



University of  
Zurich<sup>UZH</sup>

# Cooperative Signaling Protocol

*Dominik Bünzli*  
*Zürich, Switzerland*  
*Student ID: 14-707-111*

Supervisor: Bruno Rodrigues, Eder John Scheid  
Date of Submission: February 15, 2019



# Zusammenfassung

Distributed Denial of Service (DDoS) Attacken stellen immer noch eine sehr grosse Gefahr für die Verfügbarkeit von Diensten in der vernetzten Welt dar. Da immer mehr Services von diesen Diensten abhängen, kann ein solcher Angriff weitreichende Folgen haben. Meist reicht jedoch die Kapazität eines einzelnen Systems nicht mehr aus für die Abwehr koordinierter Attacken vieler einzelner Clients, weshalb die kooperative Abwehr solchen Angriffe immer wichtiger wird. Da man sich jedoch in einem kompetitiven Umfeld nicht auf die Aufrichtigkeit einzelner Akteure verlassen kann, die immer im Sinne des ganzen und nicht eigennützig handeln (Trittbrettfahrer), ist ein mit Anreizen ausgestattets Protokoll fast unumgänglich. Hierfür bietet sich die Blockchain im Zusammenspiel mit Smart Contracts an die durch ihre Eigenschaft, nur Änderungen und neue Einträge anfügen, jedoch niemals existierende löschen zu können, eine Vertrauensbasis schafft. Diese Arbeit soll die Basis des von [1] und [2] vorgestellten Cooperative Signaling Protocols implementieren. Hierfür soll der Kontrakt mit der Programmiersprache Solidity geschrieben und auf eine Ethereum Blockchain deployt werden. Das schlussendliche Ziel ist es, die Interaktion zweier Akteure in einem solchen Vertrag aufzuzeigen und auf dessen Effizienz und Performance zu evaluieren.



# Abstract

Distributed Denial of Service (DDoS) attacks still pose a huge threat to the availability of services in the networked world. As more and more applications depend on these services, such an attack can have far-reaching consequences. In most cases, however, the capacity of a single system is no longer sufficient to defend against coordinated attacks by many individual clients, which is why cooperative defence against such attacks is becoming increasingly important. Since, however, in a competitive environment, one cannot rely on the sincerity of individual actors who always act in the interests of the whole and not selfishly (free riders), a protocol equipped with incentives is almost inevitable. In this case the Blockchain offers itself in cooperation with Smart Contracts, which create a basis of trust by their characteristic to only add changes and new entries, but never to be able to delete existing records. This paper is intended to implement the basis of the Cooperative Signaling Protocol presented by [1] and [2]. For this purpose the contract shall be written with the programming language Solidity and deployed on an Ethereum Blockchain. The final goal is to demonstrate the interaction of two actors in such a contract and to evaluate its efficiency and performance.



# Acknowledgments

First of all, I would like to thank Bruno Rodrigues for his valuable support and feedback with which he has contributed significantly to the successful completion of this work. I would also like to thank Professor Dr. Burkhard Stiller, head of the Communication Systems Group at the University of Zurich, for the opportunity to write this highly interesting work at his chair.



# Contents

<b>Zusammenfassung</b>	<b>i</b>
<b>Abstract</b>	<b>iii</b>
<b>Acknowledgments</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Task Description . . . . .	2
1.2 Outline . . . . .	2
<b>2 Basic Concepts</b>	<b>3</b>
2.1 Blockchain . . . . .	3
2.2 Ethereum & Smart Contracts . . . . .	4
2.3 Blockchain Signaling System . . . . .	4
<b>3 Protocol Requirements</b>	<b>7</b>
3.1 Cooperative Signaling Protocol Scheme . . . . .	7
<b>4 Design</b>	<b>9</b>
4.1 First Approach . . . . .	9
4.2 Implementation With State Pattern . . . . .	10
4.3 Refined First Prototype . . . . .	12

<b>5</b>	<b>Implementation</b>	<b>13</b>
5.1	First Prototype - Basic Understanding . . . . .	13
5.2	State Pattern Sounds Very Reasonable . . . . .	14
5.3	Simplicity is Key . . . . .	14
5.4	Surrendering is Not an Option . . . . .	15
5.5	Disillusionment spreads . . . . .	18
<b>6</b>	<b>Evaluation</b>	<b>19</b>
6.1	Testing . . . . .	19
6.2	Performance and Efficiency . . . . .	21
<b>7</b>	<b>Summary and Conclusions</b>	<b>23</b>
	<b>Bibliography</b>	<b>25</b>
	<b>Abbreviations</b>	<b>27</b>
	<b>Glossary</b>	<b>29</b>
	<b>List of Figures</b>	<b>29</b>
	<b>List of Listings</b>	<b>31</b>
<b>A</b>	<b>Contents of the CD</b>	<b>35</b>

# Chapter 1

## Introduction

As mentioned by [1] and [2] in their work, Distributed Denial of Service (DDoS) attacks still pose an unmitigated threat to the availability of the Internet. In combination with a growing number of devices, also given by the rapid growth of Internet Of Things, attackers thus have an ever larger field of potential targets. However, most of the currently available detection or mitigation systems are either in-house or single-domain. In particular, in-house systems have shown that some of them do not have the hard- or software capacity required to fend off major attacks. Since DDoS attacks are more and more often coordinated, a distributed and coordinated defense is required. This way, the load, which normally lies on a single target, can be distributed among many and thus, increasing the chance of successfully repulsing the attack. However, in a competitive environment it cannot simply be trusted that the individual participants will behave for the good of all. Nor can it be expected that everyone voluntarily participates in the effective work and is not just a beneficiary. [2] are trying to tackle this problem with a three-step procedure that they presented in their paper. Firstly, an attacker publishes the harmful IP addresses, secondly these addresses get blocked or filtered by the other participants (called mitigators) through the actual mitigation services and in a third step, the target evaluates the effectiveness. To ensure this process, a reward system for cooperation between participants is required.

At this point, the Blockchain comes into play, as it is not only suitable for signaling against attacks, but also as a trustworthy and distributed platform for reputation management. This work deals with the implementation of the basic contract between a target and a mitigator of the Cooperative Signaling Protocol, which was presented by [1] and [2] in their work and is further described in chapter 3. For the actual implementation, Ethereum will serve as platform and the whole process will be implemented as a Smart Contract.

## 1.1 Task Description

The task is divided into two goals which are described as follows:

- **Design and development of the prototype** This MBM needs to implement the aforementioned exchange of messages as defined in the Cooperative Signaling Protocol [2]. To achieve this, the Ethereum Blockchain has to be used with at least two instances (target and mitigator) interacting with each other.
- **Evaluation** This MBM needs to deliver a short evaluation of the implementation and a indication of performance expectations.
- **Report** Providing a written report on the performed work and the achieved results, which needs to read in a self-contained manner.

The decision which development environment to use was left to the writer, but it was recommended to use Truffle in conjunction with Ganache as this provides an out-of-the-box solution for Ethereum based Blockchain development.

## 1.2 Outline

The remaining of this work is structured as follows. The next chapter will mainly focus on related work and provide a brief introduction to the overall context this paper is written in. Chapter 3 describes the underlying scheme which has to be implemented in further details. Chapter 4 focuses on the design decisions that were made, whereas Chapter 5 will describe the actual implementation and the steps taken. Finally, Chapter 6 is used to evaluate the solution at hand and in Chapter 7 conclusions are drawn

# Chapter 2

## Basic Concepts

This chapter is intended to give a more detailed insight into the technologies and basic concepts that the writer was confronted with during the implementation of his task.

