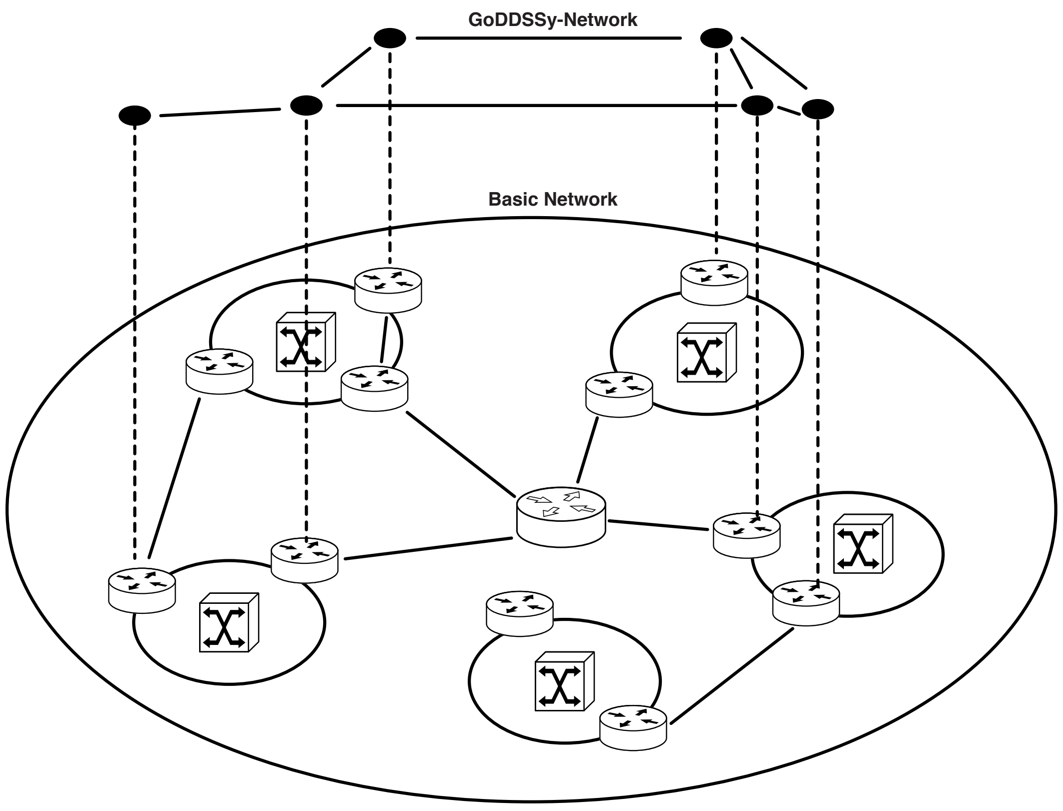

Figure 3.1: The GoDDSSy network as an overlay.

and listen to other peers. The system's core uses a state pattern so communication can occur systematically.

3.2.1 Components

GoDDSSy is a system consisting of different components (Figure 3.2), each of which has its responsibility. They work in synergy to form a signaling system. The different components communicate with one another via various APIs.

3.2.1.1 REST API

The RESTful API is the interface that exposes services to the outside. Several different routes are exposed, so that information from the mitigation module to the signaling system can be sent or that the mitigation module or even a system administrator can read information from GoDDSSy. RESTful information exchange is accomplished through HTTP POST and GET requests. JSON is the data exchange format that is used. JSON has the advantage of being faster and using fewer resources than other formats, such as XML.

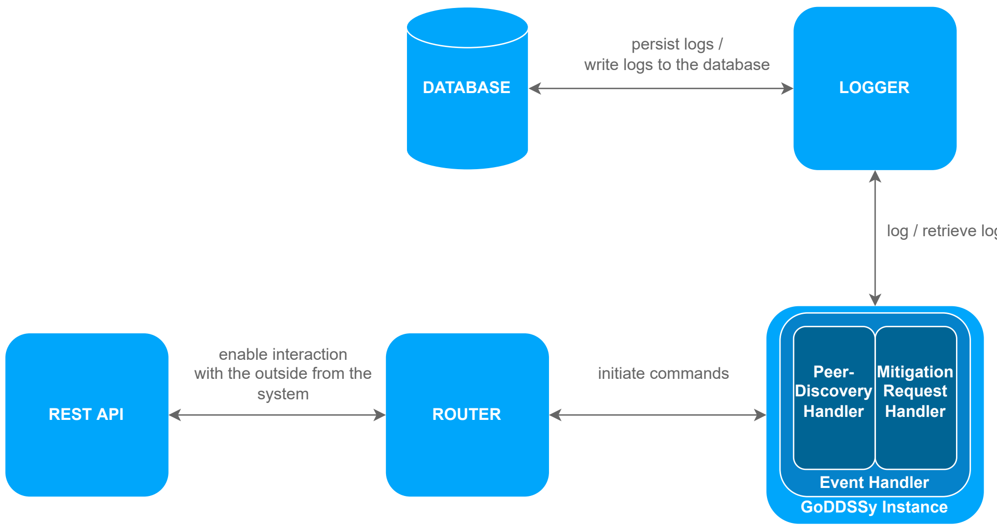
DATABASE
persist logs / write logs to the database
LOGGER
log / retrieve logs
REST API
enable interaction with the outside from the system
ROUTER
initiate commands
Peer-Discovery Handler
Mitigation Request Handler
Event Handler
GoDDSSy Instance

Figure 3.2: The architecture of GoDDSSy.

3.2.1.2 Router

The router is in charge of mapping REST API calls to various methods, which are forwarded to GoDDSSy system. The router creates a GoDDSSy object and calls the various methods on the object specified via RESTful API.

3.2.1.3 GoDDSSy Instance

The GoDDSSy instance contains an event handler which can be divided in two different sub handlers, namely the peer discovery handler (PDH) and the mitigation request handler (MRH), as shown in Figure 3.2. If GoDDSSy receives a new message, it is analyzed in the event handler, which then decides whose responsibility it is to process this message further. If it is a discovery message, the message is forwarded to the PDH. A peer discovery message contains information about other peers in the network. The PDH then checks this information and decides if it wants to connect with them or not. If it is a mitigation request, it is forwarded to the MRH, which checks the state of GoDDSSy. Depending on the state, the MRH forwards the message to the responsible function, which performs further action. These actions are described in Section 3.4.

3.2.1.4 Logger

The logger is a layer between the database and the GoDDSSy instance. It establishes a connection to the database. To keep track of interactions, a set of parameters is logged each time the system communicates with another node. Additionally, the instance has the ability to get the history from the logger and deliver it to the router, which will then make it available to the RESTful API.

3.3 Communication

The interconnection between the different nodes happens in a pub/sub manner and with p2p technology. The pub/sub pattern is ideal for this application since it is a reliable protocol. In addition, it is relatively fast (*e.g.,* in contrast to a blockchain-based approach [28]) and enables the nodes to interact with each other in real time. The p2p approach is chosen since these nodes should be able to interconnect between different organizations and, therefore, cannot share one infrastructure. For this reason, of all the different pub/-sub technologies considered for this thesis, the libp2p GossipSub implementation best fits this purpose.

The mitigation of DDoS attacks is most effective close to the attacker, i.e., close to the source of attacking traffic. Therefore, GoDDSSy makes use of a topic-based pub/sub protocol. Each node can subscribe to one or many regions. If one of the nodes is under attack, it can publish a message under the specific region (mapped as a pub/sub topic) in which the attack is happening. There exist multiple possibilities of different regions that can be regarded (a) in a national manner, as an example; for Switzerland, the different cantons could be seen as regions; or (b) in an international setting, the different continents or even countries can be split into different topics.

