University of Zurich^UZH

Cooperative Signaling Protocol

Dominik Buenzli
Zürich, Switzerland
Student ID: 14-707-111

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

University of Zurich
Department of Informatics (IFI)
Binzmueuhlestrasse 14, CH-8050 Zuerich, Switzerland

MASTER BASIS MODULE – Communication Systems Group, Prof. Dr. Burkhard Stiller

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

Zusammenfassung

Das sammeln und analysieren von Daten nimmt einen immer höheren Stellenwert in unserem Alltag ein und eine Vielzahl von Systemen, sei dies in privaten Haushalten, Spitälern oder militärischen Umgebungen verlassen sich heute auf sogenannte drahtlose Sensornetzwerke. Ein solches Netzwerk besteht aus einzelnen Knoten mit eingeschränkter Hardware, die ihre Umgebung überwachen und Daten via Multi-Hopping oder direkter Verbindung an eine zentrale Stelle, dem sogenannten Gateway, senden. Im Anschluss werden die Daten einem autorisierten Benutzerkreis zur Verfügung gestellt. Diese Bachelorarbeit behandelt hauptsächlich das bestehende Web-based Mobile Access And Data Handling Framework (WebMaDa), welches benutzt wird um die Daten einem autorisierten Benutzerkreis via Web zugänglich zu machen. Momentan ist diese Plattform jedoch stark auf persönliche Interaktion der einzelnen Benutzer ausgerichtet. Im Rahmen dieser Arbeit werden die bestehenden Prozesse rund um die Erstellung von Benutzern und dem Vergeben von Zugriffsrechten auf die Sensordaten überarbeitet. Damit einhergehend soll die Transparenz erhöht werden, um zu garantieren, dass zu jedem Zeitpunkt immer klar ist welcher Benutzer oder Administrator Änderungen am System vorgenommen hat. Schlussendlich müssen die protokollierten Einträge der zuständigen Person übersichtlich dargestellt werden. Die Umsetzung wurde erfolgreich mit mehreren Sensoren, Benutzern und Netzwerken getestet.

Abstract

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

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 connection 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 hardware 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. Thus, the load, which normally lies on a single target, can be distributed among many, 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. The scheme consists of three steps according to XXX and XXX. Firstly, an attacker publishes the harmful IP addresses, secondly these are blocked or filtered by the other participants 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 (XXX and XXX). 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 Cooperative Signaling Protocol, which was presented by XXX and XXX in their work and is further described in chapter 3. Ethereum will serve as platform and the whole process will be implemented as Smart Contract.

1.1 Task Description

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

- **Design and development of the protoype.** This MBM needs to implement the aforementioned exchange of messages as defined in the cooperative signaling protocol. To achieve this, the Ethereum Blockchain has to be used with at least two instances (target and mitigator) interacting with each other.
- **Evalaution** The MBM needs to deliver a short evaluation of the implementation and a indication of performance expectations.

  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 related work that the writer was confronted with during the implementation of his Tak.

2.1 Blockchain

2.2 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.

According to [3], this allows developers to create markets, store registers of debts or promises, move funds or many other use cases, all without a trusted third party.

2.3 Smart Contracts

2.4 Blockchain Signaling System

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 Optimistic Fair Exchange

3.2 Cooperative Signaling Protocol Scheme

The overall objective of the Cooperative Signaling Protocol is to have a mitigation service evaluated by the service providing mitigator and the potential target of an attack. Depending on the evaluation or 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 t0 starts which 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 deathline. 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 self-referral (no response), M will also respond negatively. All these decisions eventually lead to the options listed in the last column.