### 2.1 Blockchain

According to [9], the Blockchain consists of data sets which are composed of a chain of data packets (blocks). The individual blocks contain several transactions as well as the timestamp, the hash value of the previous block, and a nonce (random value to verify the hash). A very simplified overview of this architecture is depicted in fig. 2.1. With the help of this information, it is possible to ensure the integrity of the chain, because a change of the values in a block would change its hash and therefore, the validity of the chain would not be given anymore. If the majority of the network participants find a consensus as to whether the transactions in a new block, as well as the block itself, are valid, the block is added to the chain and thus forming a complete ledger of the transaction history. This mechanism is described by [9] as the process in which the validators agree on a state of the ledger. Compared to traditional server architectures, where each application is run in its own isolated silos, the Blockchain relies on anyone who can set up a node to replicate the data needed to make a deal with the others and be compensated for it. According to [3] this ensures that the data remains private and the applications function decentralized. This system allows people around the globe to create a trustworthy environment and transfer different types of assets peer-to-peer.

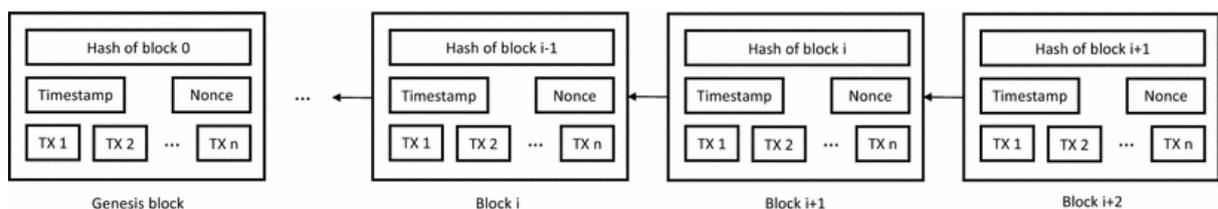


Figure 2.1: Simplified structure of a Blockchain

## 2.2 Ethereum & Smart Contracts

Ethereum is described in [3] as a distributed platform that executes Smart Contracts, i.e. applications that run exactly as they are programmed, without the possibility of downtime, censorship, fraud or third-party intervention. Furthermore it is described as an extension of the Bitcoin Blockchain but with the goal to support a broader scope of applications. The contracts are designed to enable developers to create markets, store registers of debts or promises, move funds or many other use cases, all without a trusted third party. It is even assumed by some sources (e.g. [9]) that Smart Contracts and the Blockchain have the potential to eliminate lawyers and banks that have been involved in the transfer of assets until now. So basically it can be said that Smart Contracts are a digital representation of a contract that regulates the exchange of assets between parties under certain conditions and are deployed on the Blockchain.

## 2.3 Blockchain Signaling System

The system presented by [1] should help to share hardware and defense capabilities with other systems in order to enhance the defensive capabilities against DDoS attacks, since many current centralized systems no longer meet performance requirements and do not have the resources to detect and mitigate large-scale attacks themselves. They propose Smart Contracts and the Blockchain to distribute DDoS attack information across multiple domains and create financial incentives for the involved actors to participate in the process. In the approach presented, Smart Contracts and decentralized applications (dApps) are the main components. Smart Contracts describe how information is exchanged between systems, and dApps are used to contain the parameters that define how an Autonomous System (AS) behaves in a cooperative defense. A high-level architecture of the system is depicted in fig. 2.2. The system should work as follows: An Autonomous System first creates an account and reports this to the central Smart Contract that has information about the individual networks that the AS operates. The token price could also be recorded in it. The individual ASes Smart Contracts implement their own method to get the addresses from the central Smart Contract. It must be ensured that this table is always up-to-date, otherwise the requests may not be accepted if the addresses change. Using this information, ASes can now send requests for cooperative defense to other ASes with a list of addresses. These requests can then be accepted or rejected based on the individual policies. A transaction is completed when a log describing the actions taken is sent. To prevent misuse, [1] propose financial penalties, termination of membership, or legal actions via Service Level Agreement (SLA) definition.

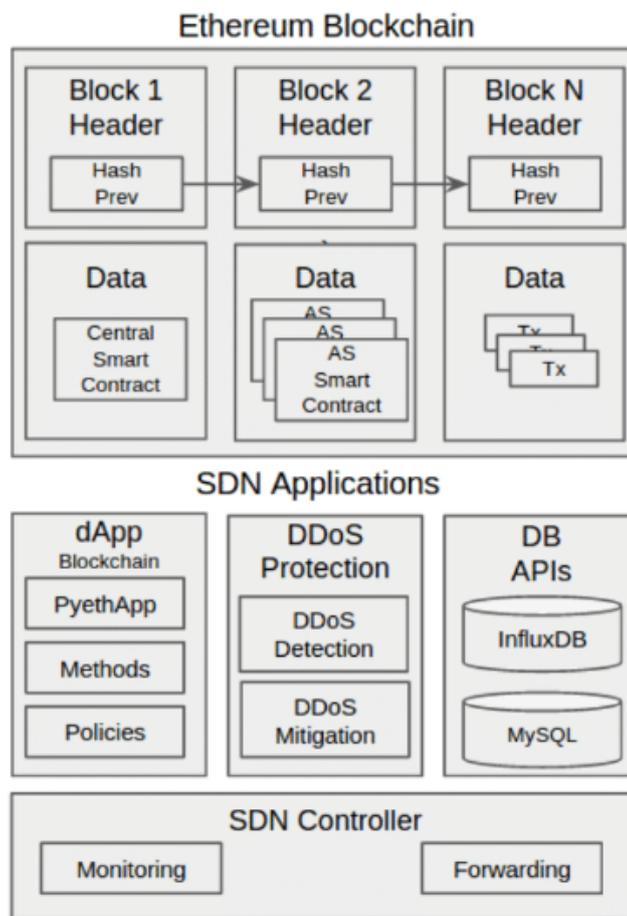


Figure 2.2: Architecture of the BloSS as introduced by [1]



# Chapter 3

## Protocol Requirements

This chapter is intended to provide a more detailed insight into the protocol to be implemented and explain why such a protocol is necessary to ratify the quality of service provided by the mitigators.

### 3.1 Cooperative Signaling Protocol Scheme

The overall objective of the Cooperative Signaling Protocol is to provide the platform for the cooperation between a mitigator and a target and to handle the rating of a corresponding mitigation service. Depending on the evaluation and final state of the process, the mitigator, the target, or neither will be rewarded. The schema of the protocol is depicted in fig. 3.1.

The process begins with an initial cooperative defense request of target T to a potential mitigator M, which can accept or reject it. If the request is accepted, T must transfer the promised sum to the contract, which is kept there until the final evaluation. However, if the request is rejected, the process is terminated. After sending the agreed sum, the deadline timer  $t_0$  starts that defines in which time interval the mitigator M must send a confirmation of the performed service. M can now act rationally and send a proof or let the time elapse. In both cases, however, it is not possible to guarantee the basic correctness or quality of the proof. Even if the Blockchain ensures an audit trail, as mentioned by [1], no ground truth can be ensured. This problem exists for the upload of the proof as well as for the user rating, but unfortunately there is no fully automated way to guarantee truthfulness. Next, T has to rate M's service accordingly, which is again limited by a validation deadline. If a proof has been uploaded, T can either be satisfied, not satisfied or not answering. After the expiration of the deadline or an early response from T, M may in turn issue a rating. A rational M will rate T as negative if the service is refused. However, if T has given positive feedback, M will also give positive feedback. If T is selfish (no response), M will also respond negatively. All these decisions eventually lead to the options listed in the last column of fig. 3.1, which describes who will receive a reward.