3.4 GoDDSSy Protocol

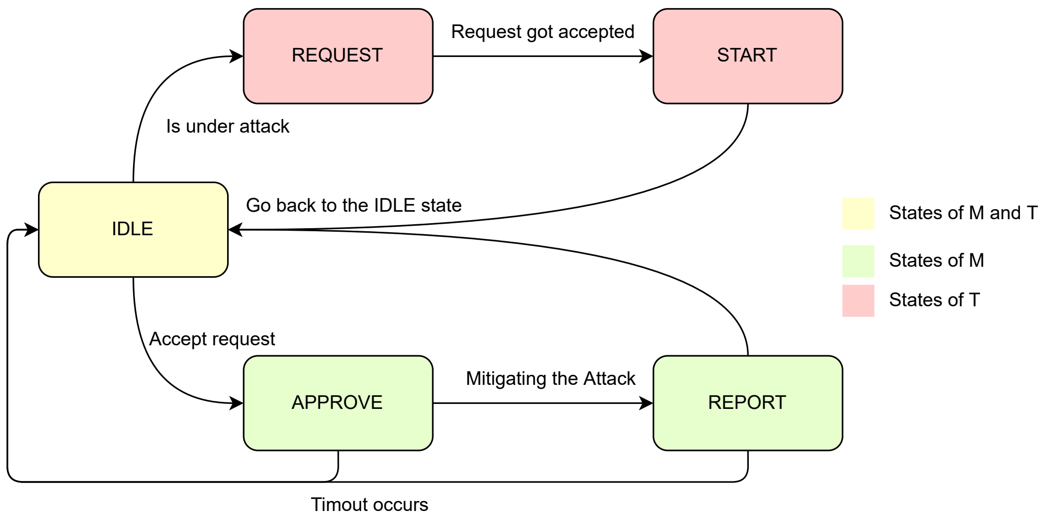

Figure 3.3: The states of the GoDDSSy protocol.

When an attack is happening, GoDDSSy makes use of the signaling protocol. The protocol procedure can be shown in Figure 3.3. In this example, two actors exist the target (T) and the mitigator (M). Both of these actors are interconnected with other nodes in the signaling system. If the Target is under attack, it sends a message to the other nodes in the GoDDSSy network. After this event, T goes into a REQUEST state. Since the system is based on a topic-based pub/sub, T sends out this notification on the topic, to be more precise, the region of the T's location. M has to be subscribed to the topic of the same region to act. Then, M has two possibilities. (a) M can approve the message, or (b) M cannot respond to the request.

T establishes a window of time for M's response. M is not eligible for mitigation if M does not respond promptly, *i.e.,* in a specified time frame defined in the protocol. If M responds quickly, M switches to the APPROVE state, and T sends M a list of the attackers' IPv4 addresses. T's state changes from REQUEST to the START, where T is open for a report once the time available for receiving approvals has expired.

M also receives an IPv4 address list and a time window. This window specifies the time range in which T may receive reports. When the window closes, M has time to complete the mitigation and sends a report to T before changing its state back to the REPORT state. Upon receiving the report, T acknowledges the report. Then, M switches to the initial state IDLE and is ready for another Mitigation. The same holds for T. After the time window for reports ends, T switches to the state IDLE.

Chapter 4

Implementation

4.1 Software Stack

An appropriate software stack is defined to implement the required design from Chapter 3. In section 2.3, various pub/sub frameworks were examined. Their primary distinction is in their architecture, which differs between distributed and decentralized models. Once GoDDSSy is intended to be used as a signaling service between different organizations, a decentralized method is the most suitable choice. Since the libp2p GossipSub implementation is the only decentralized framework available, it is ideal for this project.

The pub/sub protocol is implemented in two ways in libp2p, which provides a flexible network stack. GossipSub is one of them, and FloodSub is another. Compared to FloodSub, GossipSub is more resource efficient due to the more sophisticated routing [32].

As GoDDSSy primarily relies on libp2p's network stack [21], and libp2p has implemented its modules in various programming languages, the best fit is chosen. The goal of libp2p is to have implementations for various programming languages. To date, no language implementations have been completed; however, the stack is specified as useable for Go and JavaScript. The latter option is chosen due to a relatively higher knowledge of JavaScript language.

GoDDSSy exposes a REST interface to interact with the mitigation module. The web-framework express [18] was used for this functionality. For the runtime environment, Node.js [17] was selected as a basis to develop the entire project because it can execute JavaScript without a browser and is suitable for web servers.

Docker [5], a virtualization tool, was used as a basis to ensure portability and straightforward deployment of the project. Docker has various advantages, such as simplifying scaling a container to multiple instances. Furthermore, Docker simplifies the software deployment process.

4.2 Configuration

A configuration file must be passed to the GoDDSSy node before it can be used. These configurations are considered for the signaling protocol and the connection to other nodes.

```
export const config = {
    topics: ["WORLD"],
    connection: {
        peerInfo: undefined,
        relayInfo: {
            id: "12D3KooWKBNWEq4kGPfDSZasihaiGijfoQBqzuAViL1GSV6CA1SF
               ",
            ipv4: "172.18.1.3"
        }
    },
    deadlines: {
        Request: 680,
        Approve: 680,
        Start: 680,
        Report: 680
    }
}
```

Listing 4.1: The configuration of the neutral GoDDSSSy

First, there is a `topic` configuration. The configuration file lists the topics that should be considered because it is a topic-based signaling system. It is possible to list one or more topics. The topic world is selected in listing 4.1. This is for evaluation purposes, as the signaling system is tested with nodes worldwide.

In the connection section, the `peerInfo` can be defined. This is only the case if it is a relay node since they have static ids. This is required since the neutral nodes in the network have to connect on startup to a known relay node. The configuration in 4.1 does not have a `peerInfo` defined since it is the configuration of a neutral node.

The `relayInfo` is the counterpart to the `peerInfo`. This only applies if the node is neutral. It contains information about the relay node's `ipv4` address and `id`. This configuration allows the node to connect to a known relay on startup (bootstrap). Furthermore, deadlines for the signaling protocol must be established as input. A deadline in milliseconds is specified for each step.

4.3 Core Functionality

4.3.1 Configuring the libp2p Node

The libp2p networking stack is built modularly, and each module used in the libp2p node is imported and initialized separately in the configuration.

```
#nodeConfig = {
    addresses: {
      listen: ["/ip4/0.0.0.0/tcp/8080"],
    },
    transports: [new TCP()],
    streamMuxers: [
      new Mplex({
        maxMsgSize: Infinity,
        maxStreamBufferSize: Infinity,
        maxInboundStreams: Infinity,
        maxOutboundStreams: Infinity,
        disconnectThreshold: Infinity,
      }),
    ],
    connectionEncryption: [new Noise()],
    pubsub: new GossipSub({
      allowPublishToZeroPeers: true
    })
```

Listing 4.2: The configuration of the libp2p node

A multi-address is defined to listen to other network peers. The address then is provided to the underlying transport protocol. In this project, the address listens for all incoming IPv4 addresses on port 8080 via the Transmission Control Protocol (TCP). A transport method is defined so the node can listen for new connections. The node connects to other nodes via a TCP connection, as specified in the listen address.

In addition, a stream multiplexer is defined to share a single TCP connection while having multiple streams with other peers. The multiplexer module also determines the maximum message size, maximum inbound and outbound streams, and buffer size. Since we want to test the protocol's limits, the maximum value is set for all values.

An encryption module is included in the node's configuration. In the libp2p framework, the noise encryption module is the only encryption module for JavaScript. By upgrading the transport, the encryption module creates a secure channel.

Finally, the pub/sub module is defined in case the nodes should be able to interact via the pub/sub protocol. This is where the GossipSub module comes in. This configuration allows publishing messages to zero peers, allowing discover messages to be sent even if no other peers are connected.