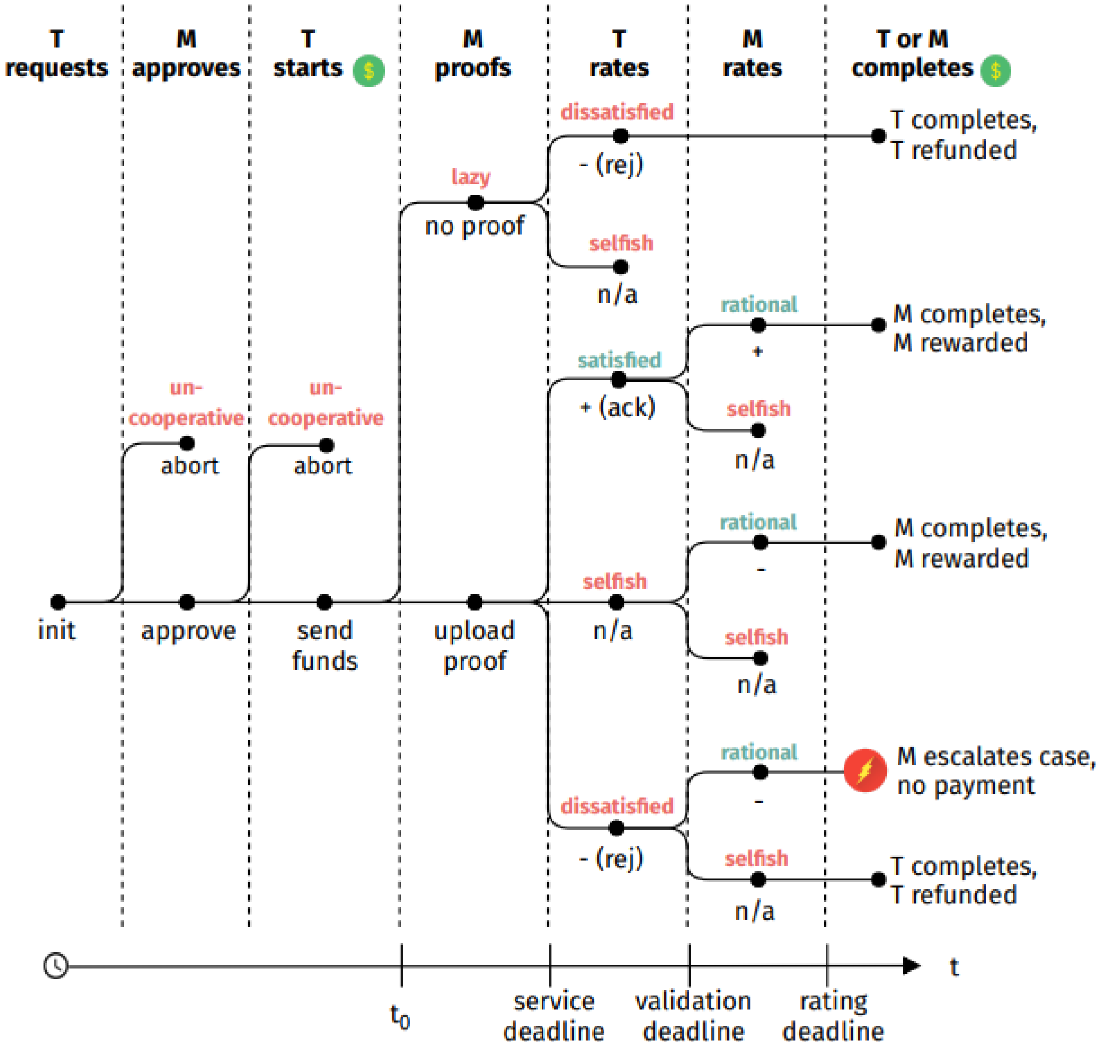

Figure 3.1: Definition of the Protocol by [1]

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. Truffle was selected as the development environment in cooperation with Ganache. The first approach was clearly aimed 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 implement the state pattern. Due to the writer's suspicion that the implementation using state pattern could be too complex, it was decided to offer an 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 consisted of a very simple and limited architecture as depicted in 4.1. The aim was to provide the user with an interface where a user could perform actions via the main contract (Protocol.sol). A new process could be initiated and driven by 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) was initiated and the payment or non-payment was 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 he wants per blocked address and therefore it is not enough to start the process with the user's account only), or that an adjustment of the evaluation is a bit cumbersome if the whole code has to be handled.