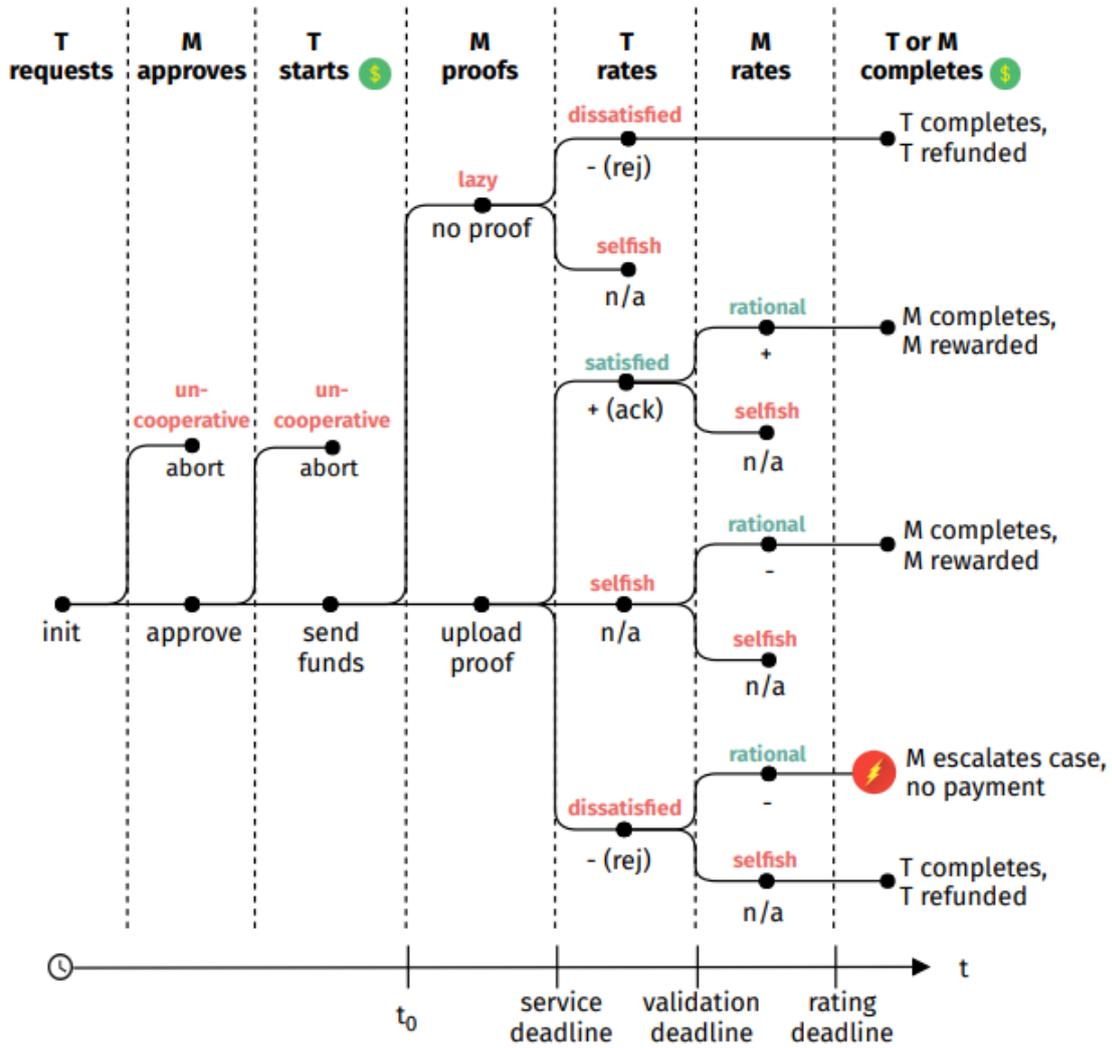


Figure 3.1: Definition of the protocol by [1]

# Chapter 4

## Design

Since this was a completely new topic for the writer, it was decided to explore the possible problems through an explorative process. However, before programming could be started, a development environment had to be selected first. As Truffle was mentioned as a comprehensive and easy to use tool in cooperation with Ganache, it was decided to make the initial steps with this suite. The first approach was clearly aiming at ensuring the functionality, which unfortunately was at the expense of expandability and maintainability. In a second attempt, the problems of the first prototype were taken into consideration and above all the maintainability and expandability were addressed. In order to achieve these goals, it was decided to follow an object oriented approach and therefore implementing the state pattern. Due to the writer's suspicion that the created implementation could be too complex and costly for a smart contract, it was decided to offer a third alternative, which continues the simplicity of the first prototype, but makes some selective adjustments.

### 4.1 First Approach

The first approach for the development of the Cooperative Signaling Protocol consists of a very simple and limited architecture as depicted in fig. 4.1. The aim is to provide an interface where a user could perform actions via the main contract (Protocol.sol). A new process could be initiated and the current state advanced via function calls (approve, fund, proof, rateByTarget, rateByMitigator). It was crucial to include the address of the desired process, since an account can, for example, be involved in several active processes and therefore the sender of a message cannot be traced to the process. At the end, the evaluation (located in the process itself) is initiated and the payment or non-payment is regulated based on the defined scheme. A closer look revealed some problems or misunderstandings regarding the implementation (e.g. a mitigator has to specify how much funds are necessary per blocked address and thus, it is not enough to start the process with the user's account only), or that an adjustment of the evaluation algorithm is a bit cumbersome if the whole code has to be touched just to rework a small part. For these reasons, it was decided to adapt the structure and introduce a state pattern based on the individual states. This architecture will be explained in more detail in section

4.2. In order to be able to draw a certain comparison nevertheless, the simpler first model was extended by the desired features and continued parallel to the development of state pattern approach. This third way, called the simple approach, is further described in section 4.3. The writer believes that this will enable him to draw conclusions about whether it is even suitable to integrate a more complex architecture into a Smart Contract or not.