4.3.2 Initialization

GoDDSSy is initialized with the initialize function upon startup. Since the implementation adheres to the singleton pattern, no multiple GoDDSSy instances exist. After creating a GoDDSSy object, the initialize function is called, which accepts configuration information.

```
if (this.#config.connection.peerInfo) {
      this.#nodeConfig.peerId = await createFromJSON(
        config.connection.peerInfo
      );
```

```
    } else {
        this.#nodeConfig.peerId = await createEd25519PeerId();
    }
```

If a Peer ID is specified in the configuration, the Peer ID with the public and private keys is generated using this information. Otherwise, the node is assigned a random Peer ID.

It is determined whether relay information is specified in the configuration once the node has started. If so, to establish a connection, the node manually dials the relay using its multi-address.

```
if (this.#config.connection.relayInfo) {
        const dialAddress = `/ip4/${config.connection.relayInfo.ipv4}/
           tcp/8080/p2p/${config.connection.relayInfo.id}`;
        await this.#node.dial(dialAddress);
    }
```

After dialing the relay node successfully, the node adds the subscriptions to the pub/sub module. First, topics defined in the configuration files are added. Besides that, the node subscribes to its own Peer ID so that other peers can contact it directly. The node also subscribes to a discover channel, which is required for discovering other peers in the signaling network.

```
this.#config.topics.forEach((topic) => {
        this.#node.pubsub.subscribe(topic);
    });
this.#node.pubsub.subscribe(this.#node.peerId.toString());
this.#node.pubsub.subscribe("discover");
```

The event listener is defined in the final phase of initialization. A message event is triggered whenever a message is sent across GossipSub and received by another node. An event listener is defined to process these events. GoDDSSy registers two event listeners, one for handling signaling protocol messages and the other for handling discovery messages.

```
this.#node.pubsub.addEventListener("message", (evt) => {
        this.connectionHandler(evt);
        this.messageHandler(evt);
    });
```

4.3.3 Peer Discovery

GossipSub handles peer discovery on a local level. However, if the signaling system is deployed to servers worldwide, the peers are not discovered by the libp2p protocol. As a result, a peer-finding method is built into GoDDSSy.

The interval function is used during the relay's initialization to convey information about known peers to the connected nodes. Once nodes connect to the relay nodes upon launch, they can profit from the information they receive. The discovery mechanism is divided into two parts: (a) event emission by the relay node and (b) event handling by the network's neutral nodes.

In a given interval, the `sendPeerInformation` is called. This function's task is to gather all of the multi-adresses of the nodes to which the relay is connected. They are designed so that peers receiving this information can connect with them. The structured data is then distributed to the peers via the specified topic, "discover".

```
sendPeerInformation() {
  const conMultiAddr = this.getConnectedMultiAddr();
  const adresses = conMultiAddr.map((a) => {
    return { peerId: a.split("/")[6], multiAddr: a };
  });
  this.sendMessage("discover", { adresses }, true);
}
```

When the "discover" event occurs, the counterpart to the emitting function registered in the message event handler is executed. After parsing the relay's received messages, the node compiles a list of all connected peers. Following that, the node goes through the list of received addresses and determines whether the node is already connected to the node or if the address is the node's own. If this is not the case, the node dials the address in order to connect.

```
async connectionHandler(evt) {
  if (evt.detail.topic != "discover") {
    return;
  }
  const adresses = this.parseMessage(evt).adresses;
  const connectedPeerIds = this.#node
    .getConnections()
    .map((c) => c.remotePeer);
  for (let address of adresses) {
    if (
      !connectedPeerIds.includes(address.peerId) &&
      address.peerId != this.#node.peerId.toString()
    ) {
      this.dialMultiAddr(address.multiAddr);
    }
  }
}
```

4.3.4 The GoDDSSy Protocol

The core of GoDDSSy is revealed in this section. The protocol described in Section 3.4 has been implemented. As previously stated, it follows a state pattern and is triggered by a message sent by the mitigation module via the REST API.

Following the call to the REST API, the function for sending a mitigation request is invoked. A list of IPv4 addresses as well as a topic and a message is specified in the call. The addresses are then stored in the object, and its state is changed to that of a request. For evaluation purposes, a durations object is initialized to store the various mitigators and their response durations. The deadlines for switching states and the time frame for carrying out the mitigation are defined. All time limits are defined by the configuration described in section 4.2. Following all of these steps, the function responsible for sending messages via the pub/sub protocol is called.

```
sendMitigationRequest(topic, msg, addresses) {
  this.#addressesToBlock = addresses;

  this.#state = states.Request;

  this.#durations = {
    amountOfAddresses: addresses.length,
    nodes: {},
  };
  this.#durations.RequestStateStartTime = this.getMillis();

  this.defineDeadlines();

  this.#timeWindow = {
    start: this.getMillis() + this.#config.deadlines.Request,
    end:
      this.getMillis() +
      this.#config.deadlines.Request +
      this.#config.deadlines.Start,
  };
  this.sendMessage(topic, { message: msg });
}
```

After this initial request, the protocol is initiated, and whenever a message is received by any node, this message handler is invoked. First, the message's content is analyzed to ensure that it is a topic of interest to the node. Following that, the message content is parsed so that it can be analyzed in greater depth. The responsible function is then called based on the state of the object.

```
async messageHandler(evt) {

  if (
    !this.#node.pubsub.getTopics().includes(evt.detail.topic) ||
    evt.detail.topic == "discover"
  ) {
    return;
  }

  const message = this.parseMessage(evt);
  const topic = evt.detail.topic;
  const from = evt.detail.from.toString();
  this.logToStatus(
    `received message: "${message.message}" on topic ${evt.detail.
       topic.slice(
      -5
    )} from ${from.slice(-5)}`
  );
  this.logToStatus(`Latency: ${this.getMillis() - message.time}`);

  switch (this.#state) {
    case states.Idle:
      this.handleIdleState(topic, from);
      break;
      case states.Request:
        this.handleRequestState(topic, from, message);
        break;
      case states.Approve:
        this.handleApproveState(topic, from, message);
        break;
      case states.Start:
        this.handleStartState(topic, from, message);
        break;
      case states.Report:
        this.handleReportState(topic, from);
        break;
    }
  }
```

In the IDLE state, where peers remain if they are not mitigating an attack or requesting mitigation, they first ensure that they are not responding to their own requests. The step is saved to the database so that the node's actions can be tracked. The target is then sent an approve message to confirm that the peer is open for mitigation. After these steps are completed, the node transitions from the IDLE state to the Approve state. A deadline is set to prevent the node from being blocked indefinitely if the target does not respond. When the deadline is reached, the node returns to its initial state and becomes idle for further mitigations. The ID of the timeout is stored in a field so that it can be cleared if the target responds in time.

```
handleIdleState(topic, from) {
  if (topic === this.#node.peerId.toString()) {
    return;
  }
  this.logToDatabase(from, topic, this.#state);
  this.sendMessage(from, { message: "Approve" });
  this.#targetID = from;
  this.#state = states.Approve;
  this.#timeout = setTimeout(() => {
    this.#state = states.Idle;
    this.logToStatus("Going back to IDLE");
  }, this.#config.deadlines.Approve);
}
```

The target has been in the request state since the beginning of the mitigation and is waiting for the mitigators to respond. If the mitigators respond correct according to the protocol, the responding node is considered for further action. The durations object stores the time the message was sent, the time it was received, and the delta for evaluation purposes. Following the logging, the `adressInformation` object is created, which contains a message, a list of addresses to be blocked, and the time window that was previously specified when the mitigation request was invoked. This object is then delivered to the potential mitigators.

```
handleRequestState(topic, from, message) {
  if (message.message != "Approve") {
    return;
  }
  this.#durations.nodes[from] = {};
  this.#durations.nodes[from].Approve = {
    sent: message.time,
    received: this.getMillis() - this.#durations.
        RequestStateStartTime,
    delta: this.getMillis() - message.time,
  };
  this.logToDatabase(from, topic, this.#state);
  const addressInformation = {
    message: "addresses",
    addresses: this.#addressesToBlock,
    timeWindow: this.#timeWindow,
  };
  this.sendMessage(from, addressInformation);
}
```