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

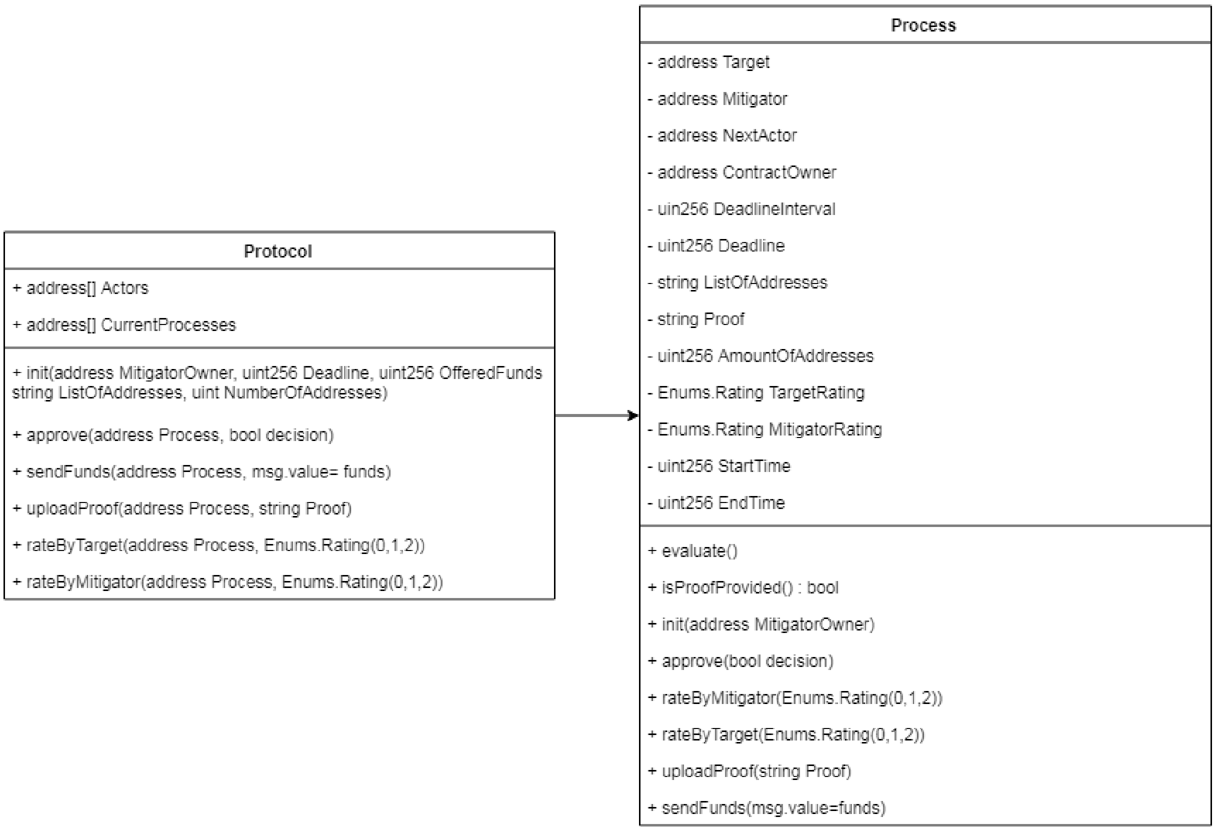
Protocol
+ address[] Actors
+ address[] CurrentProcesses
+ init(address MitigatorOwner, uint256 Deadline, uint256 OfferedFunds string ListOfAddresses, uint NumberOfAddresses)
+ approve(address Process, bool decision)
+ sendFunds(address Process, msg.value= funds)
+ uploadProof(address Process, string Proof)
+ rateByTarget(address Process, Enums.Rating(0,1,2))
+ rateByMitigator(address Process, Enums.Rating(0,1,2))
Process
- address Target
- address Mitigator
- address NextActor
- address ContractOwner
- uin256 DeadlineInterval
- uint256 Deadline
- string ListOfAddresses
- string Proof
- uint256 AmountOfAddresses
- Enums.Rating TargetRating
- Enums.Rating MitigatorRating
- uint256 StartTime
- uint256 EndTime
+ evaluate()
+ isProofProvided() : bool
+ init(address MitigatorOwner)
+ approve(bool decision)
+ rateByMitigator(Enums.Rating(0,1,2))
+ rateByTarget(Enums.Rating(0,1,2))
+ uploadProof(string Proof)
+ sendFunds(msg.value=funds)

Figure 4.1: Architecture of the first prototype

was extended by the desired features and continued parallel to the development of the model with the State Pattern. It 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.

4.2 Implementation With State Pattern

Due to the existence of individual states, at first glance 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 what price he charges per address and what his network is called. The user can then trigger actions via the interface provided by Protocol.sol. 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, and from there into the Abort, Complete or Escalate states. The 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 networkname. It as well contains the owners address.
- **StateFactory** Is used to create new state objects, has 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.
- **Init** Example of a concrete implementation of a state, contains all the logic of a state.
- **EvaluationFactory** Is used to create new evaluation objects, has a crucial role, which is explained in the next chapter.
- **IEvaluation** Is the interface for evaluation strategies
- **EvaluationWithProof** Example of a concrete implementation of a strategy, returns the address to be payed and the new state that has to be set.

4.3 Refined First Prototype

This 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 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. Is 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 started 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.