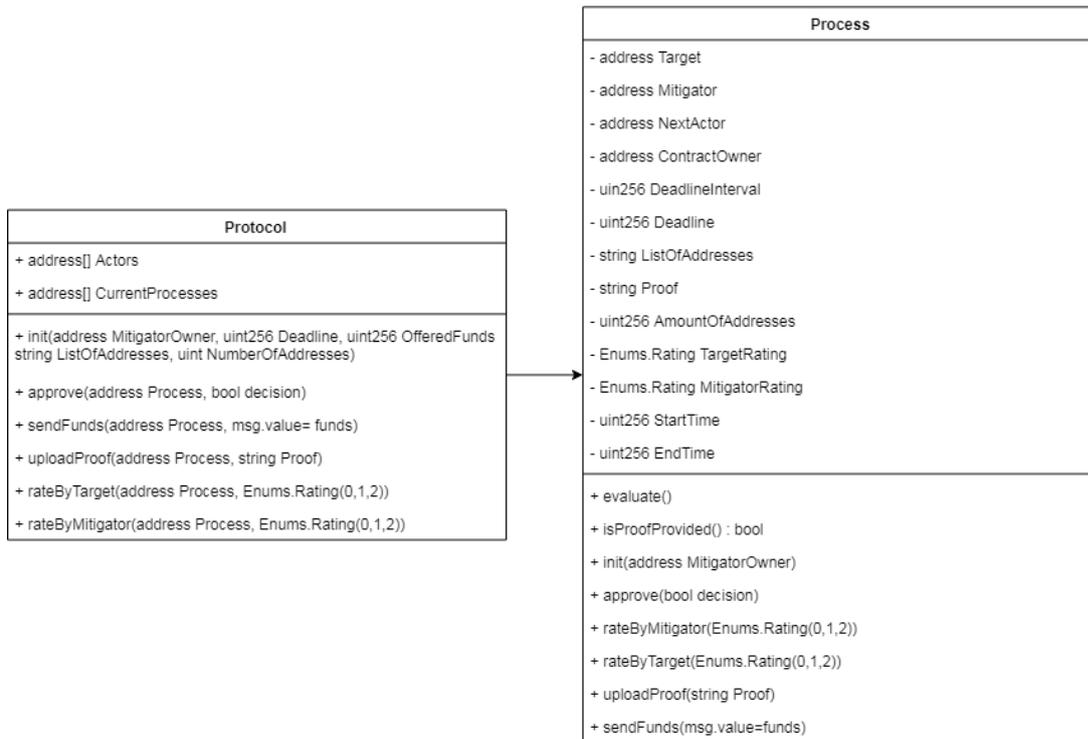


Figure 4.1: Architecture of the first prototype

## 4.2 Implementation With State Pattern

Due to the existence of individual states, the state pattern (combined with a factory) appeared to be suitable in cooperation with strategy pattern for evaluation. The planned design is shown in Fig 4.2. In this approach, a user first has to register as an actor and tell the price he charges per blocked or filtered address and what his network is called. Then the user can trigger actions via the interface provided by the main contract. It is important to note that an account can only be registered once as an actor and must first be deleted and recreated if necessary. However, this is only possible if the actor is not involved in an active process. If an authorized user now submits a request, it is forwarded to the corresponding state object, which interprets the call differently according to the state action. If everything goes according to plan, the process will finally fall into the evaluation state at the end of a path, from where it will calculate who will receive a reward and what final status will be set. Finally, the process falls into the abort, complete or escalate state. The described design consists of the following elements:

- **Protocol** Is the interface for the clients. It handles the registration of actors and manages the progress of the process. It serves as a facade for the underlying structure
- **Process** Contains the address of the current state and the address of the data object. It is concerned with calling the execution function on state objects and the instantiation of new ones.
- **Actor** Contains user related data like the price per unit or network name. It also contains the owners address.
- **StateFactory** Is used to create new state objects. It plays a crucial role which is explained in the next chapter.
- **IState** Is the interface of the states, provides four different execute methods because Solidity does not yet have generics.
- **EvaluationFactory** Is used to create new evaluation objects. It plays a crucial role which is explained in the next chapter.
- **IEvaluation** Is the interface for evaluation strategies. Contracts implementing this interface agree on the function evaluation() which returns the address to be payed and the state to be set

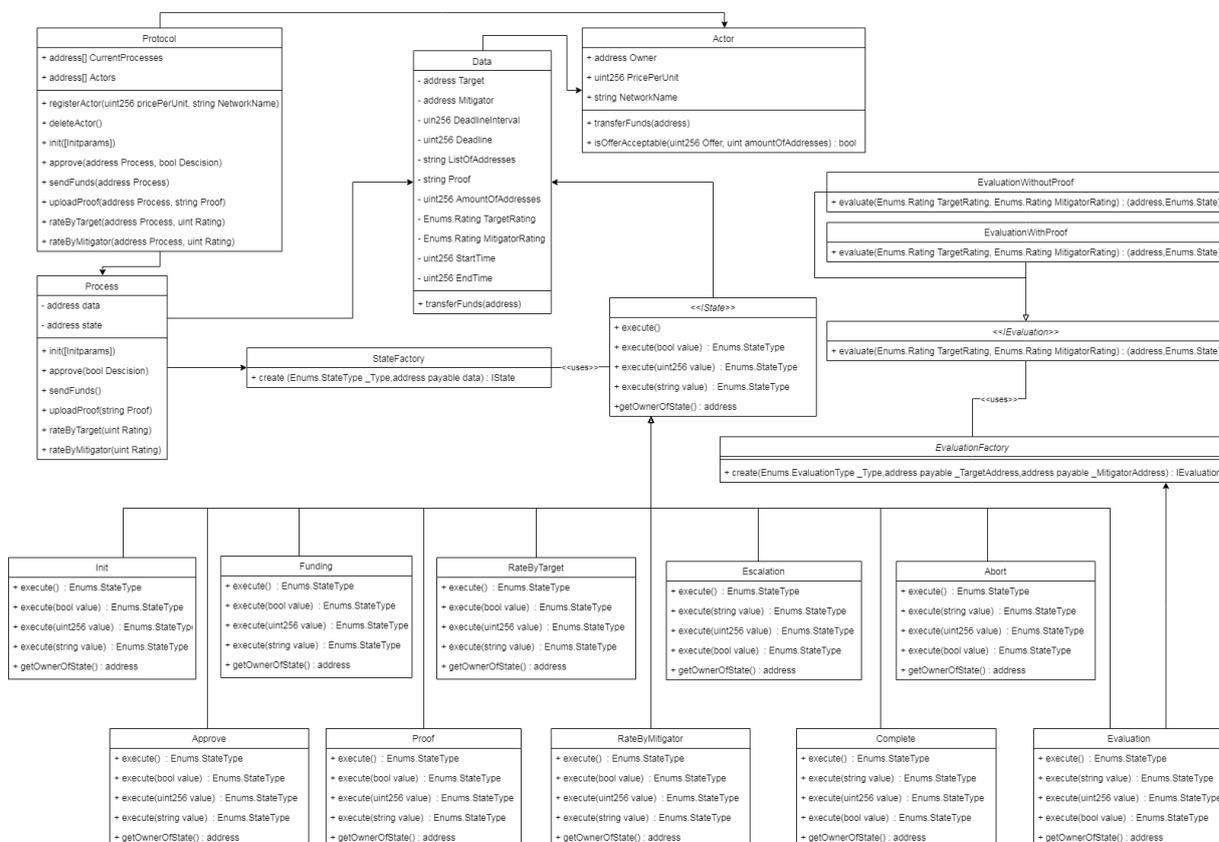


Figure 4.2: Architecture of the implementation with State Pattern

### 4.3 Refined First Prototype

The third approach, called simple approach, was based on a lean architecture without forgetting the problems of the first prototype. Therefore, it was decided to use a factory for the evaluation algorithm in combination with the strategy pattern (guarantees a certain degree of flexibility). In addition, as with the state pattern approach, an actor contract was used to set the price per unit and the network name. However, since the state pattern is not used in this approach, the individual states were handled using Enumerations. This has the consequence that adaptations to the states become somewhat more cumbersome and complicated. The design as pictured in fig. 4.3 consists of the following elements:

- **Protocol** Is the interface for the clients. It handles the registration of actors and manages the progress of the process. It is also able to skip a state, when the deadline is exceeded.
- **Process** Contains the actual state and data of a process that has been initiated by the protocol. In all methods it is assured that only the owning contract (protocol) can set new values. This was introduced to prevent a user from instantiating an existing process and change values without using the provided interface.
- **Actor** Contains user related data like the price per unit or network name. It as well contains the owners address.
- **EvaluationFactory** Is responsible for creating concrete evaluation objects.
- **IEvaluation** Is the interface for evaluation strategies. Contracts implementing this interface agree on the function `evaluation()` which returns the address to be payed and the state to be set.