Upon receiving a message in the approved state, it is verified that the message comes from the initial target. The timeout which was specified to return back to the IDLE state if the target does not answer in time is cleared so that it can move on with the mitigation. The duration of the transfer then is calculated to report back to the target. Completing these steps, the mitigator would have time for mitigation. So that the node can answer in the time window specified by the target, the node waits until the time window is open. Then the mitigator reports back to the target.

```
handleApproveState(topic, from, message) {
  if (from != this.#targetID) {
    return;
  }
  clearTimeout(this.#timeout);
  this.#ipTransferDuration = this.getMillis() - message.time;
  this.logToDatabase(from, topic, this.#state);
  const adressesToBlock = message.addresses;
  this.logToStatus(
    `Received ${adressesToBlock.length} IPv4 adresses to block`
  );
  // Mitigation would be done at this place
  setTimeout(() => {
    this.sendMessage(from, {
      message: "Mitigation is done",
      ipTransferDuration: this.#ipTransferDuration,
    });
    this.#state = states.Report;
    this.#timeout = setTimeout(() => {
      this.#state = states.Idle;
    }, this.#config.deadlines.Report);
  }, message.timeWindow.start - this.getMillis());
}
```

The target waits for incoming confirmations during the open time window and only considers peers who were involved in previous steps. The duration and the state are recorded once more. To provide confirmation, an acknowledgment message is sent back to the mitigators, completing the process. If the timeout specified at the start of the mitigation request expires, the node returns to the IDLE state. In this state, the node is ready to mitigate and assist other peers if necessary.

```
handleStartState(topic, from, message) {
  if (!this.#durations.nodes[from]) {
    return;
  }
  this.#durations.nodes[from].Start = {
    sent: message.time,
    received: this.getMillis() - this.#durations.
       RequestStateStartTime,
    delta: this.getMillis() - message.time,
    timeWindow: this.#timeWindow,
    ipTransferDuration: message.ipTransferDuration,
  };
  this.logToDatabase(from, topic, this.#state);
  this.sendMessage(from, { message: "Ack" });
}
```

In the last state which concerns the mitigator in this situation, the timeout is cleared again. The step is then logged to the database. Similar to the target, the mitigator returns back to the IDLE State where the node is open for further mitigations.

```
handleReportState(topic, from) {
  clearTimeout(this.#timeout);
  this.logToDatabase(from, topic, this.#state);
  this.#state = states.Idle;
}
```

4.4 Local Testing

The OS-level virtualization tool docker was used to test the interaction between the peers. The docker-compose tool is used to run multiple instances.

This is required in order to provide the application with different configuration files. Docker uses volumes to solve this issue. The volumes defined in the docker-compose file are shared by the machine's storage and the running docker container. Listing 4.3 shows that the `goddssy-node` and `goddssy-relay` have different configuration files mounted to the container's configuration folder.

Specific ports are defined in order to have access to the container from the outside of GoDDSSy. As a result, port 3000 is open for access to the REST API.

In addition, networks must be configured in the docker-compose.yml file. If no network is configured, the containers run in isolation and cannot communicate with one another. In this scenario, the network, referred to as `net`, solves the problem of isolation. As can be seen, the relay has a static IPv4 address, allowing other nodes to dial the known peer.

Aside from the ease of starting the various nodes, the docker-compose setting has another benefit for testing. It is possible to choose to scale up an existing service during the startup process. As a result, to have five instances of the GoDDSSy node, one can simply run `docker-compose up -d -scale goddssy-node=5`.

```yaml
version: "3.3"
services:
  goddssy-relay:
    image: tobifra/goddssy
    volumes:
      - ./multiconfig/relay:/usr/src/app/config
      - ./connections:/usr/src/app/connections
    networks:
      net:
        ipv4_address: 172.18.1.3

  goddssy-node:
    image: tobifra/goddssy
    volumes:
      - ./multiconfig/node:/usr/src/app/config
      - ./connections:/usr/src/app/connections
    ports:
      - 3000
    depends_on:
      - goddssy-relay
    networks:
      - net

networks:
  net:
    driver: bridge
    ipam:
      config:
        - subnet: 172.18.1.0/24
```

Listing 4.3: The docker-compose.yml file, used for local testing

Chapter 5

Evaluation

5.1 Evaluation Setup

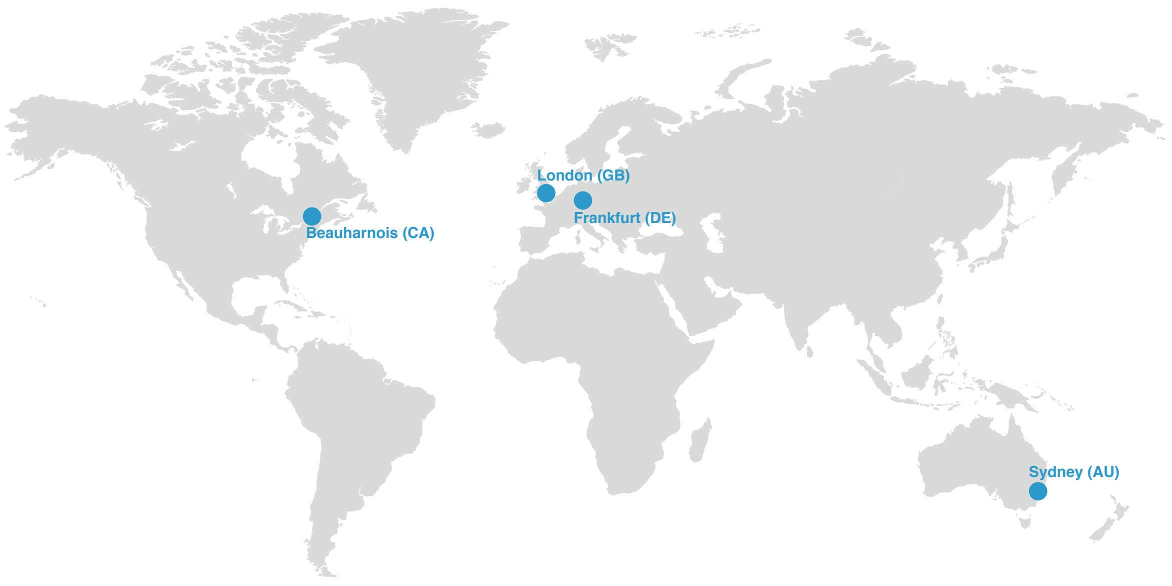

Figure 5.1: GoDDSSy deployment locations on the world map.

In practice, GoDDSSy is integrated into an existing web service. Four GoDDSSy instances are deployed around the world to simulate a real-world scenario. As a result, four Virtual Private Server (VPS) instances are deployed in Beauharnois, London, Frankfurt, and Sydney. Each VPS comes equipped with a one-core processor, 1 gigabyte of memory, 10 gigabytes of SSD storage, and a 100 megabit-per-second bandwidth. Debian 11 is the operating system that the VPS is using.

Since a Docker setup is already in place for local testing, it aids the deployment. Separate docker-compose configuration files are created, similar to those shown in listing 4.3. The local configuration differs from the deployment configuration in that the latter only contains the configuration for one node. In addition, another port is exposed for interaction over GossipSub. A Python script is used to connect to each server and deploy the software. Secure Shell (SSH) and the Secure Copy Protocol (SCP) are used to transfer the files and restart the containers.