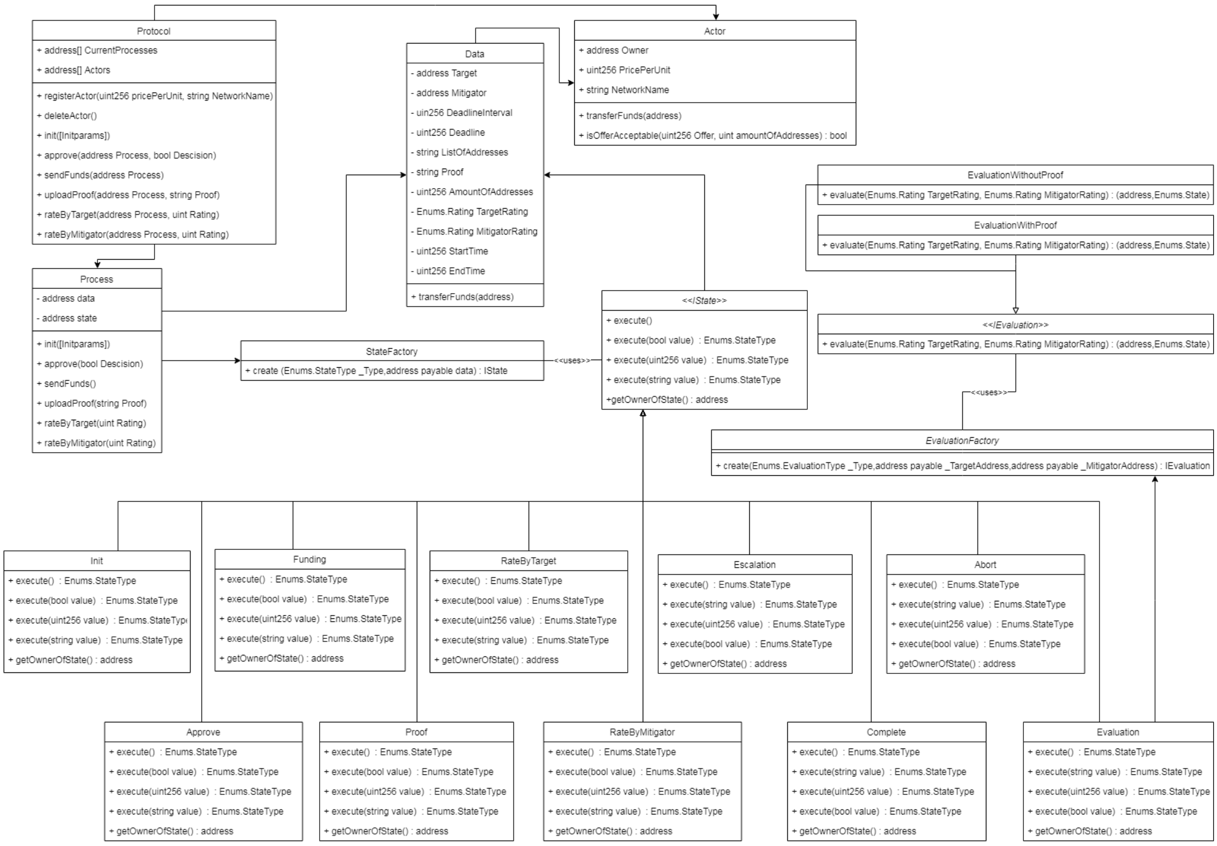
Protocol
+ address[] CurrentProcesses
+ address[] Actors
+ registerActor(uint256 pricePerUnit, string NetworkName)
+ deleteActor()
+ init([Initparams])
+ approve(address Process, bool Descision)
+ sendFunds(address Process)
+ uploadProof(address Process, string Proof)
+ rateByTarget(address Process, uint Rating)
+ rateByMitigator(address Process, uint Rating)
Data
- address Target
- address Mitigator
- uin256 DeadlineInterval
- uint256 Deadline
- string ListOfAddresses
- string Proof
- uint256 AmountOfAddresses
- Enums.Rating TargetRating
- Enums.Rating MitigatorRating
- uint256 StartTime
- uint256 EndTime
+ transferFunds(address)
Actor
+ address Owner
+ uint256 PricePerUnit
+ string NetworkName
+ transferFunds(address)
+ isOfferAcceptable(uint256 Offer, uint amountOfAddresses) : bool
EvaluationWithoutProof
+ evaluate(Enums.Rating TargetRating, Enums.Rating MitigatorRating) : (address,Enums.State)
EvaluationWithProof
+ evaluate(Enums.Rating TargetRating, Enums.Rating MitigatorRating) : (address,Enums.State)
Process
- address data
- address state
+ init([Initparams])
+ approve(bool Descision)
+ sendFunds()
+ uploadProof(string Proof)
+ rateByTarget(uint Rating)
+ rateByMitigator(uint Rating)
<<IState>>
+ execute()
+ execute(bool value) : Enums.StateType
+ execute(uint256 value) : Enums.StateType
+ execute(string value) : Enums.StateType
+getOwnerOfState() : address
<<IEvaluation>>
+ evaluate(Enums.Rating TargetRating, Enums.Rating MitigatorRating) : (address,Enums.State)
StateFactory
+ create (Enums.StateType _Type,address payable data) : IState
<<uses>>
<<uses>>
EvaluationFactory
+ create(Enums.EvaluationType _Type,address payable _TargetAddress,address payable _MitigatorAddress) : IEvaluation
Init
+ execute() : Enums.StateType
+ execute(bool value) : Enums.StateType