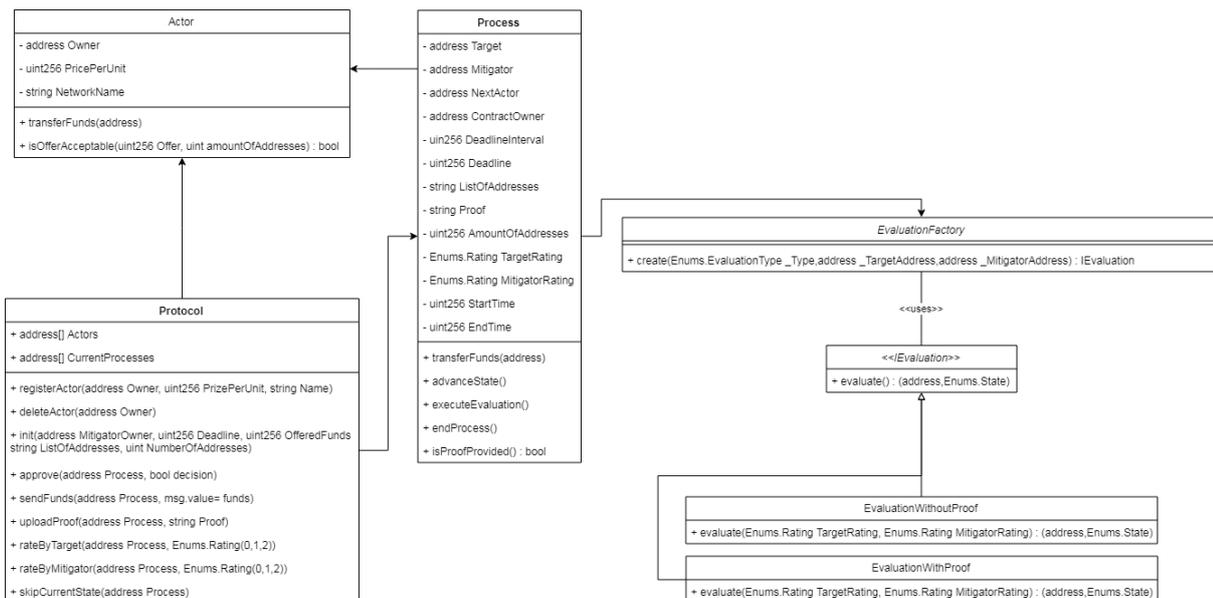


Figure 4.3: Architecture of the simple approach

# Chapter 5

## Implementation

The implementation of the Cooperative Signaling Protocol prototype in this report can basically be divided into four stages that led to the current, final state. These steps are described in more detail in the following sections and it is explained how each step has made an important contribution to the overall understanding of the problem.

### 5.1 First Prototype - Basic Understanding

The first and most important step was certainly to understand the basic functioning of Solidity and Smart Contracts. In the writers own experience, a certain level of knowledge is reached after a short time in order to be able to make the first steps with Solidity. Nevertheless the beginning with Truffle turned out to be a bit difficult, as the official examples published on the internet did not work on various machines and operating systems due to a missing library [5]. Therefore it was decided to use the online IDE Remix [6] for the first steps and to test the prototype there. This worked pretty well in the beginning and the first contract could be developed in a relatively short time. The architecture was as simple as described in section 4.1. This meant that the individual states were handled as enumerations and all operations that changed the state were handled in the process contract itself. This had the consequence that the structure looked extremely simple from the outside, but was internally complicated and not comprehensible, since each method could make any changes to variables and thus, side effects could occur. Furthermore, the evaluation of the final state was not externalized but also stored in the process contract, which in turn inflated the contract considerably and made it illegible. Not to mention the difficulties of possible changes to the algorithm. After a first consultation with the supervisor Bruno Rodrigues, some shortcomings of the approach turned out and the writer went back to the planning phase to work out a second approach.

## 5.2 State Pattern Sounds Very Reasonable

The second approach aimed at ensuring an expandable structure. The existence of different stages in which the process can be located meant that the opportunity for a state pattern was already implicitly given. Furthermore, care was taken to separate the individual evaluation algorithms (with and without proof) and to load them by a strategy pattern. The first implementation with the state pattern was unfortunately not successful, since the bytecode always exceeded the maximum size of 24 Kb due to the used inheritance. The plan for the individual states was to inherit from an abstract contract, which already offered them a basic implementation of the execute methods and the execution permission. In addition, the instance variables and the constructors were already predefined. This architecture is illustrated in fig 5.1. Consequently, the contract had to embed all states via import, which in turn meant that the bytecode of the process contract was always extended by the code of the states (which inherited from the abstract state). In the end, this resulted in a bytecode that was way too big for the entire contract, thus provoking an `OutOfGas` exception. Not to mention that the evaluation algorithms were also defined as abstract contracts, which represented the same problem and inflated the bytecode of the contract again. But what exactly is this bytecode and why is it so crucial? To run Smart Contracts you need the Ethereum Virtual Machine, which is the runtime environment. This virtual machine does not work directly with the Solidity code, but with the compiled bytecode. This code is a set of instructions for the virtual machine based on a strict technical specification. After some time and pondering, the writer decided to omit this approach for a moment and proceed with the first prototype. This decision was taken because the state pattern approach, with its complexity and the understanding of the matter of the writer at that time, did not allow for any progress. The next development steps with the first, simpler prototype are explained in more detail in the following section.

## 5.3 Simplicity is Key

The continuation of the old approach first required a takeover of the implementation of the actor contract and the associated adjustment of the entire chain of dependencies, which until then had only been carried out via the account address. Subsequently, the evaluation was extracted from the process and stored in separate contracts. To guarantee a certain level of flexibility, the strategy pattern was used again, that allows for dynamic exchange of algorithms at runtime. Still the strategies were implemented as contracts, and the evaluation was also an abstract contract, which again bloated up the bytecode unnecessarily. Nevertheless, the bytecode of the contract could be reduced in size overall, making it deployable compared to the initial state pattern approach. In spite of all these changes, the whole construct remained a bit messy, since almost all logic (except for the evaluation) was still represented in the process itself. Together with the single getters and setters this led to a big and confusing contract.