5.2 Scenarios

Transfer Limit of attack information. We want to see how many IPv4 addresses can be sent over GoDDSSy in the first setup. We use the deployed GoDDSSy network to test this. The target is selected, and a list of addresses is sent via the RESTful interface. The list of IP addresses has a fixed length and is then linearly increased. Following each mitigation request, the durations of nodes are collected, which allows to determine which node responded. This script is run until none of the nodes respond.

Transmission Time Versus Amount of Attack Information. In the next scenario, we want to determine if the transmission time is affected by the amount of attack information being exchanged. To do this, a target is selected, and several mitigation requests are sent to the one node using an automated script. The list of addresses starts with a specific length. The log is played 100 times for each length to the point where the system stops working. The transmission times are then compared with the number of addresses.

Determination of the Optimal Time Constraint per Step. In this scenario, we want to figure out how to adjust the deadlines for each step in the signaling protocol so that the response time is as short as possible while allowing as many mitigators as possible to respond to the attack. Therefore, the geographical distance between each node must be considered. To test this, the deployed setup is used, and the time delta from sending a mitigation request to receiving approval for all nodes is measured to determine the optimal time constraint per step.

Disruption by Flooding with Messages. The system is tested to see whether it can withstand a peer who floods the network with messages. Therefore, the deployed setup is used in two scenarios. In one case, a peer receives 100 mitigation requests with no disruption; in the other, the same number of mitigation requests are sent, but one peer floods the network with spam on an unrelated topic. The time difference between sending a mitigation request and receiving approval is measured and compared in both cases.

5.3 Results

5.3.1 Transfer Limit of Attack Information

In this scenario, the address list is increased by 1000 addresses in each step, beginning with 270’000 and ending with 285’000 addresses. Table 5.1 shows that the GoDDSSy protocol works with 279’000 addresses and completes all steps. If 1’000 addresses expand the address list, the other nodes no longer respond to attack information sent to them. The other nodes do not respond with an approved message at 280'000 addresses. Messages of 4'359 kilobytes are the largest that can be sent over GossipSub, even though the stream buffer size and maximum message size are set to infinity in the node configuration, as shown in section 4.3.1. If the message size is increased afterward, the target node loses connection.

		Server Location					
Addresses	**KB**	**United Kingdom**		**Canada**		**Australia**	
		Approval	**Report**	**Approval**	**Report**	**Approval**	**Report**
279'000	**4359**	✓	✓	✓	✓	✓	✓
280'000	**4375**	✓	✗	✓	✗	✓	✗
281'000	**4391**	✗	✗	✗	✗	✗	✗

Table 5.1: The transfer limits of attack information.

5.3.2 Transmission Time Versus Amount of Attack Information

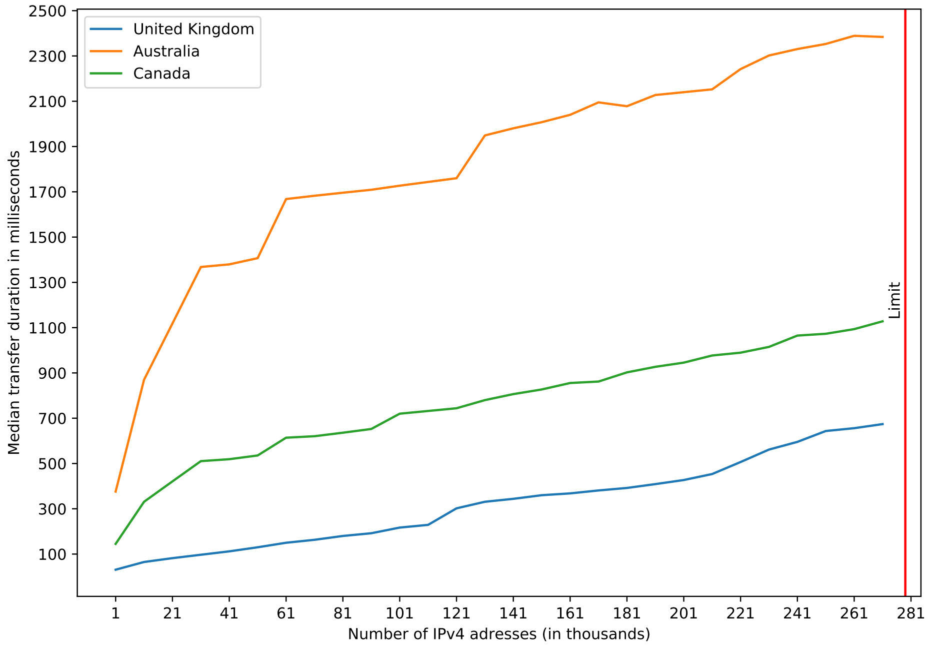

Figure 5.2: The transfer duration compared to the number of IPv4 addresses per node.

To give each GoDDSSy node enough time to respond, the deadline for each step is set to 5000 milliseconds. Following that, a list of 1000 IPv4 addresses is sent to the node in Germany, which requests mitigation from the other nodes. The same request is repeated 100 times to eliminate outliers, and 10'000 addresses increase the address list after repetitions. This is carried out up to the determined maximum in the first scenario. Times from all

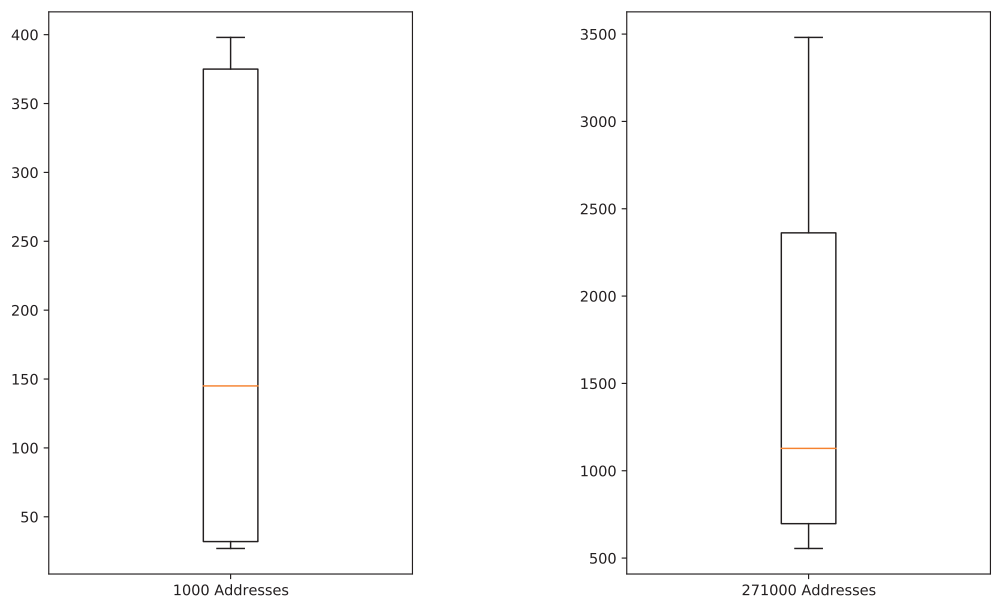

Figure 5.3: The comparison of the transfer time of 1'000 versus 271'000 addresses for all the nodes.