+ execute(uint256 value) : Enums.StateType
+ execute(string value) : Enums.StateType
+ getOwnerOfState() : address
Funding
+ execute() : Enums.StateType
+ execute(bool value) : Enums.StateType
+ execute(uint256 value) : Enums.StateType
+ execute(string value) : Enums.StateType
+ getOwnerOfState() : address
RateByTarget
+ execute() : Enums.StateType
+ execute(bool value) : Enums.StateType
+ execute(uint256 value) : Enums.StateType
+ execute(string value) : Enums.StateType
+ getOwnerOfState() : address
Escalation
+ execute() : Enums.StateType
+ execute(string value) : Enums.StateType
+ execute(uint256 value) : Enums.StateType
+ execute(bool value) : Enums.StateType
+ getOwnerOfState() : address
Abort
+ execute() : Enums.StateType
+ execute(string value) : Enums.StateType
+ execute(uint256 value) : Enums.StateType
+ execute(bool value) : Enums.StateType
+ getOwnerOfState() : address
Approve
+ execute() : Enums.StateType
+ execute(bool value) : Enums.StateType
+ execute(uint256 value) : Enums.StateType
+ execute(string value) : Enums.StateType
+ getOwnerOfState() : address
Proof
+ execute() : Enums.StateType
+ execute(bool value) : Enums.StateType
+ execute(uint256 value) : Enums.StateType
+ execute(string value) : Enums.StateType
+ getOwnerOfState() : address
RateByMitigator
+ execute() : Enums.StateType
+ execute(bool value) : Enums.StateType
+ execute(uint256 value) : Enums.StateType
+ execute(string value) : Enums.StateType
+ getOwnerOfState() : address
Complete
+ execute() : Enums.StateType
+ execute(string value) : Enums.StateType
+ execute(uint256 value) : Enums.StateType
+ execute(bool value) : Enums.StateType
+ getOwnerOfState() : address
Evaluation
+ execute() : Enums.StateType
+ execute(string value) : Enums.StateType
+ execute(uint256 value) : Enums.StateType
+ execute(bool value) : Enums.StateType
+ getOwnerOfState() : address