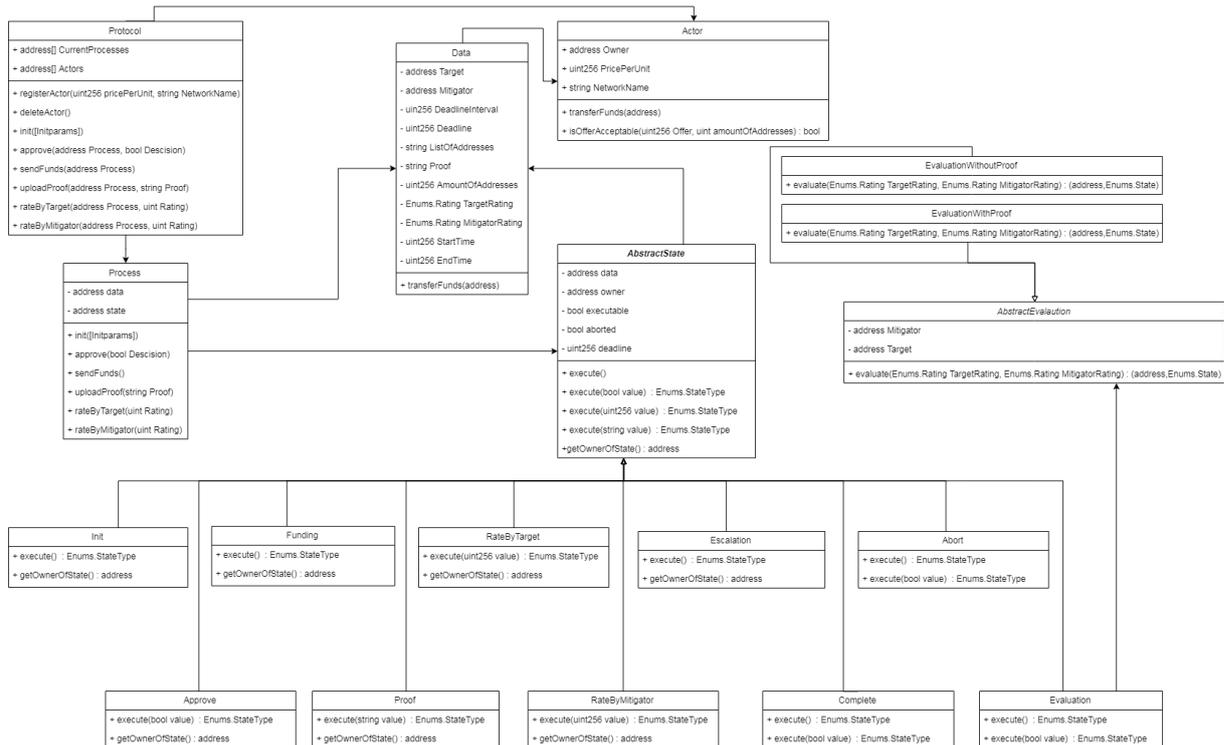


Figure 5.1: Architecture of the first try with a state pattern

## 5.4 Surrendering is Not an Option

After reading some posts in forums and blogs, the writer found an article [4], which described various opportunities to reduce the bytecode size with some techniques. New hope arose that the elaborated solution is still feasible. Probably the most important one mentioned is the externalization of code in libraries. This is especially helpful for factories as it allows the instantiation of contracts to be outsourced. Libraries have the advantage that they are deployed only once on the Blockchain and thus represent a kind of singleton. As a limited factor it can be mentioned that libraries cannot receive ethers and do not have their own storage. But in our case this does not matter for the factories. Furthermore the implementation of interfaces was recommended, especially in combination with libraries and the factories. So that an object with the interface type can be returned from the factory, which decouples the individual components even more from each other. Or as it was written by [4] "An interface is a special type of contract, limited to what the Contract Application Binary Interface (ABI) can represent". This means that it is only possible to describe the function signature in the interface, but not the implementation. This enhances the readability of the code, and the return value can still be guaranteed. The ABI can basically be defined as how to communicate with a smart contract. It defines how to call its functions and how to get data back.

Finally, the underlying structure was adjusted according to the new findings and set up as described in section 4.2. The following sections show the implementation of the StateFactory, as well as a state, and how they interact. What finally happens in the process is illustrated in Listing 5.1. A simple call to the function is enough to get the new status and generate the new state with it. To shorten the code sections a bit, the entire listing of contracts in the StateFactory was omitted and only the Proof and Target Rating State was listed instead. The code of the state has also been shortened a bit. This means that e.g. the instance variables and the constructor have been omitted. The constructor simply sets the owner, the data object and the next deadline.

```

1 function uploadProof(string memory value) public{
2     require(currentState==Enums.StateType.PROOF,"State is not correct");
3     currentState = State.execute(value);
4     State =StateFactory.create(currentState ,address(Data));
5 }

```

Listing 5.1: ProcessCallState

```

1 import "./Enums.sol";
2 import "./IState.sol";
3 import "./StateProof.sol";
4 import "./StateRatingByTarget.sol";
5
6
7 library StateFactory{
8
9     function create(Enums.StateType _Type,address payable data)
10    public returns (IState){
11
12    if(_Type == Enums.StateType.PROOF){
13        return new StateProof(data);
14    }else if(_Type == Enums.StateType.RATE_T){
15        return new StateRatingByTarget(data);
16    }else{
17        revert("Type not in StateFactory");
18    }
19 }
20 }

```

Listing 5.2: StateFactory

```

1 contract StateProof is IState{
2
3     function execute(bool /*value*/) external returns(Enums.StateType) {
4         revert("Not implemented");}
5
6     function execute(uint256 /*value*/) external returns(Enums.StateType)
7         {revert("Not implemented");}
8
9     function execute() external returns(Enums.StateType) {
10        require(executable, "Process not executable");
11        if(canBeSkipped()){
12            executable=false;
13            return Enums.StateType.RATE_T;
14        }else{
15            require(owner == tx.origin, "Error owner != tx.origin");
16        }
17        executable=false;
18        return Enums.StateType.RATE_T;
19    }
20
21    function execute(string calldata value) external returns(Enums.
22        StateType){
23        require(executable, "Process not executable");
24        if(canBeSkipped()){
25            executable=false;
26            return Enums.StateType.RATE_T;
27        }else{
28            require(owner == tx.origin, "Error owner != tx.origin");
29        }
30        IData(data).setProof(value);
31        executable=false;
32        return Enums.StateType.RATE_T;
33    }
34
35    function canBeSkipped() private view returns(bool){
36        if(now>deadline){return true;}
37        return false;
38    }
39
40    function abort() public returns(Enums.StateType){
41        require(owner == tx.origin, "Error owner != tx.origin");
42        aborted=true;
43        executable = false;
44        return Enums.StateType.RATE_T;
45    }
46
47    function getOwnerOfState() external view returns(address payable){
48        return owner;
49    }
50
51    function getStateType() external view returns(Enums.StateType){
52        return Enums.StateType.PROOF;
53    }
54 }

```

Listing 5.3: ConcreteState

## 5.5 Disillusionment spreads

After the implementation of the contract with the state pattern according to all rules of object-oriented programming, the author realized that this approach probably did not quite fit the intention of the task of this MBM and contained far too much complexity. It was concluded that the key to success was likely to lie in simplicity. This sudden realization came from the fact that deployment was very expensive and therefore probably unsuitable for possible requests. Although it provided a very high maintainability, it seemed that this is not the keypoint of this contract. So a new approach was designed that really only provides the minimal functions and doesn't outsource anything. In this case, all actions are now controlled by a single contract, which is recreated for each request. The architecture of the contract is shown in fig. 5.2. This contract is an even more simplified version of the first prototype, which attempts to handle all operations through a single contract.

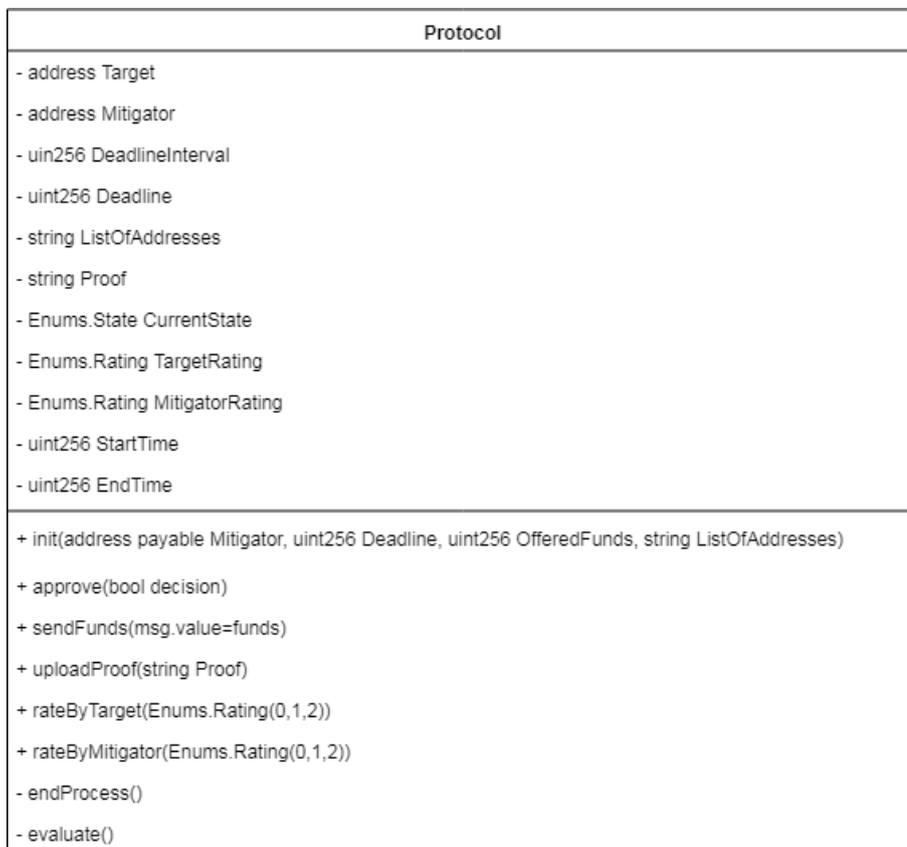


Figure 5.2: Slim Approach

Last but not least it was of course inevitable to test all implementation of the contract extensively, in order to ensure its correctness. This will be explained in more detail in the next chapter, the evaluation. In addition, the performance and efficiency of the contract will be examined in more detail.