	Server Location					
N	Australia		Canada		United Kingdom	
	Mean Duration [ms]	Standard Deviation [ms]	Mean Duration [ms]	Standard Deviation [ms]	Mean Duration [ms]	Standard Deviation [ms]
1'000	377.15	3.98	146.25	5.09	31.69	4.35
11'000	876.16	14.27	331.69	4.27	67.93	9.65
21'000	1125.36	17.51	422.28	5.89	84.68	8.5
31'000	1380.8	26.62	511.7	17.88	101.79	17.63
41'000	1392.06	27.5	520.86	9.08	114.56	12.53
51'000	1418.24	46.79	539.62	15.04	132.73	13.22
61'000	1658.77	31.87	614.63	11.92	160.38	28.05
71'000	1678.37	35.77	621.75	12.25	166.77	19.99
81'000	1694.9	35.34	641.42	21.96	183.83	14.84
91'000	1709.81	38.01	659.93	25.55	194.87	10.32
101'000	1728.18	30.83	721.94	41.53	218.54	11.06
111'000	1745.83	38.2	736.19	47.95	236.21	21.25
121'000	1768.15	57.79	749.64	28.59	289.48	39.45
131'000	1949.87	39.7	787.25	32.72	334.56	22.35
141'000	1978.89	44.1	812.83	31.56	350.51	29.29
151'000	2018.9	61.68	830.3	32.36	361.37	19.04
161'000	2046.9	60.35	864.34	42.4	371.58	22.66
171'000	2085.48	54.84	876.22	38.63	385.15	21.94
181'000	2089.59	61.76	909.67	68	399.95	31.62
191'000	2124.37	60.62	948.28	119.39	414.11	23.95
201'000	2143.81	56.94	967.36	136.9	431.05	21.38
211'000	2164.44	97.56	992.8	75.13	462.98	35.48
221'000	2297.23	255.21	1018.47	134.24	511.47	49.3
231'000	2373.47	259.53	1040.13	111.24	553.68	61.42
241'000	2358.77	136.08	1078.31	55.62	595.52	60.28
251'000	2387.05	109.6	1080.96	45.5	631.54	53.83
261'000	2407.61	96.41	1102.98	51.34	653.24	58.25
271'000	2405.91	72.59	1181.99	263.09	664.13	48.69

Table 5.2: The means and standard deviations of the scenario in Section 5.3.1.

nodes are retrieved and collected after each request. For the evaluation, the median IP transfer time of each 100 attempts from each node for each address list length is taken.

In Figure 5.3, the transmission times are plotted against the length of the address list. It can be seen that with a shorter address list, the transmission time is significantly shorter than with a long address list. One can also see that the transmission time depends very much on the geographical distance.

5.3.3 Determination of the Optimal Time Constraint per Step

The measurements were taken from the scenario in section 5.3.2, which measures 2'800 signaling requests. Each step measures the time between sending a request and receiving approval. The findings reveal that the node in the UK responds in a median time of 75 milliseconds, the node in Canada in a median time of 300 milliseconds, and the node in Australia in a median time of 778 milliseconds. However, nodes are not always time consistent. The node in Australia, for example, has a maximum response time of 970 milliseconds.

When configuring GoDDSSy, one must consider whether the protocol should run as quickly as possible or allow as many nodes to respond. Because the topic in this scenario was the entire world, the nodes were sometimes geographically far apart. The high response time would not be an issue if one chose a regional topic.

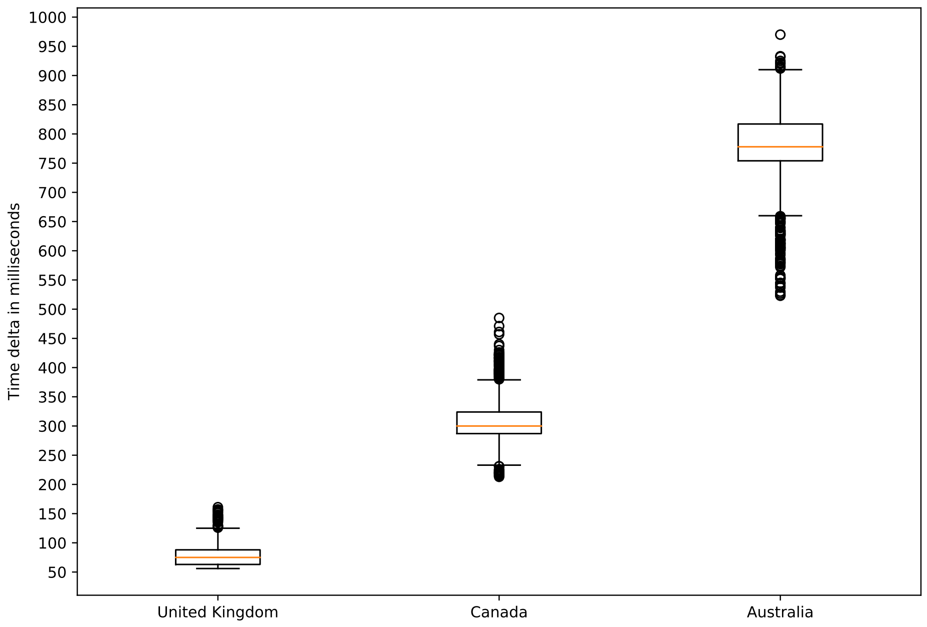

Figure 5.4: Time delta between sending the mitigation request and receiving approval.

	Mean [ms]	**Standard Deviation [ms]**
Australia	780.42	47.59
Canada	307.85	28.94
United Kingdom	76.82	15.76

Table 5.3: The means and standard deviations of the scenario in Section 5.3.3.

5.3.4 Disruption by Flooding with Messages

To give enough time, the deadlines in the first scenario are set to 2'000 milliseconds for each step, and 10'000 addresses are sent in each request to the GoDDSSy node in Germany. After each request, the durations of nodes are collected, and the time difference between sending a mitigation request and receiving approval is determined. The same was done in the second scenario. Still, during mitigations, another node sends a mitigation request every ten milliseconds on a topic to which one of the other nodes is subscribed, so the GossipSub is flooded with messages. Figure 5.5 compares both times. As can be seen, there is no significant difference in measurement time between the scenarios with and without spam.

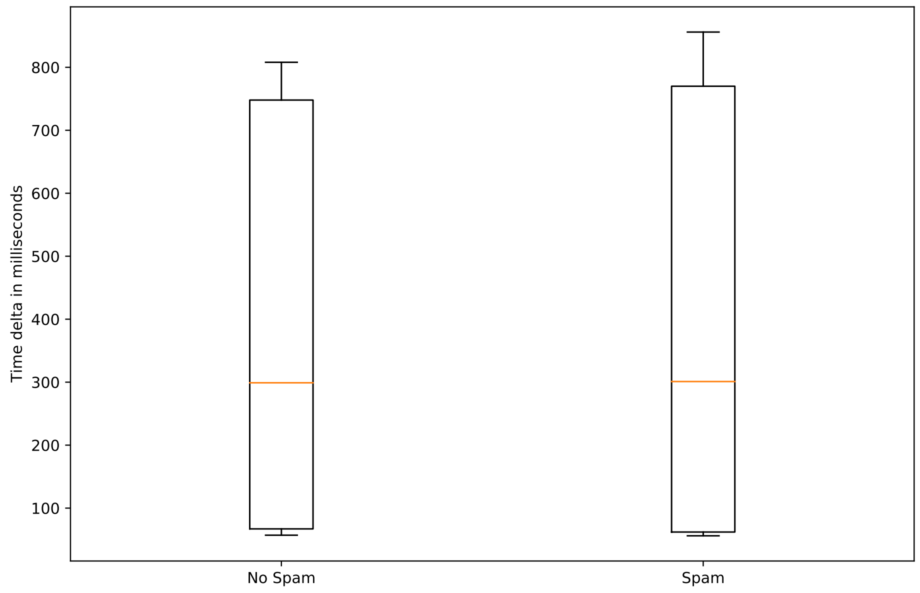

Figure 5.5: Comparison of the answering time in the case with and without spam.

	Mean [ms]	**Standard Deviation [ms]**
With Spam	419.47	356.87
Without Spam	375.31	290.02

Table 5.4: The means and standard deviations of the scenario in Section 5.3.4.