Figure 4.2: Architecture of the implementation with State Pattern

- **Actor** Contains user related data like the price per unit or networkname. It as well contains the owners address.
- **EvaluationFactory** Is responsible for creating concrete evaluation objects.
- **IEvaluation** Is the interface for evaluation strategies.
- **EvaluationWithProof** Example of a concrete strategy, returns the address to be payed and the new state that has to be set.

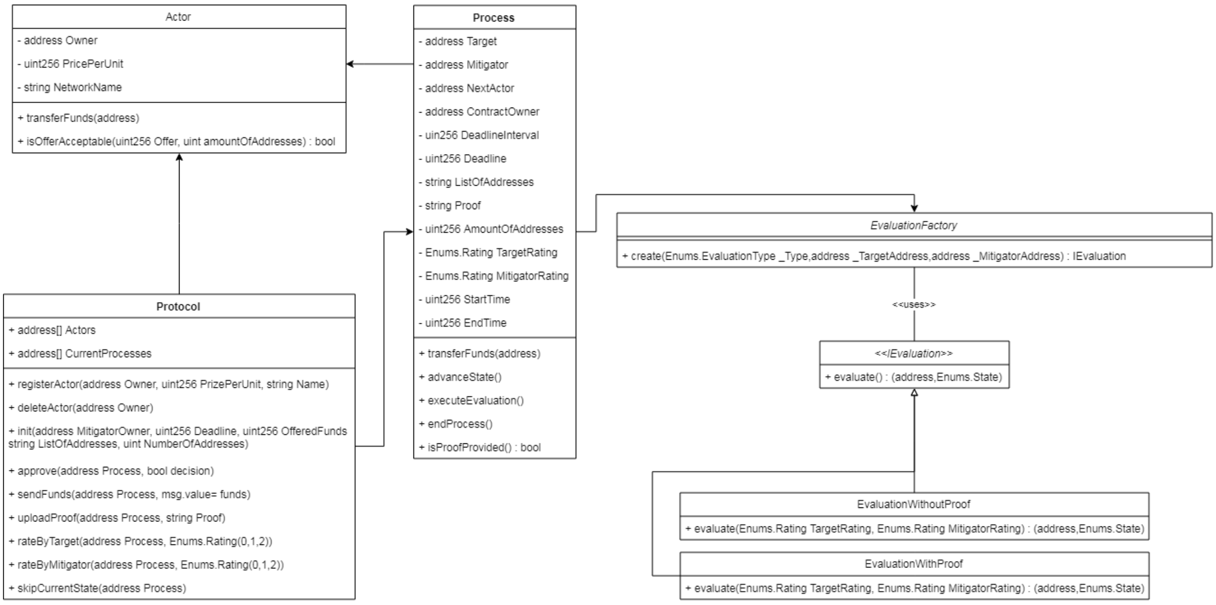
Actor
- address Owner
- uint256 PricePerUnit
- string NetworkName
+ transferFunds(address)
+ isOfferAcceptable(uint256 Offer, uint amountOfAddresses) : bool
Process
- address Target
- address Mitigator
- address NextActor
- address ContractOwner
- uin256 DeadlineInterval
- uint256 Deadline
- string ListOfAddresses
- string Proof
- uint256 AmountOfAddresses
- Enums.Rating TargetRating
- Enums.Rating MitigatorRating
- uint256 StartTime
- uint256 EndTime
+ transferFunds(address)
+ advanceState()
+ executeEvaluation()
+ endProcess()
+ isProofProvided() : bool
Protocol
+ address[] Actors
+ address[] CurrentProcesses
+ registerActor(address Owner, uint256 PrizePerUnit, string Name)
+ deleteActor(address Owner)
+ init(address MitigatorOwner, uint256 Deadline, uint256 OfferedFunds string ListOfAddresses, uint NumberOfAddresses)
+ approve(address Process, bool decision)
+ sendFunds(address Process, msg.value= funds)
+ uploadProof(address Process, string Proof)
+ rateByTarget(address Process, Enums.Rating(0,1,2))
+ rateByMitigator(address Process, Enums.Rating(0,1,2))
+ skipCurrentState(address Process)
EvaluationFactory
+ create(Enums.EvaluationType _Type,address _TargetAddress,address _MitigatorAddress) : IEvaluation
<<uses>>
<<IEvaluation>>
+ evaluate() : (address,Enums.State)
EvaluationWithoutProof
+ evaluate(Enums.Rating TargetRating, Enums.Rating MitigatorRating) : (address,Enums.State)
EvaluationWithProof
+ evaluate(Enums.Rating TargetRating, Enums.Rating MitigatorRating) : (address,Enums.State)

Figure 4.3: Architecture of the *simple* implementation

Chapter 5

Implementation

The implementation of the Cooperative Signaling Protocol prototype in this paper can basically be divided into four steps that led to the current final state. These are explained in more detail in the following sub-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 our 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, because 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 time and to test the first prototype there. This worked pretty well in the beginning and the first prototype could be created 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 a state pattern was already implicit. 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 byte code always exceeded the maximum size of 24 Kb due to the Inheritance used with the states. The individual states should 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 is illustrated in fig 5.1. Consequently, the process had to embed all states via import, which in turn meant that the bytecode of the process contract was always extended by the byte code of the states (which inherited from the abstract state). In the end, this resulted in a byte code that was too long for the entire contract, which ended in an OutOfGas exception. In addition, the evaluation algorithms were also defined as abstract contracts, which represented the same problem and inflated the bytecode of the process 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 a runtime environment. This virtual machine does not work directly with the solidity code, but with the compiled bytecode. The bytecode itself is a set of instructions for the virtual machine based on a strict technical specification.