# Chapter 6

## Evaluation

The main purpose of the evaluation was to check the correctness, performance and efficiency of the contracts. In order to achieve these goals, detailed test cases were first written and then the speed of execution and the cost of deploying the contract were examined.

### 6.1 Testing

As with the implementation, testing turned out to be an incremental learning process. As already mentioned, the development environment was originally switched from Truffle to Remix, since an error occurred on various machines and operating systems during the initial testing of the Truffle sample application itself. The first steps with smart contracts were then taken in Remix and first experiences were gained. With the increasing size of the contracts and the possibility of auto compilation, the browser became more and more slow and it was only possible to work if this option and some syntax checks were disabled in the settings. When it went to testing for the first time, a first damper occurred. Since the contract is supposed to represent the interaction of two actors, there must of course also be the possibility to simulate the sending of requests from different accounts in the tests. This is however not possible with pure Solidity code, but needs Javascripts.

Since Truffle provides a quite pleasant possibility to write tests for Smart Contracts with the Mocha Framework, the writer decided to give Truffle another chance.

In listing. 6.1 is an example of such a test. Since Truffle has no way of finding out which contracts need to interact in the test, a usable contract abstraction must be requested using the `artifacts.require("")` function. A speciality of the truffle tests compared to the normal Mocha framework is the `contract()` function, as it offers a clean room feature, i.e. the contract is always deployed anew during execution. After the initial imports, the addresses are assigned to the actors and the protocol is instantiated, which is annotated with the function `it()`. Finally, the actual tests can be carried out via `assert` statements.

```

1 var Protocol = artifacts.require("../Protocol.sol");
2 let catchRevert = require("../exceptions.js").catchRevert;
3
4 contract("Full Run Test", async function(accounts) {
5
6     var TargetOwner = accounts[0];
7     var MitigatorOwner = accounts[1];
8     var ListOfAddresses = "Network1,Network2";
9     var instance;
10
11     it("Instantiation", async function() {
12
13         instance = await Protocol.new();
14
15         await instance.init(MitigatorOwner,120,await web3.utils.toWei('2.0',
16             "ether"),ListOfAddresses, {from: TargetOwner});
17
18         assert.equal(await instance.getCurrentState(), 1, "State is wrong");
19         assert.equal(await instance.getListOfAddresses(), ListOfAddresses, "
20             List of addresses is wrong");
21     });
22 });

```

Listing 6.1: Test With Mocha

Since functions can be reverted in Solidity, if certain conditions are not fulfilled, this possibility must also be validated in tests. However, since the tests normally fail if they are reverted, a remedy had to be found. An entry in the Ethereum forum of Stackexchange [8] helped by sharing a script that captures the errors and, depending on whether they are expected or not, no longer throws any errors. This very useful script is shown in listing 6.2. You can choose between different error messages that are expected (e.g. a captured revert, an OutOfGas exception or a wrong sender).

```

1 const PREFIX = "Returned error: VM Exception while processing
2     transaction: ";
3
4 async function tryCatch(promise, message) {
5     try {
6         await promise;
7         throw null;
8     }
9     catch (error) {
10        assert(error, "Expected an error but did not get one");
11        assert(error.message.startsWith(PREFIX + message), "Expected an
12            error starting with '" + PREFIX + message + "' but got '" +
13            error.message + "' instead");
14    }
15 };
16
17 module.exports = {
18     catchRevert      : async function(promise) {await tryCatch(
19         promise, "revert"
20     );},
21     catchOutOfGas   : async function(promise) {await tryCatch(
22         promise, "out of gas"
23     );}
24 };

```

Listing 6.2: Catch a revert, based on various causes

## 6.2 Performance and Efficiency

First the Gas usage and the effective Ether needed to create the contracts were considered. Of course, it was clear that the more complex contracts would require more funding for deployment. However, the writer did not expect this to be so severe as shown in figure 6.1. Just splitting the contracts into different state objects (state pattern approach) increases the required amount of gas almost eight times and costs about seven times as much to create. The differences to the creation in the simple variant are approximately in the middle and need only about four times as much gas as the slim contract. The total costs in Ether are about 4.5 times as high. Nevertheless it has to be said that with the state pattern and the simple approach the libraries only have to be deployed once each time and so there might be a saving over a longer period of time.

Slim Approach	Simple Approach	State Pattern Approach
<pre> 2_contract_migration.js ===== Replacing 'Protocol' &gt; transaction hash: 0xd31e5c8c1ab73 &gt; Blocks: 0 &gt; contract address: 0x054617cfc641e &gt; account: 0x0Ca165e6E5eed &gt; balance: 97.69637902 &gt; gas used: 1127005 &gt; gas price: 20 gwei &gt; value sent: 0 ETH &gt; total cost: 0.0225401 ETH  &gt; Saving migration to chain. &gt; Saving artifacts  &gt; Total cost: 0.0225401 ETH  Summary ===== &gt; Total deployments: 2 &gt; Final cost: 0.02719972 ETH </pre>	<pre> 2_deploy_contracts.js ===== Replacing 'EvaluationFactory' &gt; transaction hash: 0xa0bc28ac7fc56a &gt; Blocks: 0 &gt; contract address: 0x73E7B837ed30a2 &gt; account: 0x0Ca165e6E5eed9 &gt; balance: 62.01738182 &gt; gas used: 504058 &gt; gas price: 20 gwei &gt; value sent: 0 ETH &gt; total cost: 0.01008116 ETH  Linking * Contract: Protocol &lt;--&gt; Library: Eval  Replacing 'Protocol' &gt; transaction hash: 0x042b042d31564c &gt; Blocks: 0 &gt; contract address: 0x7e41fcF5241233 &gt; account: 0x0Ca165e6E5eed9 &gt; balance: 61.9213523 &gt; gas used: 4801476 &gt; gas price: 20 gwei &gt; value sent: 0 ETH &gt; total cost: 0.09602952 ETH  &gt; Saving migration to chain. &gt; Saving artifacts  &gt; Total cost: 0.10611068 ETH  Summary ===== &gt; Total deployments: 3 &gt; Final cost: 0.1107703 ETH </pre>	<pre> Replacing 'EvaluationFactory' &gt; transaction hash: 0x38eb8e3ccdedf &gt; Blocks: 0 &gt; contract address: 0x5a241D0F3a821 &gt; account: 0x0Ca165e6E5eed &gt; balance: 15.1122000 &gt; gas used: 504122 &gt; gas price: 20 gwei &gt; value sent: 0 ETH &gt; total cost: 0.01008244 ETH  Linking * Contract: StateFactory &lt;--&gt; Library:  Replacing 'StateFactory' &gt; transaction hash: 0xd156b05fc3104f &gt; Blocks: 0 &gt; contract address: 0x69295F4d50b2f &gt; account: 0x0Ca165e6E5eed &gt; balance: 15.0175546 &gt; gas used: 4732310 &gt; gas price: 20 gwei &gt; value sent: 0 ETH &gt; total cost: 0.0946462 ETH  Linking * Contract: Protocol &lt;--&gt; Library: Sta  Replacing 'Protocol' &gt; transaction hash: 0xc72b7187704e5 &gt; Blocks: 0 &gt; contract address: 0x76456CDE1FC51 &gt; account: 0x0Ca165e6E5eed &gt; balance: 14.93742008 &gt; gas used: 4006686 &gt; gas price: 20 gwei &gt; value sent: 0 ETH &gt; total cost: 0.08013372 ETH  &gt; Saving migration to chain. &gt; Saving artifacts  &gt; Total cost: 0.18486236 ETH  Summary ===== &gt; Total deployments: 4 &gt; Final cost: 0.18952198 ETH </pre>

Figure 6.1: Overview of the costs and gas used to deploy the contract

Second, the effective duration needed to end the contract was looked at. The times of the test execution and the time it takes from the beginning of the process to the end were taken into account. The timestamps gathered are depicted in fig. 6.2. The State Pattern and the Simple approach took about the same amount of time, but the first one ran a bit more efficiently. Again, the slim approach, which due to its simplicity had the least effort with the requests, was the fastest one.