5.4 Discussion

The evaluation examined four different scenarios. The first scenario addresses the overall system's performance. The speed of GoDDSSy is measured concerning the data size in the second scenario. This experiment demonstrates a significant difference in transfer speed between small and large data loads. In the third scenario, the protocol's speed is investigated when no large amounts of data are exchanged. The round trip times of the approved request are compared for all nodes and revealed substantial differences due to geographical distances. All the scenarios discussed above deal with protocol deadlines. The final scenario addresses the protocol's resistance to attacks. The node response times are compared in a scenario with and without flooding of the GossipSub. The results show that there is no significant difference between the two scenarios.

GoDDSSy has a fixed limit of 279'000 addresses that can be exchanged, which is only 4359 kilobytes of data. This can be limiting if an attack occurs with multiple attackers exceeding this limit. Using the Mirai botnet as an example, which has up to 380'000 devices [14], the system would be overwhelmed. This data-exchange limit is reached because the libp2p framework does not support larger payloads, and communication is done via the pub/sub framework. To overcome this limitation, the design could be altered to transfer attack information via a more appropriate channel for larger payloads rather than directly over GossipSub. BloSS [28] for example solves this problem by exchanging a link by which the mitigator can retrieve attack information.

Setting deadlines is a tradeoff, as demonstrated in scenarios in the section 5.3.2 and 5.3.3. On the one hand, the system should allow for quick mitigations; on the other hand, nodes should be able to participate in signaling. In each scenario described above, the node in Australia has a significantly longer response time than the node in the United Kingdom. Furthermore, because the deadlines are highly dependent on the data loads exchanged by the signaling system, no general statement can be made. However, in this configuration, a decentralized system with nodes distributed across large distances is evaluated. The deadlines could be set shorter if the topics were narrowed to a more specific region, such as CH instead of WORLD, as mentioned in section 3.3. This would include both the time it takes to transmit attack information and the time it takes to approve a response. Nonetheless, the amount of attack information that can be exchanged over GoDDSSy is limited, so the deadlines could always allow the maximum transmission time to exchange the limit of 279'000 addresses.

The final scenario demonstrates GoDDSSy's resistance to message flooding. Even if a peer sends many messages at a high frequency, the system can handle it. However, sending a high volume of attack messages to a topic to which other peers are subscribed is still possible. Peers would then attempt to mitigate the attack, but would be prevented from doing other mitigations.

Chapter 6

Final Considerations

6.1 Summary

The thesis proposes a signaling protocol that enables actors in a trusted environment to exchange attack information. A description of agile and resilient is given to establish quantifiable parameters and assess the suggested signaling protocol. In the end, the proposed system is evaluated according to the description.

To reach the goals, a closer look at DDoS attacks is given. It is distinguished between application and network/transport layer attacks, and insights are given into how DDoS attacks occur today. Also it is explained how DDoS attacks are detected and where the challenge lies in this task. In addition, an explanation of how a pub/sub system works, which can be used to exchange information in a subject-based manner, is provided.

A review of related work is also given, outlining the signaling strategies used by the cooperative defense systems Cossack and CoFence. A closer examination of BloSS and its protocol, which outlines the various phases between the mitigator and the target in cooperative DDoS signaling, is conducted. Besides that, an overview of different pub/sub tools is given. Kafka, Emitter, RabbitMQ, and GossipSub are explained and compared.

In addition, an overview of the design of GoDDSSy is provided, and the use of the system is explained. The architecture is described in more detail, including information about the RESTful API, the router, the database, and the GoDDSSy instance. This thesis also describes how the protocol operates if a system actor is attacked.

A prototype of the designed system is then implemented using the libp2p network stack and Node.js to have a server-side JavaScript application. The software is then packed inside docker containers to simplify the deployment process and was then tested locally.

Four GoDDSSy instances are deployed to servers in the United Kingdom, Germany, Canada, and Australia for the evaluation process. The system is then evaluated in four scenarios to test GoDDSSy for resiliency and agility. The limit of the amount of attack information that can be transferred is evaluated, and the transfer limit lies at 279'000 addresses. In addition, the transfer time of attack information is measured regarding the size of the information, and it is found that the duration is dependent on the size. Furthermore, it evaluates the optimal time constraint per step of the mitigation protocol. It is found that it strongly depends on the actor's geographical location in the system. The final scenario tests if the system can withstand if a node floods GoDDSSy with spam, and measurements reveal that GoDDSSy could withstand an attack like this.

6.2 Conclusions

The goal of this thesis is to design and build a cooperative signaling system that allows multiple actors in a trusted environment to exchange attack information while remaining agile and resilient. These requirements are met. The various scenarios tested in the Evaluation (Chapter 5) provide evidence for this.

To make claims about GoDDSSy's agility and resiliency, it must be compared to another signaling system. BloSS provides a foundation for this. In the best-case scenario, the system built in the scope of this thesis can exchange up to 279'000 IPv4 addresses. This includes all in all with additional required information 4.375 Megabytes. Compared to BloSS, which uses Ethereum blocks with an average size of about 0.093 MB [7], this is comparatively much. In terms of the transmission time of attack information and the time constraint of each step, the transmission time is as low as a few seconds and the response time is as low as a few hundredths of a second. When compared to BloSS, this is fast because BloSS is based on Ethereum and a new block is generated every 15 seconds.

The requirement of resilience is met because GoDDSSy can withstand the flood of messages from malicious peers. The system is still operational, and the response times are not significantly influenced, as can be seen in Figure 5.5.

However, one of the main limitations of GoDDSSy is that data exchange is still very limited due to limitations in the pub/sub framework that was used. If the attack involves more than 279'000 addresses, GoDDSSy is unable to notify other peers. Furthermore, unlike BloSS, GoDDSSy does not track node reputation and does not penalize malicious actors in the system.

The main challenges were using a network framework that was relatively new and was not widely used. The libp2p network stack is still in development, and many breaking changes continue to occur. The lack of documentation and examples also made it difficult to build a working system, requiring a closer look into the framework's source code.

Another challenge encountered during this thesis was the deployment of the setup after it had been tested locally. Despite the fact that the application had been running inside a Docker container since the beginning, it was still difficult to deploy the application in a continuous and fast manner. The fact that the deployed version of GoDDSSy didn't function the same way as it did on the local setup presented another challenge during the thesis. Finding other peers proved to be difficult for the system. So that it could find other peers, the signaling mechanism had to be modified during the deployment phase.

6.3 Contributions

The following contribution were made throughout this thesis:

- Design and implementation of a system to detect DDoS attacks across multiple nodes.
- Examination of the libp2p network stack and document a libp2p node's setup.
- Description of the libp2p GossipSub implementation's restrictions in detail.
- Containerization of a system which implements the libp2p network stack.

6.4 Future Work

In this thesis, GossipSub was used for establishing the core communication module of GoDDSSy. Since its advantage, compared to other signaling approaches, is that it is fast and has low overhead, it is not the best fit to transmit a lot of data, as seen in the evaluation. For further work, one could still communicate the protocol over GossipSub, but use another method to exchange the attack information. The target could, for example, send a link to the mitigator, enabling it to access the information. This could have a benefit in increasing the limit for data transmission. As of now, GoDDSSy does not track each node's reputation to measure how credible each actor in the system is. In further work, a reputation tracking system could be developed to punish malicious nodes or exclude them from the cooperative signaling system if they send false attack alarms. This could contribute to the stability of the system.

Bibliography

[1] *Apache Kafka Documentation.* URL: `https://kafka.apache.org/documentation/`.

[2] Mitko Bogdanoski, Tomislav Suminoski, and Aleksandar Risteski. “Analysis of the SYN flood DoS attack”. In: *International Journal of Computer Network and Information Security (IJCNIS)* 5.8 (2013), pp. 1–11.