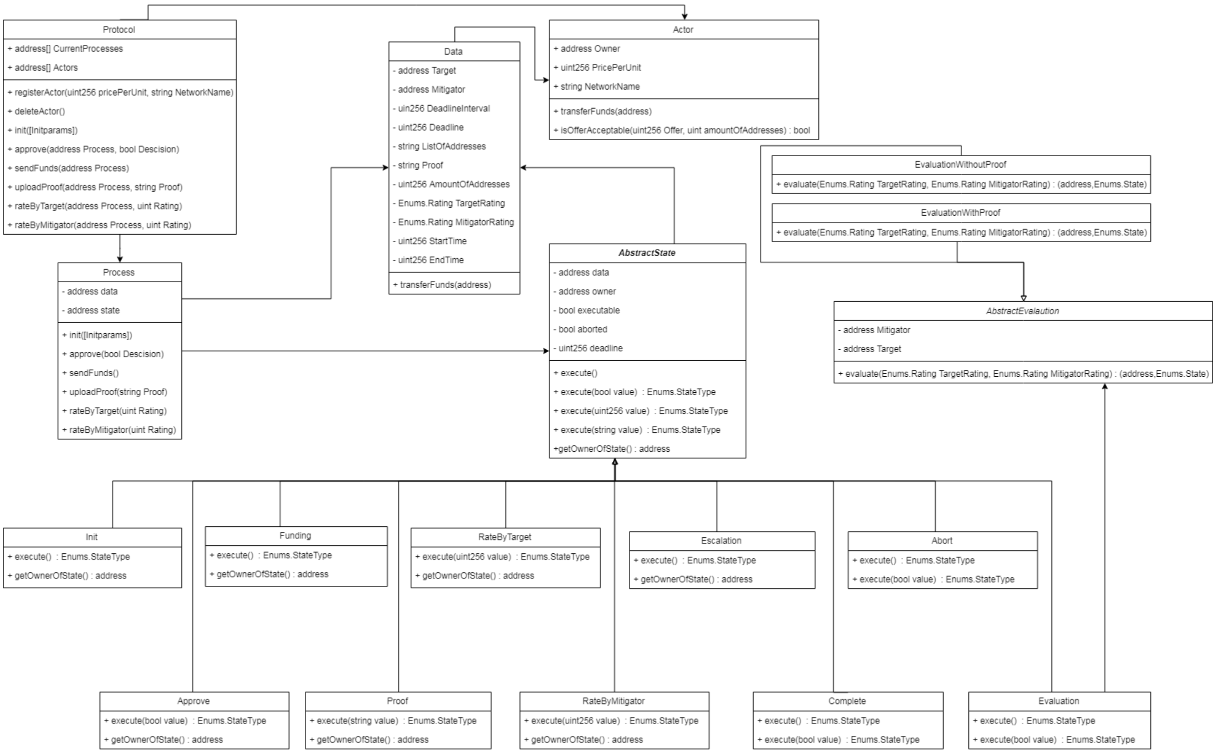
Protocol
+ address[] CurrentProcesses
+ address[] Actors
+ registerActor(uint256 pricePerUnit, string NetworkName)
+ deleteActor()
+ init([Initparams])
+ approve(address Process, bool Descision)
+ sendFunds(address Process)
+ uploadProof(address Process, string Proof)
+ rateByTarget(address Process, uint Rating)
+ rateByMitigator(address Process, uint Rating)
Data
- address Target
- address Mitigator
- uin256 DeadlineInterval
- uint256 Deadline
- string ListOfAddresses
- string Proof
- uint256 AmountOfAddresses
- Enums.Rating TargetRating
- Enums.Rating MitigatorRating
- uint256 StartTime
- uint256 EndTime
+ transferFunds(address)
Actor
+ address Owner
+ uint256 PricePerUnit
+ string NetworkName
+ transferFunds(address)
+ isOfferAcceptable(uint256 Offer, uint amountOfAddresses) : bool
EvaluationWithoutProof
+ evaluate(Enums.Rating TargetRating, Enums.Rating MitigatorRating) : (address,Enums.State)
EvaluationWithProof
+ evaluate(Enums.Rating TargetRating, Enums.Rating MitigatorRating) : (address,Enums.State)
Process
- address data
- address state
+ init([Initparams])
+ approve(bool Descision)
+ sendFunds()
+ uploadProof(string Proof)
+ rateByTarget(uint Rating)
+ rateByMitigator(uint Rating)
AbstractState
- address data
- address owner
- bool executable
- bool aborted
- uint256 deadline
+ execute()
+ execute(bool value) : Enums.StateType
+ execute(uint256 value) : Enums.StateType
+ execute(string value) : Enums.StateType
+getOwnerOfState() : address
AbstractEvalaution
- address Mitigator
- address Target
+ evaluate(Enums.Rating TargetRating, Enums.Rating MitigatorRating) : (address,Enums.State)
Init
+ execute() : Enums.StateType
+ getOwnerOfState() : address
Funding
+ execute() : Enums.StateType
+ getOwnerOfState() : address
RateByTarget
+ execute(uint256 value) : Enums.StateType
+ getOwnerOfState() : address
Escalation
+ execute() : Enums.StateType
+ getOwnerOfState() : address
Abort
+ execute() : Enums.StateType
+ execute(bool value) : Enums.StateType
Approve
+ execute(bool value) : Enums.StateType
+ getOwnerOfState() : address
Proof
+ execute(string value) : Enums.StateType
+ getOwnerOfState() : address
RateByMitigator
+ execute(uint256 value) : Enums.StateType
+ getOwnerOfState() : address
Complete
+ execute() : Enums.StateType
+ execute(bool value) : Enums.StateType
Evaluation
+ execute() : Enums.StateType
+ execute(bool value) : Enums.StateType

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

After some time and pondering, the writer decided to further develop the first prototype and to continue with it, 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 still an abstract contract, which again bloated up the bytecode unnecessarily. Still, the byte code 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.