Slim Approach	Simple Approach	State Pattern Approach
<pre> Contract: Full Run Test   ✓ Instantiation &lt;266ms&gt;   ✓ Approve &lt;95ms&gt;   ✓ Send Funds &lt;11ms&gt;   ✓ Upload Proof &lt;125ms&gt;   ✓ Rate By Target &lt;124ms&gt;   ✓ Rate By Mitigator &lt;168ms&gt; StartTime: 1550003399 EndTime: 1550003400 Duration: 1   ✓ Time &lt;64ms&gt; </pre>	<pre> Contract: Full Run Test   ✓ Actor Creation &lt;542ms&gt;   ✓ Fullrun &lt;2233ms&gt; StartTime: 1550004542 EndTime: 1550004544 Duration: 2   ✓ Time &lt;11ms&gt; </pre>	<pre> Contract: Full Run Test   ✓ Actor Creation &lt;384ms&gt;   ✓ Instantiation &lt;370ms&gt;   ✓ Approve &lt;279ms&gt;   ✓ Send Funds &lt;240ms&gt;   ✓ Upload Proof &lt;386ms&gt;   ✓ Rate By Target &lt;234ms&gt;   ✓ Rate By Mitigator &lt;333ms&gt; StartTime: 1550003726 EndTime: 1550003727 Duration: 1   ✓ Time &lt;88ms&gt; </pre>

Figure 6.2: Overview of the time it took in each full run to complete

# Chapter 7

## Summary and Conclusions

The core topic of this work was to implement a Smart Contract, according to the guidelines of [1] and [2]. As development environment Truffle was used in conjunction with Ganache and the whole solution had to be deployed on an Ethereum Blockchain. The overall goal of the contract is to allow the target of a DDoS attack (target) and potential helpers, so called mitigators, to reach an agreement, which would include rewards or penalties if expectations were not met.

Smart Contracts were used for this purpose, as they create a basis of trust without having to involve a third party. This is achieved through the property of Blockchains that only new entries can be added, but old ones never deleted. Thus all changes are always traceable. The implementation itself turned out to be an ongoing learning process from which three different prototypes emerged. These were the slim approach, which consists of a single contract and is characterized by speed and little need for resources. There is the middle way, the simple approach, which outsources parts of its architecture and involves some concepts of object-oriented development (factory method and strategy pattern) and last but not least there is the state pattern approach, which is built completely in an object-oriented way and is supposed to ensure maintainability and extensibility. In all respects, it must be noted that from a cost perspective the slim approach is certainly the most promising option, as the others are relatively resource-intensive and may not be the best choice for the Blockchain. Nevertheless, the prototypes with the object-oriented approach certainly offer a certain advantage when it comes to maintenance issues, but it remains to be seen whether this additional effort will be worth it and whether the system becomes so complex that it really pays off. In the end, it must be said that so much thought can flow in an architecture, but if it works against the technology used, it will almost inevitably fail. Still, the writer has learned a lot from the work presented here and is grateful for the opportunity to familiarize himself with Smart Contracts and learn a little more about the blockchain. With the three different prototypes a solid basis for further work on a Cooperative Signaling Protocol can certainly be laid.



# Bibliography

- [1] Bruno Rodrigues, Thomas Bocek, Burkhard Stiller: *Blockchain Signaling System (BloSS): Enabling a Cooperative and Multi-domain DDoS Defense*, Demonstrations of the 42nd Annual IEEE Conference on Local Computer Networks (LCN-Demos 2017), Singapore, 8-12 October 2017
- [2] Andreas Gruehler, Bruno Rodrigues: *A Reputation and Reward Scheme for a Cooperative, Multi-domain DDoS Defense*, Universitaet Zuerich, Communication Systems Group, Department of Informatics, Zuerich, Switzerland. In progress
- [3] Ethereum: <https://www.ethereum.org>, 2019, Online, accessed: February 7 2019
- [4] Out Of Gas Error: <https://medium.com/daox/avoiding-out-of-gas-error-in-large-ethereum-smart-contracts-18961b1fc0c6>, 2019, Online, accessed: February 7 2019
- [5] Assert.sol not found Error <https://github.com/trufflesuite/truffle/issues/968>, 2019, Online, accessed: February 7 2019
- [6] Remix IDE <https://remix.ethereum.org/>, 2019, Online, accessed: February 11 2019
- [7] Truffle Testing Documentation <https://truffleframework.com/docs/truffle/testing/writing-tests-in-javascript>, 2019, Online, accessed: February 7 2019
- [8] Truffle Testing Catch Revert <https://ethereum.stackexchange.com/questions/48627/how-to-catch-revert-error-in-truffle-test-javascript>, 2019, Online, accessed: February 7 2019
- [9] Michael Nofer, Peter Gomber, Oliver Hinz, Dirk Schiereck: *Blockchain*, Springer Fachmedien Volume 59, Issue 3, Wiesbaden, June 2017
- [10] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Huaimin Wang: *Blockchain Challenges and Opportunities: A Survey*, Int. J. Web and Grid Services, Vol. 14, No. 4, 2018
- [11] Reddy et al.: *Distributed Denial-of-Service Open Threat Signaling (DOTS) Data Channel*, IETF (Internet Engineering Task Force), draft, <https://www.ietf.org/id/draft-ietf-dots-data-channel-11.txt>, accessed: 11.02.2019



# Abbreviations

ABI	Application Binary Interface
AS	Autonomous System
BloSS	The Blockchain Signaling System
dApp	Decentralized App
DDoS	Distributed Denial of Service
SLA	Service Level Agreement



# Glossary

**ABI** Application Binary Interface

**BloSS** The Blockchain Signaling System (BloSS) proposed by [1] is a novel approach deploying hardware to simplify the signaling of DDoS attacks in a cooperative network defense system.

**dApp** Decentralized App

**DDoS** Distributed Denial of Service, is defined by [11] as an attempt to make machines or network resources unavailable to their intended users by flooding a target with data sent simultaneously from multiple clients.

**Smart Contract** A digital protocol that facilitates the exchange of assets between parties and provides a trustworthy environment.

**Solidity** Programming language which Ethereum Smart Contracts are built upon.

**Truffle** Framework for Smart Contract development



# List of Figures

2.1	Simplified structure of a Blockchain . . . . .	3
2.2	Architecture of the BloSS as introduced by [1] . . . . .	5
3.1	Definition of the protocol by [1] . . . . .	8
4.1	Architecture of the first prototype . . . . .	10
4.2	Architecture of the implementation with State Pattern . . . . .	11
4.3	Architecture of the simple approach . . . . .	12
5.1	Architecture of the first try with a state pattern . . . . .	15
5.2	Slim Approach . . . . .	18
6.1	Overview of the costs and gas used to deploy the contract . . . . .	21
6.2	Overview of the time it took in each full run to complete . . . . .	22



# List of Listings

5.1	ProcessCallState . . . . .	16
5.2	StateFactory . . . . .	16
5.3	ConcreteState . . . . .	17
6.1	Test With Mocha . . . . .	20
6.2	Catch a revert, based on various causes . . . . .	20



# Appendix A

## Contents of the CD

The attached CD contains the following files and directories:

- `Zusfsg.txt`: Plain text version of the German abstract.
- `Abstract.txt`: Plain text version of the English abstract
- `Report.pdf`: PDF of the Report
- `code\`: Directory containing the code of the MBM
- `tex`: The LaTeX source of the Report