[3] Glenn Carl et al. “Denial-of-service attack-detection techniques”. In: *IEEE Internet computing* 10.1 (2006), pp. 82–89.

[4] Paul J Criscuolo. *Distributed denial of service: Trin00, tribe flood network, tribe flood network 2000, and stacheldraht ciac-2319.* Tech. rep. California Univ Livermore Radiation Lab, 2000.

[5] Docker. *Develop faster. Run anywhere.* Oct. 2022. URL: `https://www.docker.com/`.

[6] *Emitter: Publish/subscribe.* URL: `https://emitter.io/develop/`.

[7] Etherscan. *Ethereum Average Block Size Chart.* Nov. 2022. URL: `https://etherscan.io/chart/blocksize`.

[8] Patrick Th Eugster et al. “The many faces of publish/subscribe”. In: *ACM computing surveys (CSUR)* 35.2 (2003), pp. 114–131.

[9] Laura Feinstein et al. “Statistical approaches to DDoS attack detection and response”. In: *Proceedings DARPA Information Survivability Conference and Exposition* 1 (2003), 303–314 vol.1.

[10] Andreas Gruhler, Bruno Bastos Rodrigues, and Burkhard Stiller. “A Reputation Scheme for a Blockchain-based Network Cooperative Defense”. In: *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)* (2019), pp. 71–79.

[11] Alexander Gutnikov, Oleg Kupreev, and Yaroslav Shmelev. *DDoS attacks in Q2 2022.* 2022. URL: `https://securelist.com/ddos-attacks-in-q2-2022/107025/`.

[12] Yongqiang Huang and Hector Garcia-Molina. “Publish/subscribe in a mobile environment”. In: *Wireless Networks* 10.6 (2004), pp. 643–652.

[13] Constantinos Kolias et al. “DDoS in the IoT: Mirai and other botnets”. In: *Computer* 50.7 (2017), pp. 80–84.

[14] Brian Krebs. *Source code for IOT botnet ’mirai’ released.* 2016. URL: `https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/`.

[15] *LibP2P: Publish/subscribe.* URL: `https://docs.libp2p.io/concepts/publish-subscribe/`.

[16] Kaname Nishizuka et al. *Inter-organization cooperative DDoS protection mechanism.* Internet-Draft draft-nishizuka-dots-inter-domain-mechanism-02. Internet Engineering Task Force, Dec. 2016. 27 pp.

[17] OpenJS Foundation. *Run JavaScript Everywhere.* Oct. 2022. URL: `https://nodejs.dev/`.

[18] OpenJS Foundation. *Fast, unopinionated, minimalist web framework for Node.js.* Oct. 2022. URL: `https://expressjs.com/`.

[19] Opeyemi Osanaiye, Kim-Kwang Raymond Choo, and Mqhele Dlodlo. "Distributed denial of service (DDoS) resilience in cloud: Review and conceptual cloud DDoS mitigation framework". In: *Journal of Network and Computer Applications* 67 (2016), pp. 147–165.

[20] C. Papadopoulos et al. "Cossack: coordinated suppression of simultaneous attacks". In: *Proceedings DARPA Information Survivability Conference and Exposition.* Vol. 1. 2003, 2–13 vol.1.

[21] Protocol Labs. *A modular network stack.* Oct. 2022. URL: `https://libp2p.io/`.

[22] Bahman Rashidi and Carol Fung. "CoFence: A collaborative DDoS defence using network function virtualization". In: *2016 12th International Conference on Network and Service Management (CNSM).* 2016, pp. 160–166.

[23] Bruno Rodrigues, Thomas Bocek, and Burkhard Stiller. "Enabling a Cooperative, Multi-domain DDoS Defense by a Blockchain Signaling System (BloSS)". In: *Demonstration Track.* 2017, pp. 1–3.

[24] Bruno Rodrigues, Thomas Bocek, and Burkhard Stiller. "The use of blockchains: Application-driven analysis of applicability". In: *Advances in Computers.* Vol. 111. 2018, pp. 163–198.

[25] Bruno Rodrigues and Burkhard Stiller. "Cooperative signaling of DDoS attacks in a blockchain-based network". In: *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos.* 2019, pp. 39–41.

[26] Bruno Rodrigues and Burkhard Stiller. "The Cooperative DDoS Signaling based on a Blockchain-based System". In: *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM).* IEEE. 2021, pp. 760–765.

[27] Bruno Rodrigues et al. "A blockchain-based architecture for collaborative DDoS mitigation with smart contracts". In: *IFIP International Conference on Autonomous Infrastructure, Management and Security.* Springer, Cham. 2017, pp. 16–29.

[28] Bruno Rodrigues et al. "Blockchain signaling system (bloss): Cooperative signaling of distributed denial-of-service attacks". In: *Journal of Network and Systems Management* 28.4 (2020), pp. 953–989.

[29] Bruno Rodrigues et al. "Evaluating a blockchain-based cooperative defense". In: *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM).* IEEE. 2019, pp. 533–538.

[30] Bruno Rodrigues et al. "SC-FLARE: Cooperative DDoS signaling based on smart contracts". In: *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC).* IEEE. 2020, pp. 1–3.

[31] Bruno Bastos Rodrigues, Thomas M. Bocek, and Burkhard Stiller. "Enabling a Cooperative, Multi-domain DDoS Defense by a Blockchain Signaling System (BloSS)". In: *LCN 2017.* 2017.

[32] Dimitris Vyzovitis and Yiannis Psaras. "Gossipsub: A secure pubsub protocol for unstructured, decentralised p2p overlays". In: (2019).

[33] Dimitris Vyzovitis et al. "GossipSub: Attack-Resilient Message Propagation in the Filecoin and ETH2.0 Networks". In: *CoRR* abs/2007.02754 (2020).

[34] Haining Wang, Danlu Zhang, and Kang G Shin. “Change-point monitoring for the detection of DoS attacks”. In: *IEEE Transactions on dependable and secure computing* 1.4 (2004), pp. 193–208.

[35] Saman Taghavi Zargar, James Joshi, and David Tipper. “A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks”. In: *IEEE Communications Surveys Tutorials* 15.4 (2013), pp. 2046–2069.

Abbreviations

DoS	Denial of Service
DDoS	Distributed Denial of Service
EB	Event Broker
BloSS	Blockchain Singnaling System
pub/sub	Publish and Subscribe
MRH	Mitigation Request Handler
EM	Event Manager
GoDDSSy	GossipSub DDoS Signaling System
TCP	Transmission Control Protocol
VPS	Virtual Private Server
SSH	Secure Shell
SCP	Secure Copy Protocol
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
PDH	Peer Discovery Handler

List of Figures

2.1 DDoS attack carried out by a botnet of compromised devices. 4

3.1 The GoDDSSy network as an overlay. 10

3.2 The architecture of GoDDSSy. 11

3.3 The states of the GoDDSSy protocol. 12

5.1 GoDDSSy deployment locations on the world map. 25

5.2 The transfer duration compared to the number of IPv4 addresses per node. 27

5.3 The comparison of the transfer time of 1'000 versus 271'000 addresses for all the nodes. 28

5.4 Time delta between sending the mitigation request and receiving approval. 29

5.5 Comparison of the answering time in the case with and without spam. . . . 30

List of Tables

2.1 A comparison of different pub/sub libraries. 7

5.1 The transfer limits of attack information. 27

5.2 The means and standard deviations of the scenario in Section 5.3.1. 28

5.3 The means and standard deviations of the scenario in Section 5.3.3. 30

5.4 The means and standard deviations of the scenario in Section 5.3.4. 30

Listings

4.1 The configuration of the neutral GoDDSSSy 16
4.2 The configuration of the libp2p node 17
4.3 The docker-compose.yml file, used for local testing 24