5.4 Surrendering is Not an Option

After reading some posts in forums and blogs, the writer found the 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 an object with the interface type can be returned from the factory, which decouples the single components even more from each other. Or as it was written [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 no 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 protocol was 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.

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

Listing 5.1: ProcessCallState

```
import "./Enums.sol";
import "./IState.sol";
import "./StateProof.sol";
import "./StateRatingByTarget.sol";


library StateFactory{

    function create(Enums.StateType _Type,address payable data)
    public
  returns (IState){

    if(_Type == Enums.StateType.PROOF){
      return new StateProof(data);
    }else if(_Type == Enums.StateType.RATE_T){
      return new StateRatingByTarget(data);
    }else{
      revert("Type not in StateFactory");
    }
  }
}
```

Listing 5.2: StateFactory

```
import "./IState.sol";

contract StateProof is IState{

  address payable data;
  address payable owner;
  bool internal executable = true;
  bool internal aborted = false;
  uint256 internal deadline;

    constructor(address payable _data) public payable {
    data = _data;
    owner = IActor(IData(data).getMitigator()).getOwner();
    deadline = now + IData(data).getDeadlineInterval() * 1 seconds;
  }

  function execute(bool /*value*/) external returns(Enums.StateType) {
     revert("Not implemented");}
  function execute(uint256 /*value*/) external returns(Enums.StateType) {revert("Not implemented");}

  function execute() external returns(Enums.StateType) {
    require(executable,"Process not executable");
    if(canBeSkipped()){
      executable=false;
      return Enums.StateType.RATE_T;
    }else{
      require(owner == tx.origin,"Error owner != tx.origin");
    }
    executable=false;
    return Enums.StateType.RATE_T;
  }

  function execute(string calldata value) external returns(Enums.StateType){
        require(executable,"Process not executable");
    if(canBeSkipped()){
      executable=false;
      return Enums.StateType.RATE_T;
    }else{
      require(owner == tx.origin,"Error owner != tx.origin");
    }
        IData(data).setProof(value);
    executable=false;
    return Enums.StateType.RATE_T;
    }

  function canBeSkipped() private view returns(bool){
    if(now>deadline){return true;}
    return false;
  }

  function abort() public returns(Enums.StateType){
    require(owner == tx.origin,"Error owner != tx.origin");
    aborted=true;
    executable = false;
    return Enums.StateType.RATE_T;
  }

   function getOwnerOfState() external view returns(address payable){return owner;}

   function getStateType() external view returns(Enums.StateType){return Enums.StateType.PROOF;}

}
```

Listing 5.3: ConcreteState

Chapter 6

Evaluation

Chapter 7

Summary and Conclusions

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

[2] Andreas GrÃ¼hler, Bruno Rodrigues: *A Reputation and Reward Scheme for a Cooperative, Multi-domain DDoS Defense*, UniversitÃ¤t ZÃ¼rich, Communication Systems Group, Department of Informatics, ZÃ¼rich, 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

Abbreviations

6LowPAN	IPv6 over Low power Wireless Personal Area Networks
BPMN	Business Process Model and Notation
CoMaDa	Configuration, Management, Data Handling
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheet
DDoS	Distributed Denial of Service
IEEE	Institute of Electrical and Electronics Engineers
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IoT	Internet of Things
IP	Internet Protocol
IPFIX	IP Flow Information Export
LoWPAN	Low-Power Wireless Personal Area Network
OS	Operating System
TCP	Transmission Control Protocol
JS	JavaScript
JSON	JavaScript Object Notation
PHP	PHP: Hypertext Preprocessor
REST	Representational State Transfer
TLS	Transport Layer Security
TUM	Technische Universität München
UI	User Interface
SSL	Secure Sockets Layer
URL	Uniform Resource Locator
UZH	University of Zurich
UDP	User Datagram Protocol
WebMaDa	Web-based Mobile Access And Data Handling
WPAN	Wireless Personal Area Network
WSN	Wireless Sensor Networks

Glossary

ABI Application Binary Interface

List of Figures

List of Listings